MSc IT (S3) 02
Exam. Code:Exam. Code:M3OOP

Object-Oriented Programming though C++

SEMESTER-I

MASTER OF SCIENCE IN INFORMATION TECHNOLOGY

BLOCK- 2



KRISHNA KANTA HANDIQUI STATE OPEN UNIVERSITY

Subject Experts

Prof. Anjana Kakati Mahanta, Gauhati University

Prof. (Retd.) Pranhari Talukdar, Gauhati University

Dr. Jyotiprokash Goswami, Assam Engineering College

Prof. Shikhar Kumar Sarma, Gauhati University

Programme Co-ordinator

Dr. Ridip Dev Choudhury, KKHSOU

Course Co-ordinators :

Dr. Ridip Dev Choudhury, KKHSOU

Dr. Tapashi Kashyap Das, KKHSOU

Dr. Sruti Sruba Bharali, KKHSOU

SLM Preparation Team

UNITS	CONTRIBUTORS
8	**Dr. Amitabha Nath**, NEHU
9, 11	**Dr. Tapashi Kashyap Das**, KKHSOU
10	**Mr. Dwipen Laskar**, Gauhati University
12	**Ms. Jonalee Barman Kakati**, NERIM
13, 14	**Dr. Sruti Sruba Bharali**, KKHSOU
15	**Ms. Sangeeta Borkakoty**, USTM

Editorial Team

Content : **Dr. Swapnanil Gogoi**, IDOL , Gauhati University

Structue, Format & Graphics: **Dr. Tapashi Kashyap Das**, KKHSOU

January, 2022 **ISBN: .....**

This Self Learning Material (SLM) of the Krishna Kanta Handiqui State Open University is made available under a Creative Commons Attribution-Non Commercial-Share Alike 4.0 License (international): http://creativecommons.org/licenses/by-nc-sa/4.0/

Printed and published by Registrar on behalf of the Krishna Kanta Handiqui State Open University

Headquarters: Patgaon, Rani Gate, Guwahati-781017

City Centre: NH 37, Resham Nagar, Khanapara, Guwahati-781022; Web:www.kkhsou.in

MASTER OF SCIENCE IN INFORMATION TECHNOLOGY

Object-Oriented Programming through C++

Block-2

DETAILED SYLLABUS

		Pages
UNIT 8 :	**Introduction to Classes and Objects [6 marks]** Classes in C++, class declaration, declaring objects, Defining Member functions, Inline member function, Array of objects, Objects as function argument, Static data member and member function, Friend function and friend class.	**7-37**
UNIT 9 :	**Constructors and Destructors [6 marks]** Constructors, Instantiation of objects, Default constructor, Parameterized constructor, Copy constructor and its use, Destructors, Constraints on constructors and destructors, Dynamic initialization of objects.	**38-63**
UNIT 10:	**Operator Overloading [10 marks]** Overloading unary operators: Operator keyword, arguments and return value; Overloading Unary and binary operators: arithmetic operators, manipulation of strings using operators, Type conversions.	**64-95**
UNIT 11:	**Inheritance [8 marks]** Derived class and base class: Defining a derived class, Accessing the base class member, Inheritance: multilevel, multiple, hierarchical, hybrid; Virtual base class, Abstract class.	**96-131**
UNIT 12:	**Virtual Functions and Polymorphism [10 marks]** Virtual functions, Pure virtual functions; Polymorphism, Categorization	**132-149**

of polymorphism techniques: Compile time polymorphism, Run time polymorphism.

UNIT 13: File Handling [8 marks] **150-168**

File classes, Opening and Closing a file, File modes, Manipulation of file pointers, Functions for I/O operations

UNIT 14: Templates [8 marks] **169-181**

Introduction, Function template, Function template overloading, Class templates, Template arguments.

UNIT 15: Object Oriented Design [6 marks] **182-209**

Object Oriented Design Approaches, Object Modeling Techniques (OMT) tools: Object Model, Dynamic Model, and Functional Model. (Object Diagram, State Diagram, and DFD), Phases of Object-Oriented Development: Object Analysis, System Design, Object Design.

BLOCK INTRODUCTION

The second block comprises of eight units. This block provides an insight of various concepts related to Object-Oriented Programming. The second block is organised as follows:

Unit 8 is the first unit of this block and it explains the concept of class which is the most important features of C++. How classes are declared and defined, creation of object are discussed in this unit. The unit also discusses how to write member functions, inline function, friend function, friend class at the end.

Unit 9 is about constructors and destructors. Different rules of writing constructors and destructors, different types of contructors and dynamic initialization of objects are discussed in this unit.

Unit 10 discusses another most important features of C++ language, i.e., operator overloading. It contains overloading of unary and binary operators and concept of typeconversion at the end.

Unit 11 covers another important concept of OOP, i.e., inheritance. How to derive a new class from an existing class is discussed in this unit. The effect of access specifiers private, public, and protected in inhertitance is explained. Different types of inheritance and concept of virtual base class and abstract class are narrated with examples.

Unit 12 discusses how to write virtual functions in C++. Complile time and run time polymorphism are explained with example in this unit.

Unit 13 discusses how to work with files in C++. In this unit various methods available for storing and retrieving the data from files with different file modes are discussed.

Unit 14 is on templates. Both function templates and class templates with examples are discussed in this unit.

Unit 15 is the last unit of this course which discusses Object-Oriented design, its approaches, techniques, phases and different models.

The structure of Block 2 is as follows:

UNIT 8 : Introduction to Classes and Objects
UNIT 9 : Constructors and Destructors
UNIT 10 : Operator Overloading
UNIT 11 : Inheritance
UNIT 12 : Virtual Functions and Polymorphism
UNIT 13 : File Handling

UNIT 8: INTRODUCTION TO CLASS AND OBJECTS

UNIT STRUCTURE

8.1 Learning Objectives

8.2 Introduction

8.3 Classes in C++

8.4 Class Declaration

8.5 Declaring Objects

- 8.5.1 Object Memory Allocation
- 8.5.2 Class Member Access Operator

8.6 Defining Member functions

- 8.6.1 Function Definition Inside the Class
- 8.6.2 Function Definition Outside the Class

8.7 Array of Objects

8.8 Objects as Function Argument

8.9 Static Data Member and Member Function

- 8.9.1 Static Variable
- 8.9.2 Static Function

8.10 Friend Function and Friend Class

8.11 Let Us Sum Up

8.12 Further Reading

8.13 Answers to Check Your Progress

8.14 Model Questions

8.1 LEARNING OBJECTIVES

After going through this unit, you will be able to –

- learn about class, object, and their importance
- learn how to use class and object in C++
- declare data members and member functions in a class.
- learn how to use an inline function

- learn how to create array of objects
- define static function and variable
- use friend functions in different situations.
- learn the concept of friend class

8.2 INTRODUCTION

In the previous unit, we have learned about function and various technical nuances of using a function. In this unit, we will know about class and object. *Class* in C++ can be related to *structure* of C programming language. Structure provides a great way of handling complex attributes by grouping different data types into one unit. Once a structure is defined, it can be used to create variables of that type. Structure is useful but it suffers from certain limitations such as:

- It does not permit data hiding. Structure cannot limit its members from free access to outside functions.
- Structures do not allow the inclusion of functions as part of its member.
- It does not allow struct data type to be used just like any other built-in data type.

A class in C++ intends to overcome these limitations by extending the features of existing structures to cater the requirements of Object-Oriented Programming (OOP). We will shortly find out how a class can restrict outside access to its members and enable us to use itlike any other primitive datatypes.

In principle, abstraction and encapsulation are the two most important features of Object-oriented programming. Abstraction helps in managing problem complexity by hiding underlying details. Whereas, encapsulation helps in shielding the class members from unauthorized access. All other features of OOP (e.g., inheritance, polymorphism) essentially revolve around these two only.Class holds the key to implementing these two concepts in object-oriented programming. It works as a user-defined data type that

contains a collection of variables and functions. These variables and functions define the properties and capacities of any class. If we want to access any class members, one has to instantiate an object of that class. This is a pretty good testimony about class and its importance in OOP. In fact, it's so important that Bjarne Stroustrup initially thought of keeping the name as *C with classes*. Later on, the name was eventually changed to C++. More on class and object will be covered in the upcoming sections.In the next unit, we will explore yet another important concept known as constructors and destructors. It is going to be an addon to the current discussion of class and object.

8.3 CLASSES IN C++

A class is a user-defined data type. It wraps up data and functions together into one unit. An object is an instance of that class.Therefore, a class can be viewed as a blueprint and an object as its realization.

A class definition typically contains two things. The first is a list of data members and the second is a list of member functions. Data members specify the properties whereas the member functions define the capacities of a class. The relationship between a class and objects can be understood using a conceptual diagram shown in Figure 8.1.

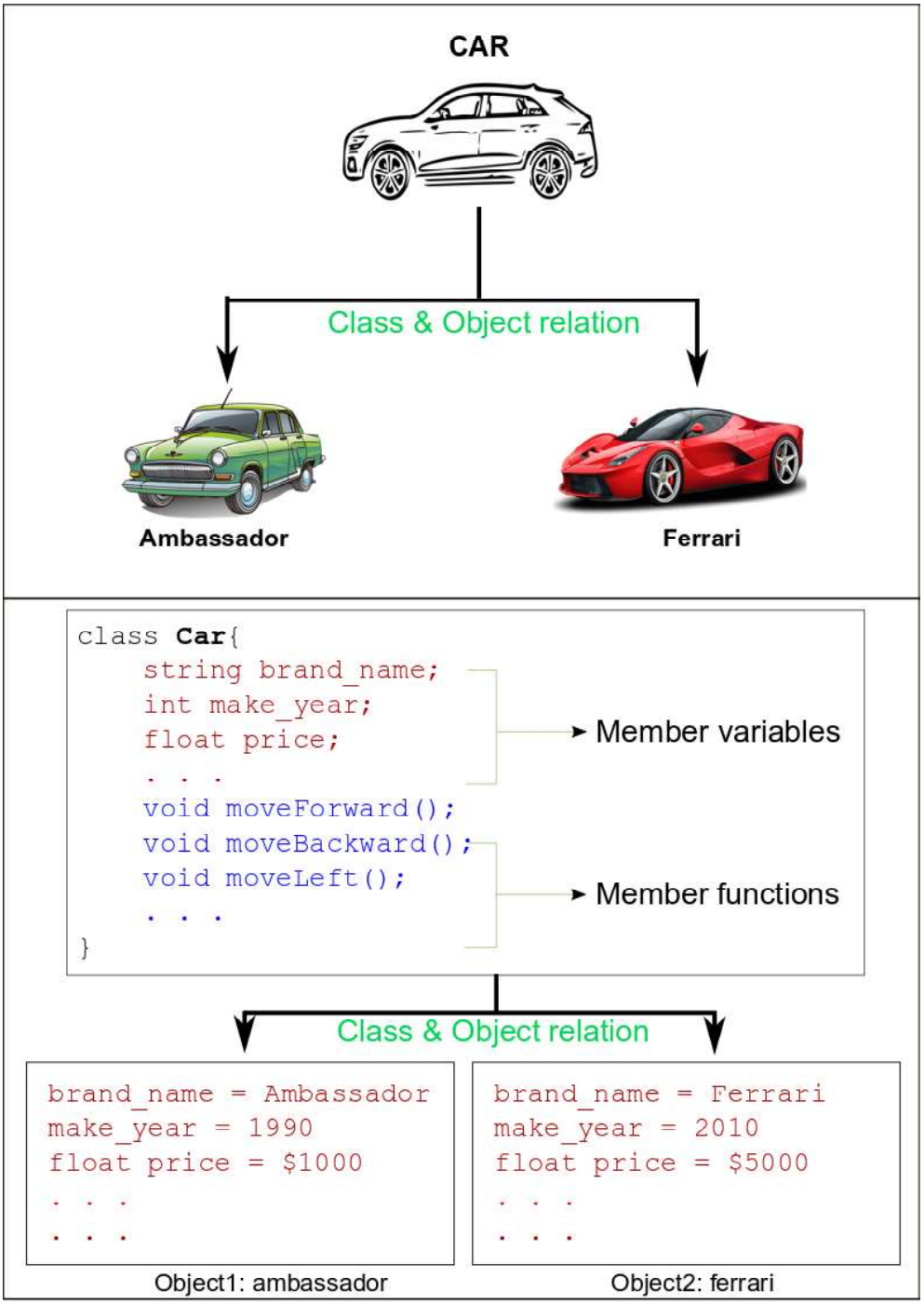


Figure 8.1 Class and object relationship

In the example shown in Figure 8.1, car represents the class and its objects are Ambassador and Ferrari respectively. Variables such as `brand_name`, `make_year`, and `price` are the data members. These member variables specify the property of a car. For instance, a 1990 model Ambassador car

cost $1000 and a 2010 model Ferrari carcosts $5000 each. Similarly, *moveForward(), moveBackward(),* etc. are the member functions that define the capacity of a car. It means these cars have the capacity to move forward, backward, and left side respectively. Hence, we can see that car acts as a concept or a blueprint whereas ambassador and Ferrari represent a specific instance or object.

8.4 CLASS DECLARATION

In C++, classes are declared using the keyword class followed by the class name. The class has a body that contains declarations of member variables and functions of any valid type. Following is an example of how we define a simple class named **Demo**.

```
#include<iostream.h>
class Demo
{
      private:
            // Private data and methods
             ---
      public:
           // Public data and methods
             ---
};
```

Class boundary terminated by a ";"

As we can see, our example class starts with the keyword *class* then followed by the user-defined class name *Demo*. The block marked with a pair of curly braces represents the body of the class. It is important to note that C++ class terminates with a semicolon. Class members can be divided into two logical parts such as (i) private members, and (ii) public members. Private members are not available for external access. It means any class members declared under private mode can only be accessed through functions available within the class. On the other hand, public members are open to access by any external function. Here, keywords private and

public are known as visibility labels. The decision to use these visibility labels depends on the discretion of the programmer. However, there are some points one has to remember before defining a class:

- The default visibility label of a class is private. It means, if we do not include any label explicitly then all the class member becomes private automatically.
- We can use these labels as many times as we want inside the class body. The only thing we have to remember is that a portion marked as private/public is valid until the next private/ public label occurs.

There is one more visibility label known as *protected* available in C++. But that will be covered when we discuss inheritance in detail.

8.5 DECLARING OBJECTS

One of the objectives of OOP is to minimize the gap between the way we use user-defined and primitive data types. For instance, C allows us to define and use user-defined Data types (e.g., structure) much like primitive data types but not always. E.g.,if we consider a structure *animal* in C then creating a variable of type *animal* using the following statement would give us an error as this is not the correct way of creating a structure type variable. However, we can always use this approach to create a variable of any primitive data type (e.g., int val;).

```
animal tiger;   // Where animal is structure in C
```

Declaring an object in C++ follows the same rule as declaring a variable of any built-in datatype. As a consequence, we can use the same statement shown above in C++ without any error. Likewise, object declaration also requires type and an object name pair in C++.

Syntax: `class_nameobject_name [, object_name1,...];`
Where `class_name` is the name of any existing class and `object_name` is the variable/object name given by the programmer. For example, the following statement defines an object *ambassador* of type *Car*:

Example: `Car ambassador;`

Once the object is created memory also gets allocated for the data members of the current object. The current object (ambassador) contains the members like *brand_name*, *make_year*, and *price*.

8.5.1 Object Memory Allocation

A class on its own represents the concept or it works as a blueprint. Therefore, when a class is defined, it does not mean that an object of that class is created. Hence, memory allocation for the objects does not happen at this stage. The actual allocation of memory only takes place when an object of that class is declared. However, this argument is not entirely true if we consider member functions in our discussion. Consider the diagram given in Figure 8.2 to understand this fact.

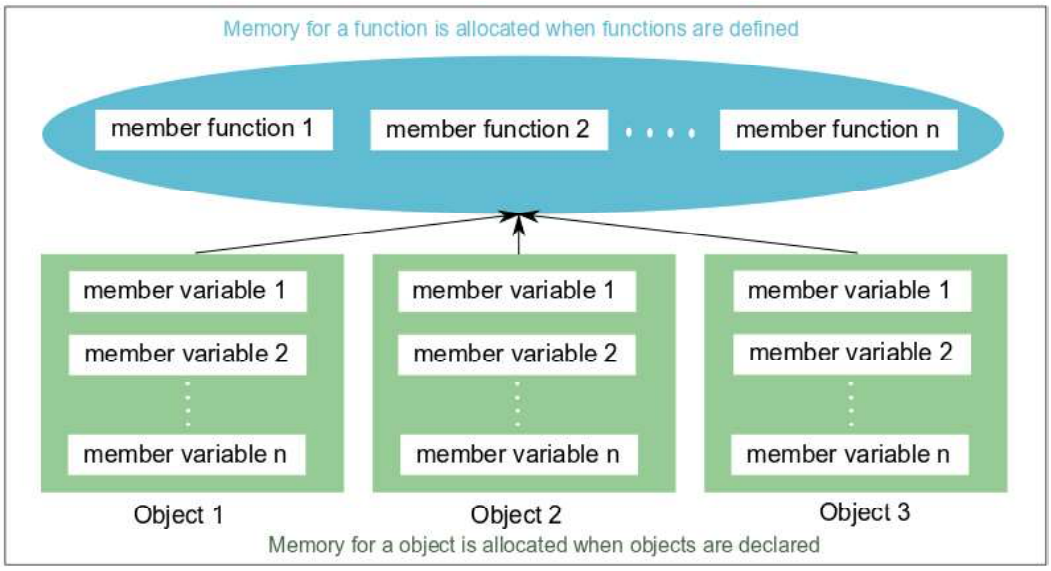


Figure 8.2 Object memory allocation

As we can see from the figure above, the memory allocation of class member happens in two different ways. Memory for member functions gets allocated when these functions are defined and it is done only once. Whereas, memory allocation for objects is done only when objects of class are created and not when a class is defined. If we consider a class and three of its objects then the

assignment of memory would look something like as shown above. All the objects will have separate address spaces as they are different and represent a different entity. However, functions need not be stored separately as these do not correspond to any distinctive features of the object. These functions can be viewed as a common tool to manipulate the local data. Therefore, there is no harm in storing functions in one common place and providing access to these functionsfrom all objects of that class.

8.5.2 Class Member Access Operator

A class typically contains some variables and functions arranged under different visibility mode. Public members are open for access tonon-member functions. However, the access request has to go through the object of that class. We can think of an object as the permission token to access its members. Therefore, once we have created an object then we can use it with the *class member access operator,* "." (dot operator) to accessany member.

Example:

```
Car ambassador;
ambassador.moveForward();
```

In the above statement, object *ambassador* is created and used to call or access the *moveForward()* function. Hypothetically, executing this function will enable our car to move forward. Similarly, we can use this dot operator to access a member variable just like we accessed a member function earlier.

Private members on the other hand are hidden from the outside world. Direct access to these members from outside of class is forbidden in OOP. This is to ensure the safety and security of private data. Therefore, if a member is private in nature, then dot operator cannot be used to access that member.

Example: `cout<<ambassador.price;` //printing value of price from an outside function

If we consider price as a private member of our class *Car*, the above statement will result in an error. So, does it mean that private members are inaccessible to the outside world? The answer is no. A private member can be accessed using the member functions of that class. This way we can safeguard private members and also facilitate access if required.Figure 8.3 will help you to understand this fact in more detail.

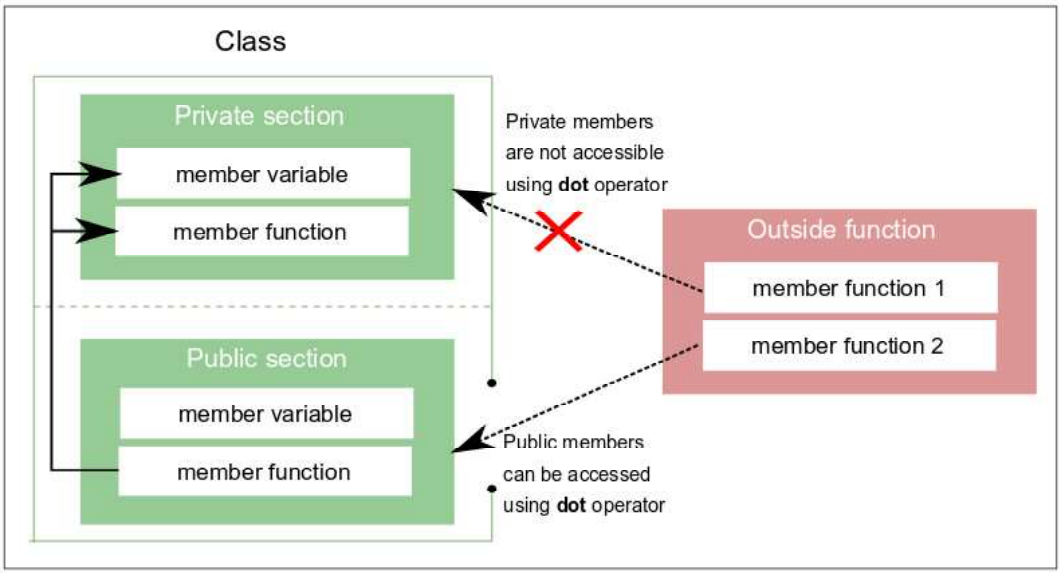


Figure 8.3 Class members and visibility modes

8.6 DEFINING MEMBER FUNCTIONS

A member function can be defined in two different ways.

- Function definition inside the class
- Function definition outside the class

The choice of a particular approach depends solely on the programmer. However, some people feel that defining member functions outside the class helps in keeping the class definition simple. Whatever the approach may be,the body of the function remains the same and only the definition part changes a little bit. We shall see both cases in the following sections.

8.6.1 Function Definition Inside the Class

In this case, a function is defined inside the class boundary itself. Usually, this approach is chosen when the class has a limited number of comparatively smaller functions. In such cases often programmers define the functions inside the class definition. A simple code block to explain its usage is given below.

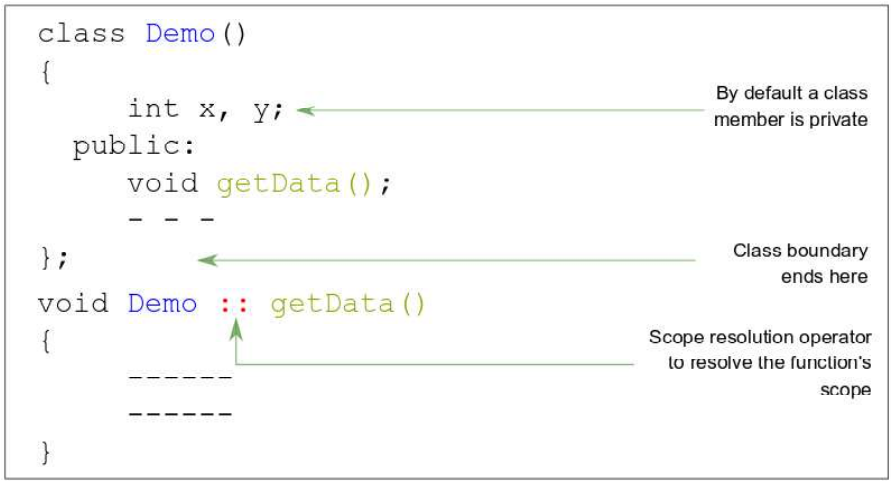
```
class Demo {
        ---
    public:
        void getData()
        {
              ----
        }
};
```

Here, the *get Data()* function is defined within the class boundary. A member function that is defined inside its class boundary is called an *inline member function*. In general, member functions having only a few lines of codes are declared inline. A function defined outside the class boundary can also be made inline but in that case, keyword *inline* needs to be included in the function definition.

8.6.2 Function Definition Outside the Class

Declaration of a member function has to be done inside the class boundary. However, the definition part can be placed anywhere in the program i.e., inside or outside of the class. Often people prefer to place function definition outside of the class boundary. It's not mandatory to do so but as mentioned earlier this approach can improve the readability of the code. There are not many differences between this and the earlier approach. The only difference lies with the fact that if we are taking the function definition out of the block

where it is declared then how do we resolve its scope. So, to get around this problem we use something known as *scope resolution operator "::".* It basically informs the compiler that the function belongs to a particular class. To understand the scope resolution operator and its use, have look at the code snippet given below.



As we can see from above, the function *get Data()* is declared inside the class but its definition lies outside. Therefore, to resolve this scope issue, C++ allows us to use the scope resolution operator while defining a member function outside its class boundary. Here in this case "::" signifies that *get Data()* belongs to *Demo* class. The function definition syntax for outside definition is:

```
return_type class_name :: function_name (args){}
```

Here, the extra information i.e., "`class_name`" and "`::`" helps the compiler to know about the actual scope of *get Data()* function.

Program 8.1: C++ program to demonstrate object and functions

```
#include <iostream.h>

//Class specification
class Dummy
{
      int intval;
      float floatval;
      public:
  void get Data() //Function input user data
      {
```

```
      cout << "Enter the integer value: ";
              cin >> intval;
      cout << "Enter the floating-point value: ";
              cin >> floatval;
      }
      void putData(); //Function output user data
      };

//Function defined outside of class boundary
      void Dummy :: putData()
      {
      cout << "\nEntered integer value is:"
      << intval << endl;
      cout << "Entered floating-point is:"
           << floatval << endl;
      }
      int main()
      {
Dummy obj; //Object of class Dummy created

      obj.getData();    //calling getData() using
object obj
      obj.putData();    //calling putData() using
object obj
      return 0;
}
```

Program 8.1 presents a simple program to demonstrate how an object of a class can be created and used to call a member function. It also shows the inside and outside definition of a member function. In this case, *getData()*, *putData()* are two member functions where the former is defined within the class itself whereas the latter is defined outside the class. One important thing to remember is that calling a member function has to be done using the object and not making any direct call. Not doing so would result in a compilation error. The output of the above program is given below

```
Enter the integer value: 5
Enter the floating-point value: 2.5

Entered integer value is: 5
Entered floating-point value is: 2.5
```

8.7 ARRAY OF OBJECTS

OOP treats user-defined datatypes similar to other built-in datatypes. It means, if we can declare an array of built-in data types (e.g., int) then we should also be able to declare an array of user-defined data types such as class. Now considering a class named Student, we can create as many objects of type Student we want. For an instance, if there is a class of 30 students then we can create 30 objects of class Student where each object represents one particular student. But, creating 30 objects of class Student one by one is tedious work. So, we can declare a *Student* type array of size 30 and accommodate all objects into that array. This concept is basically known as array of objects. Program 8.2 shows how to use an array of objects.

Program 8.2: C++ program to demonstrate array of objects

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

//Class specification
class Student{
    int roll;
    string name;
    float marks;
  public:
    void getData(int); // Inline definition is also
possible
    void putData(int);
};

//Function to get and store student information
void Student :: getData(int i)
{
cout<<"\n Enter information of "<<i<<"th student:\n";
cout<< "\nEnter your Roll: ";
    cin >> roll;
cout<< "\nEnter your name: ";
    cin >> name;
cout<< "\nEnter your marks: ";
    cin >> marks;
}

//Function to print student information
void Student :: putData(int i)
```

```
{
cout<<"\n Records of "<<i<<"th student:\n";
cout<< "\nRoll is: " << roll << endl;
cout<< "\nName is: " << name << endl;
cout<< "\nMark is: " << marks << endl << endl;
}
int main()
{
    Student st[30];     // Creating an array of type student

    for(int i=0; i<30; i++)
    {
st[i].getData(i+1);    // calling getData() of ith object
    }

    for(i=0; i<30; i++)
    {
    st[i].putData(i+1);    // calling putData() of ith object
    }
    return 0;
}
```

Here in this program, we have created a class Student with three attributes namely roll, name, and marks. It also contains two functions designed to accept input from users and print user information onto the console. Inside the main() function an array of type Student is created to accommodate 30 students. Once the array is created, a loop is used to iterate over the array for storing every student's information accordingly. The same looping approach is used again to print the student information back onto the console. For simplicity, the program is executed with an array size of 2, and the output is shown below.

```
Enter information of 1th student:

Enter your Roll: 1
Enter your name: a
Enter your marks: 90

Enter information of 2th student:

Enter your Roll: 2
Enter your name: b
Enter your marks: 100

Records of 1th student:
```

```
Your Roll is: 1
Your Name is: a
Your Mark is: 90

Records of 2th student:

Roll is: 2
Name is: b
Mark is: 100
```

8.8 OBJECTS AS FUNCTION ARGUMENT

We know that functions communicate through message passing. A message in a typical C++ program is nothing but a value stored in a variable. Therefore, when a function wants to communicate, it makes use of the argument section of a function to pass messages. Till now, we only saw variables of built-in datatype being used as arguments. Here in this section, we shall see objects being used as function arguments. Our primary focus will be on:

- Passing copy of an object as argument
- Passing address of an object as argument
- Passing reference of an object as argument

Program 8.3: C++ program to use object as argument

```
#include <iostream.h>

//Class specification
class Temperature {
    float fah;
    float cel;
  public:
    void getData();
    void convert(Temperature);  //Class type
argument
};

//Function to get temperature in Fahrenheit
void Temperature :: getData()
{
    cout << "Enter temperature in Fahrenheit: ";
cin>> fah;
}
```

```
//Function to convert temperature to celsius
void Temperature :: convert(Temperature tmp_far)
{
cel = (tmp_far.fah - 32) * 5/9; //Conversion formula
    cout << "Temperature in Celsius= " << cel <<
endl;
}

int main()
{
    Temperature temp;// Creating an object of
Temperature class

    temp.getData();  // Calling function to input
data
    temp.convert(temp);    //Calling convert function
for conversion
    return 0;
}
```

Output:

```
Enter temperature in Fahrenheit: 100
Temperature in Celsius= 37.7778
```

Program 8.3 can help us to understand the way we can pass a copy of an object to a function. Inside the main function, an object `temp` of type `Temperature` is created. Using this object first the `getData()` function is called to input temperature in °Fahrenheit value. Once the value is ready with us we can call `convert()` through the use of temp object. The important thing to note here is that the `convert()` accepts an argument of type `Temperature`. So, basically, we are passing the object temp as an argument to the `convert()` function. The function then converts the value into °Celsius form and prints the output onto the screen.

In the second case, the address of the object can be used as an argument. This is different than passing copy as an argument because any modifications by the called program get reflected on the original variable or object. The same program can be modified and adapted to reflect this concept. The modified version is shown in program 8.4.

Program 8.4: C++ program to pass address of object as argument

```
#include <iostream.h>
//Class specification
class Temperature {
    float fah;
    float cel;
  public:
    void getData();
    void convert(Temperature *);     //Pointer to
Class type argument
};

//Function to get temperature in Fahrenheit
void Temperature :: getData()
{
cout<< "Enter temperature in Fahrenheit: ";
cin>> fah;
}

//Function to convert temperature to celsius
void Temperature :: convert(Temperature *tmp_far)
{
cel = (tmp_far->fah - 32) * 5/9;     //Conversion
formula
cout<< "Temperature in Celsius= " <<cel<<endl;
}

int main()
{
    Temperature temp;//  Creating an object of
Temperature class

temp.getData(); // Calling function to input data
temp.convert(&temp); //Calling convert function for
conversion
    return 0;
}
```

Similarly, in the third case, the reference of an object can also be passed as an argument to a function. In program 8.5, reference of the object is passed to the function convert`()`.

Program 8.5: C++ program to pass reference of object as argument

```
#include <iostream.h>

//Class specification

class Temperature {
```

```
    float fah;
    float cel;
  public:
    void getData();
    void convert(Temperature &);     //Reference to
Class type argument
};

//Function to get temperature in Fahrenheit
void Temperature :: getData()
{
cout<< "Enter temperature in Fahrenheit: ";
cin>> fah;
}

//Function to convert temperature to celsius
void Temperature :: convert(Temperature &tmp_far)
{
cel = (tmp_far.fah - 32) * 5/9; //Conversion formula
cout<< "Temperature in Celsius= " <<cel<<endl;
}

int main()
{
    Temperature temp;//  Creating  an  object  of
Temperature class

temp.getData(); // Calling function to input data
temp.convert(temp);  //Calling convert function for
conversion
    return 0;
}
```

A function can also return an object as well. Here in the program the return type is void but had it been Temperature for say, then it is possible to return a Temperature type object or variable from that function. This gives great flexibility in dealing with user-defined type variables.



EXERCISE

1. Modify Program 8.3 to demonstrate how an object can be returned from a called function.
2. Write a C++ program to implement call by value and call by reference concept where the argument to be passed is an object of a given class.



CHECK YOUR PROGRESS

Q 1: A variable of a class is called as_____.

(a) object
(b) function
(c) data
(d) None of the above

Q 2: ____ of a class cannot be accessed outside the class.

(a) public members
(b) private members
(c) both (a) and (b)
(d) None of the above

Q 3: Functions declared and defined in a class is known as member functions.(True/False)

Q 4: The default visibility mode of a member in a class is public.(True/False)

Q 5: We can use ____ operator to access member functions of a class with its objects.

Q 6: Member functions of a class can be defined outside the class boundary using ___operator.

8.9 STATIC DATA MEMBER AND MEMBER FUNCTION

In C++, static keyword is used to render special characteristics to a program element. The element can be a member variable or a member function. When a member of a class is declared as static, only one copy of that member is created and it remains common to all the objects of that class no matter how many objects are created. Let us try to understand each one of these two separately.

8.9.1 Static Variable

A class variable declared as static is created only once and it is initialized to zero automatically. The scope of a static variable is limited to the block in which it is declared but its lifetime is the entire program. This means that once a variable is declared as static it retains its value till the termination of the program. Therefore, every time program control reenters the block of a static variable, it will find the old value that was assigned to it. This is in contrast with any other non-static variable and thereby makes it a special type of variable. Let us consider the following statements to understand the declaration and initialization ofa static variable in a class.

```
   class Temp
   {
static intnum; // Declaration
   } ;
 intTemp :: num = 9; //  Initialization
```

Here, Temp is a class where num is declared as a static variable. Outside the class boundary, num is initialized using scope resolution operator.

8.9.2 Static Function

Similar to a static variable, a static function is also very special. It possesses certain features which are different from non-static functions. A static function can access other static members only.

This includes variables or functions. If we try to call or access non-static members from a static context then the compiler will throw an error at us. Therefore, one has to be very careful with the usage of static variables as well as functions. The syntax for defining and calling a static function is:

```
Syntax: static ret_type function_name(){}
// Definition
Syntax: class_name :: function_name();
// Function call
```

The first statement uses the static keyword to define a static function. The rest of the codes are the same as that of a no-static function. However, calling a static function is slightly different. Here, we do not use object name and dot operator combination to call a static function. Instead, we use the class name and the scope resolution operator for making a call.

Implementation of both the concepts is presented using a conceptual file downloader. Program 8.6 can help us to understand this better.

Program 8.6: C++ program to demonstrate use of static variable

```
#include <iostream.h>
class Static_Demo
{
     //static variable    declaration
static int objectCount;
public:
//Function to implement file download
void download()
{
     //increament every time download
happensobjectCount++;
cout << "File downloaded" << endl;
}
```

```
//Static function to print total download count
static void printCount()
{
cout<< "\nYour file has been downloaded ";
cout<<objectCount<< " times" << endl;
}
};
// Initialization of static member
int Static_Demo::objectCount = 0;
int main(void)
{
Static_Demo demo;     // Create the object
for(int i=0; i<3; i++)
{
//   Call   the   download()   using   the
objectdemo.download();
}
//Calling the static function
Static_Demo::printCount();
return 0;
}
```

In this program, we have a class with a static variable and two member functions for taking care of file download and counting the number of downloads respectively. Inside the main function, we have employed a loop to simulate multiple attempts in downloading the file. Here, the *for-loop* iterates three times and makes call to `download()` for downloading a file from the server. Every time the `download()` function executes, it increments the value of the variable `objectCount`. Now, as the `objectCount` variable is a static variable, subsequent incrementing would actually increment the old value. This is the reason why we can have a function like `printCount()` to keep track of the total number of downloads that took place. Moreover, we can see that `printCount()` is a static

member function, and therefore, call to this function is different compared to the call to `download()` function. The output of the above program is given below.

```
File downloaded
File downloaded
File downloaded

Your file has been downloaded 3 times
```

To summarize, here are some of the features of a static member:

- A static variable is initialized to zero when the first object of that class is created.
- Only one copy of static type variable is created and shared among all the objects of the class.
- The scope of static variable is limited to its block but its lifetime is the entire program.
- A static function is allowed to access a static member only.
- A static function is called using the class name instead of the object name.

8.10 FRIEND FUNCTION AND FRIEND CLASS

Private members are very much restrictive and cannot be accessed from outside of a class. Only the member function of that class can access and manipulate those data. This is what we have been experiencing till now. However, this argument is not entirely true. Sometimes we might want to allow an outside function to access private data. For such cases, C++ provides analternative way in the form of “friend” to make this happen. When an external function becomes the friend of a class, it starts enjoying the same privileges that a normal member function does. For example, consider **Clothing** and **Grocery** being the two classes of a shopping mall inventory system. There is function named **GST()** that calculates the tax levied on a particular customer. In this example scenario, it would be beneficial if we make this **GST()** functionas friend to both classes. This way the function

would be able to access not only the public but also the private members of both classes. To make an outside function friend to a class, all we need to do is use the keyword **friend** in declaring the function as shown below.

```
class class_name
{
    ——
public:
    ——
    friend void function_name();
};
```

As we can see from the syntax given above, keyword **friend** is used to declare a friend function. The function can be a global function or a member function from a different class. The definition of the function can appear anywhere in the program just like any other regular function. However, it is important to note that while defining a friend function neither class name with scope resolution operator, nor the keyword “friend” is used. The following program will help you to understand this better.

Program 8.7: Program to demonstrate usage of friend function

```
// A global function as friend function
#include <iostream.h>
using namespace std;

class Person
{
private:
    string name;
    float salary;
public:
    void getData();
   friend void display(Person obj); //Friend function
};

//Member function to input data from user
void Person :: getData()
{
    cout << "Enter your name: ";
    cin >> name;
    cout << "Enter your salary: $";
    cin >> salary;
}
```

```
//Global Function
void display(Person obj){
   cout << "\nYour name is: " << obj.name << endl;
   cout << "Your salary is: $" << obj.salary << endl;
}

//The main() function
int main()
{
   Person obj;
   obj.getData();   //Using object name to call member
function
   display(obj);     //Don't need object name to call
friend function
   return 0;
}
```

Here, we have a class called *Person* along with *getData()* and *display()* as two of its functions. *getData()* is a regular member function included with an intention to input user information. We have also used a global function known as *display()* and declared it as friend to this class *Person*. As the *display()* function is a friend function, its declaration neither includes "class_name" and "::" operator nor the keyword "friend" in its definition. The reason behind not using the "class_name" and "::" is that a friend function is an outside function and it does not belong to the scope of that class. This is the same reason why we also do not use an object variable to call a friend function. E.g., call to *display()* do not feature object name, dot operator combination whereas *getData()* function does.

Another important thing about a friend function is that we have to explicitly pass the object as an argument to the function. This is required because a friend is not stored alongside any other regular member functions in the memory. In simple terms, a friend function is stored in a place that is not common to objects and functions of a class to which it is declared as friend. Now, if this is the case then how could the friend function get access to variables of any object? The solution is to pass the object itself as an argument just like we passed object *obj* of *Person* class as an argument to the *display()* function in the above program. Otherwise, the display() wouldn't

be able to access name and salary information despite being a friend. The output of the program is:

```
Enter your name: abc
Enter your salary: $1

Your name is: abc
Your salary is: $1
```

To summarize a friend function, let us see some of the features of friend function:

- A function can be declared as friend to any number of classes.
- It does not belong to the class to which it is declared as friend. Hence, object name and dot operator pair are not used to call a friend function.
- A friend function can be declared in the private or public section of a class without changing its meaning.
- In most cases, it will have object as argument. This is because unlike any regular member function it cannot access member variables directly.

Similar to a friend function, we can declare a friend class as well. We can declare a class as friend in another class so that its members can access the private and protected members of that class. This is useful when we want to allow a particular class to access the restricted members of another class.

The following example program demonstrates a case where we have two classes A and B. Class A has some private member variables and a friend class B. As B becomes friend of class A, it can now access all the variables of A. Therefore, taking advantage of this fact we will try to access and print values of variables declared in the scope of class A by utilizing class B. The program is given below. In this example, we have observed that class A is not declared as friend in class B. In this case, one important point is to remember that class A cannot be able access the private and protected members of class B.

Program 8.8: C++ program to demonstrate friend class

```
#include <iostream.h>

class A {
private:
   int val;
public:
  void getData();
   friend class B;  // Friend class declaration
};

void A :: getData()
{
    cout << "Enter the value of val: ";
    cin >> val;
}

//friend class definition
class B {
public:
   void display(A obj){
      cout<<"You have entered: " << obj.val << endl;
//private data access
   }
};

int main() {
   A obj1;
   obj1.getData();       //regular function call

   B obj2;
    obj2.display(obj1);    //friend class's display()
function called
   return 0;
}
```

And, the output is:

```
Enter the value of val: 2
You have entered: 2
```



CHECK YOUR PROGRESS

Q 7: Which of the following is not a member of a class?

(a) Static function (b) Friend function (c) Const function (d) Virtual function

Q 8: A friend class can access _____ members of a class to which it is declared as friend.

Q 9: State whether true or false "Friend class can access all the private members of the class, of which it is a friend".

Q 10: Where does keyword 'friend' should be placed?

(a) function declaration (b) function definition

(c) main function (d) block function

Q 11: Which of the following statement is correct?

(a) Friend functions are in the scope of a class

(b) Friend functions can be called using class objects

(c) Friend functions can be invoked as a normal function

(d) Friend functions can access only protected members not the private members



8.11 LET US SUM UP

The key points to keep in mind in this unit are:

- A class is a template or a blueprint whereas an object is an instance of a class.
- A class can be viewed as an extension to structures. It contains data and functions wrapped together into one unit.
- Data represents property and functions represent the capacity of a class.
- The default visibility mode of a structure is public. However, the same is private for a class.

- In the common scenario, member variables are declared as private whereas member functions are declared as public in a class.
- Public data can be accessed using dot operator but not private data. Private data can only be accessed through member functions of that class and that function must be a public one.
- The class boundary ends with a semicolon after the closing "}".
- It is possible to have the same function name used in different classes. In such a situation, class name and scope resolution will help to resolve any ambiguity.
- A function defined inside the class is known as inline function. A function defined outside the class boundary can also be made inline by using the keyword *inline*.
- Memory for an object is allocated when it is declared and not when a class is defined. Separate memory is allocated for different objects. However, member functions are stored once and are commonly accessible to all the objects of the same class.
- A member variable of a class can be declared as static. Static variables are useful while dealing with values common to the entire class.
- A static function can access other static members only.
- A friend function is a special type of function which does not belong to the scope of the class. Moreover, by virtue of being friend, it can access the private member of that class.
- We can also declare a class as friend to another class. Private members of a class can be accessed by its friend class.



8.12 FURTHER READING

1. Balagurusamy, E. (2011). *Object Oriented Programming with C++, 6e*. McGraw-Hill Education (India) Pvt Limited. https://books.google.co.in/books?id=TN9wQjjDwp0C

2. ISO/IEC. (2003). International Standard ISO/IEC 14882 Second Edition Programming languages - C++. *ISO/IEC*, 786. http://www.iso.org/iso/catalogue_detail.htm?csnumber=38110
3. Pardoe, J., & King, M. (1997). Object Oriented Programming Using C++. In *Object Oriented Programming Using C++*. https://doi.org/10.1007/978-1-349-14449-5
4. Plauger, P. J. (2002). The C/C++ programming language. *C/C++ Users Journal*.
5. Stroustrup, B. (2014). A Tour of C++. In *Climate Change 2013 - The Physical Science Basis*.
6. Van Weert, P., & Gregoire, M. (2016). C++ Standard Library Quick Reference. In *C++ Standard Library Quick Reference*. https://doi.org/10.1007/978-1-4842-1876-1



8.13 ANSWERS TO THE CHECK YOUR PROGRESS

Ans to Q.No.1: (a) object

Ans to Q.No.2: (b) private members

Ans to Q.No.3: True

Ans to Q.No.4: False

Ans to Q.No.5: dot

Ans to Q.No.6: scope resolution

Ans to Q.No.7: (b) Friend function

Ans to Q.No.8: private

Ans to Q.No.9: True

Ans to Q.No.10: (a) function declaration

Ans to Q.No.11: (c)Friend functions can be invoked as a normal function



8.14 MODEL QUESTIONS

Q1. What is the difference between a class and a structure?

Q2. Can you call a member function of a class without creating an instance of it?

Q3. What is the difference between a class and an object?

Q4. Explain the use of friend function with the help of a suitable example.

Q5. Write a C++ program to calculate the sum of two values and display the result using friend function.

Q6. Write a C++ program to display a number of objects created using static members.

Q7. Define a class named 'Bank Account' to represent following members:
Member variables- account_number, depositor_name, account_type, balance.
Member functions- getData(), deposit(), withdraw(), putData().
Write a C++ program to input and display of Bank Account details for 3 customers.

Q8. A C++ program has two functions namely *main()* and *test()*. The main() function has a variable named *val1* declared in its scope. Similarly, the *test()* function has got two variables declared as *val2* and *val3* where val3 is of type static. In addition, there exists a fourth variable *val4,* which is a global variable. Considering this condition, state the scope and lifetime of each of these variables.

*** ***** ***

UNIT 9 : CONSTRUCTORS AND DESTRUCTORS

UNIT STRUCTURE

9.1 Learning Objectives

9.2 Introduction

9.3 Constructors

9.4 Constraints on Constructors

9.5 Types of Constructor

- 9.5.1 Default Constructor
- 9.5.2 Parameterized Constructor
- 9.5.3 Copy Constructor

9.6 Overloading of Constructors

9.7 Destructor

- 9.7.1 Constraints on Destructor

9.8 Dynamic Initialization of Objects

9.9 Let Us Sum Up

9.10 Further Reading

9.11 Answers to Check Your Progress

9.12 Model Questions

9.1 LEARNING OBJECTIVES

After going through this unit, you will be able to –

- learn about constructors and destructors in C++
- describe the way of declaring constructor and destructor
- learn different types of constructors and their use
- learn the concept of constructor overloading
- initialize object dynamically

9.2 INTRODUCTION

We have already learnt to create member functions in a class. We have seen so far a few examples of classes being implemented. In all cases, we defined a separate member function for reading input values for data members.

With the use of object, member functions are invoked and data members are initialized. These member functions cannot be used to initialize the data members at the time of the creation of an object.

An ideal situation would be for an object to initialize itself at the time of creation. This type of automatic initialization is achieved using a special member function called a constructor. Constructor enables an object to initialize itself when it is created and destruct or destroys the object when it is no longer required. This unit is about constructor and destructor. Different types of constructor and their implementation will be discussed in this unit.

9.3 CONSTRUCTORS

A constructor is a special member function in a class that is executed automatically whenever a new object is created. The main operation of constructor is to allocate the required resources, such as memory, and initialize objects of its class.

Constructors are distinct from other members of the class because they have the same name as their class. It is usually convenient if we can initialize object when it is first created, without the need of making a separate call to its functions and this is possible with the use of constructors. The syntax for declaring and defining a constructor with in a class is as follows:

```
class classname
{
        private:
                datamembers;
                datamembers;
        public:
                classname();    //Constructordeclaration
};
classname::classname()      //Constructordefinition
{
                //Bodyofconstructor
}
```

For example, let us declare a class with the class name ‘circle’. If we use constructor, then the construct or name must also be ‘circle’. The program can be written as follows:

/***Program9.1:**Program to demonstrate constructor while calculating the area of a circle*/

```
#include<iostream.h>
#define PI 3.1415
class circle
{
    private:
       float radius;
    public:
        circle();       //constructor declaration
        void area()     // member function
        {
          cout<<"\nArea of the circle is :" ;
           cout<<PI*radius*radius<<" sq.units\n";
        }
};
circle :: circle()    //constructor definition
{
    cout<<"\nEnter radius of the circle: ";
    cin>>radius;
}
int main()
{
        circle c;//constructor invoked automatically
        c.area();
        return 0;
}
```

The output of the program will be: Enter radius of the circle:5
Area of the circle is: 78.5375sq. units

In the above program, the constructor **circle( )** takes the value of radius from the keyboard. Although the data member is private, the constructor

could initialize them as constructor is also a member function of its class. The statement, '**circle c;'** declares avariable **c** as object of (type) class **circle**. It also calls the construct or implicitly.

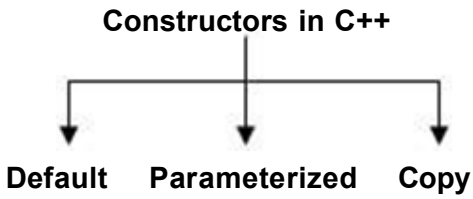
9.4 CONSTRAINTS ON CONSTRUCTORS

Constructors have some special constraints or rules that must be followed while defining them. These are as follows:

- Constructors should have the same name as the class in which it is defined.
- Constructors donot have any return type, not even *void*.
- Constructors are normally used to initialize the data members of a class.
- Constructors are also used to allocate resources like memory to the dynamic data members of a class.
- Constructors are normally declared in the public access within a class.

9.5 TYPES OF CONSTRUCTOR

Let us now discuss the different types of constructor in C++. In C++, constructors are classified into three categories. They are *default*, *parameterized,* and *copy* constructors.



9.5.1 The Default Constructor

A **default constructor** is a constructor that either takes no parameters or if it takes parameters, all the parameters should have some default values. The following code fragment shows the syntax of a default constructor.

```
class class_name
{
      private:
            datamembers;
      public:
            class_name();    //default constructor
};
```

If no constructor is defined for a class, then the compiler supplies the default constructor. The following program shows the use of default constructor.

//**Program 9.2:** Program that contains a default constructor with some default values.

```
#include <iostream.h>
class box
{
        int height, width, depth;
   public:
        box(int h=5,int w=8,int d=3) // default constructor
        {
         height=h;
         width=w;
         depth=d;
       }
       int volume()
       {
           return height*width*depth;
       }
};
int main()
{
    box b;
    cout<<b.volume();
    return 0;
}
```

The output will be 120.

//**Program 9.3:** Program that contains a default constructor with no parameter.

```
#include<iostream.h>
class value
{
   public:
          int a, b;
          value()    // Default Constructor
          {
             a = 20;
             b = 40;
          }
 };
int main()
{
     // Default constructor called automatically
     // when the object is created
     value obj;
     cout << "a: " << obj.a << endl
          << "b: " << obj.b;
     return 0;
}
```

The following code segment defines a class “myclass” with two default constructors.

```
class myclass
{
public:
myclass();//Defaultconstructorwithnoargumentsmy
class(float=10.5);//Defaultconstructorwith
//one default argument
};
```



CHECK YOUR PROGRESS

Q 1: State whether the following statements are True (T) or False (F):

(i) The constructor is not a member function.

(ii) It is wrong to specify are turn type for a constructor.

(iii) It is possible to define a class that has no constructor at all.

(iv) The name of a constructor need not be the same as that of the class to which it belongs.

(v) A class may have two default constructors.

Q 2: Choose the appropriate option:

(i) A function that is automatically called when an object is created is known as:

(a) constructor (c) delete

(b) destructor (d) free()function

(ii) Constructor with no parameter is known as

(a) stand-alone constructor (c) copy constructor

(b) default constructor (d) none of these

(iii) For a class namely *Shape*, the constructor will be like

(a) void Shape() {.....} (b) shape(){.....}

(c) ConstructorShape(){.....} (d) Shape(){.....}

Q 3: Write a programin C++ that displays the factorial of a given number using a constructor.

Q 4: Write a program in C++ to find the area of a rectangle of length 10.0 and breadth 6.0 by using a class “rectangle”. The program should also contain a constructor along with other member functions.

9.5.2 Parameterized Constructor

The constructor that can take arguments is called a ***parameterized constructor.*** The arguments can be separated by commas and they can be specified within braces similar to the argument list in function.

When a constructor has been parameterized, the object declaration with out a parameter may not work. In the case of a parameterized constructor, we must provide the appropriate arguments to the constructor when an object is declared.This can be done in two ways:

- By calling the constructor ***explicitly***.
- By calling the constructor ***implicitly***.

The implicit call method is sometimes known as the short hand method as it is shorter and is easy to implement.

Let us consider the following example to demonstrate how parameterized constructor works.

//**Program 9.4:** Program creating parameterized constructor

```
#include<iostream.h>

class student
{
     private:
        int roll, age, marks;
     public:
     student(int r,int m,int a);//parameterized constructor
     void display( )
     {
              cout<<"\nRoll number :" <<roll <<endl;
              cout<<"Total marks :"<<marks<<endl;
              cout<<"Age:"<<age<<endl;
         }
};   //end of class declaration
student::student(int r, int m, int a) //constructor definition
{
     roll = r;
     marks =m;
     age=a;
}
int main( )
```

```
{
    student manoj(5,430,16);  //object creation
    cout<<"\nData of student 1 :"<<endl;
    manoj.display();
    student rahul = student(6,380,15);//object creation
    cout<<"\n\nData of student 2 :"<<endl;
    rahul.display();
    return 0;
}
```

The output of the above program will be like this:

Data of student 1:
Roll number: 5
Total Marks: 430
Age:16

Data of student 2:
Roll number: 6
Total Marks: 380
Age: 15

In the above program, **student (int r, int m, int a,)** is a parameterized constructor with two arguments. In the main()function, we see that there are two objects, **manoj,** and **rahul** of class type student. The object manoj is initialized with the values 5,430 and 16 with the *implicit call* statement, '**student manoj(5, 430,16);'**.

The object rahul is initialized with the values 6,380 and 15 with the *explicit call* statement, '**student rahul =student (6, 380,15);'**. This statement creates a student object rahul and passes the values 6,380 and 15 to it.

Constructors with Default Arguments

It is possible to have a constructor with arguments having default values. It means that if we have a parameterized constructor with n parameters, then we can invoke it with less than n parameters

specified in the call. It is useful when most of the objects to be created are likely to have the same value for some data members. We need not specify that in every invocation. For example, suppose we want to record the data of class x (ten) students. For this, we can consider the same class **student** as shown in ***Program9.4***. Most of the students are of age16. Hence, their birth year is likely to be the same. Only a few of them may have a different year of birth. For this we can use the following statement.

student (int r,int m,int a=16);

Here the third parameter is given the default value. Now let us consider the following statement.

students (1,360);

The above statement as signs the values **1** and **360** to the argument **r** and **m** respectively and **16** to the argument **a** by default. Again, let us consider the following statement.

student t(2,400,15);

The above statement as signs **2**,**400** and **15** to **r**, **m** and **a** respectively. Thus, the actual arguments, when specified, over ride the default values.



EXERCISE

1.Q. Write a C++ program to create a class "EMPLOYEE" to initialize EMP_ID, DESIGNATION, and BASIC_PAY using a constructor and display data for three employees.

9.5.3 Copy Constructor

Copy constructor is one of the most important forms of an overloaded constructor. Object of the same class cannot be passed as an argument to a constructor of that class using pass by value method.

```
class  student
{
      private:
           .......
      public:
         student(students);  //This is not a valid declaration
};
```

But it is possible to pass an object of the same class as an argument to a constructor using pass by reference method. Such type of constructor, i.e., a constructor having parameter that is are ference to object of the same class is called a ***copy constructor***. Thus, the following declaration is a valid declaration.

```
class  student
{
      private:
      int roll,marks,age;    //data members
public:
      student(student&s);   //copy constructor
};
```

In the above statements, reference to an object of class student is passed to the copy constructor. Copy constructor is also called one argument constructor as it takes only one argument. The main use of copy constructor is to initialize the objects while increation and also used to copy an object. This constructor allows the programmer to create a new object from an existing one by initialization. Let us demonstrate copy constructor with the following program.

//***Program9.5:*** Program to demonstrate copy constructor

```
#include<iostream.h>
class student
{
    private:
        int roll, marks, age;
    public:
        student(int r,int m,int a)//parameterized
                                  //constructor
        {
            rol l = r;
            marks = m;
            age = a;
        }
        student(student &s)     //copy constructor
        {
            roll = s.roll;
            marks = s.marks;
            age = s.age;
        }
        void display( )
        {
            cout<<"Roll number :" <<roll <<endl;
            cout<<"Total marks :"<<marks<<endl;
            cout<<"Age:"<<age<<endl;
          }
};
int main( )
{
    student t(3,350,17);
    student k = t; // invokes copy constructor
    cout<<"\nData of student t:"<<endl;
    t.display();
    cout<<"\n\nData of student k:"<<endl;
    k.display();
    return 0;
}
```

The output of the above program will be like this: Data of student t:

Roll number: 3

Total marks: 350

Age :17

Data of student k:

Roll number : 3

Total marks: 350

Age : 17

The statement, 'student t (3,350,17);' invokes the parameterized constructor and 'student k=t;' invokes the copy constructor. The statement, 'student k=t;' can also be written as 'student k(t);'. This statement initializes one object **k** with another object **t**. The data members of **t** are copied member by member into **k**. When we see the output of the program, we observe that the data of both the object stand k are the same.

9.6 OVERLOADING OF CONTRUCTORS

In our previous unit, we have already learnt to overload functions by specifying different parameter lists for the same function name. In the same way, a class may contain multiple constructors i.e., a class can have more than one constructor with different parameter lists. The constructors are then recognized depending on the arguments passed. It may differ interms of the number of arguments, or data types of their arguments, or both. This is called ***overloading of constructors*** or ***constructor overloading***. In the earlier sub-section, ***Program 9.5*** *can also be considered as* an example of constructor overloading where two constructors are defined (One is parameterized and the other is a copy constructor). In ***Program9.6***, a simple example of constructor overloading is shown.

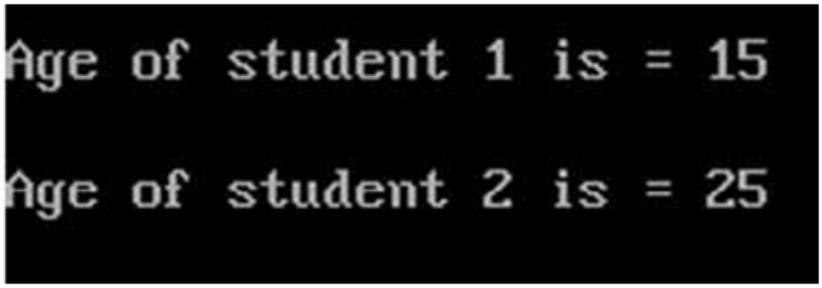
//Program 9.6: C++ program to demonstrate constructor overloading

```
#include<iostream.h>
 class student
{
 private:
    int age;

 public:
      student()       //no argument constructor
      {
       age = 15;
      }

      student(int a)      //constructor with one parameter
      {
       age = a;
      }
      int getage()      //member function
      {
       return age;
      }
};
int main()
{
  student s1, s2(25);
  cout << "\n\nAge of student 1 is = " << s1.getage() << endl;
  cout << "\nAge of student 2 is = " << s2.getage() << endl;
  return 0;
}
```

The output of **Program 9.6** is as follows:



Let us take another example to demonstrate overloading of constructors.

//**Program 9.7:** Demonstration of constructor overloading

```
#include<iostream.h>
#include<math.h>
class complex
{
    private:
       float real,imag;
    public:
       complex( )//constructor with no argument
       {
          real = imag = 0.0;
       }
       complex(float r, float i) //constructor with two arguments
       {
         real = r;
        imag = i;
       }
       complex(complex &c)   //copy constructor
       {
         real = c.real;
        imag = c.imag;
       }
       complex addition(complex d);
       void display( )   //member function
       {
          cout<<real;
          if(imag < 0)
               cout<<"-i";
          else
               cout<<"+i";
          cout<<fabs(imag);
              /*fabs calcute the absolute value
               of a floating point number */
        }
    };
    complex complex::addition(complex d)
    {
```

```
        complex temp;
               /*temporary object of
                  type complex class*/
        temp.real=real+d.real; //real parts added
        temp.imag=imag+d.imag; //imaginary parts added
        return(temp);
    }
: main( )


    complex x1,x4; //invokes default constructor
    cout<<"\nThe complex numbers in a+ib form :\n\n";
    cout<<"First complex number: ";
    x1.display();
    complex x2(1.5,5.3);//invokes parameterized constructor
    cout<<"\nSecond complex number: ";
    x2.display();
    complex x3(2.4,1.9);
    cout<<"\nThird complex number: ";
    x3.display();
    cout<<"\nAddition of 2nd and 3rd complex number: ";
    x4=x2.addition(x3);   //function call
    x4.display( );
    cout<<"\nThe result is copied to another object:";
    complex x5(x4); //invokes copy constructor
    x5.display();
     return 0;
}
```

The output of the above **Program 9.7** will be like this

```
The complex numbers in a+ib form :

First complex number: 0+i0
Second complex number: 1.5+i5.3
Third complex number: 2.4+i1.9
Addition of 2nd and 3rd complex number: 3.9+i7.2
The result is copied to another object:3.9+i7.2_
```

We have the following three constructors in the program

```
complex( ); //default constructor
complex(float r, float i) ; //parameterized
complex(complex &c); //copy constructor
```

These constructors are invoked during the creation of an object depending on the number and types of arguments passed. The default constructor, **complex( )** initializes the data members *real* and *imag* to 0.0. In main( ),the statement, '**complex x2 (1.5,5.3);'** passes two parameters to the constructor explicitly with the help of the parameterized constructor, **complex (float r, float i).** With the help of the copy constructor, **complex (complex &c),** data members of one object is copied member by member into another object.

9.7 DESTRUCTOR

A destructor is also a special member function of a class provided in the C++ programming language. As the name implies, it is usedto destroy the objects that have been created by a constructor. It is automatically called when an object is destroyed. A destructor must have the same name as that of the class name, but preceded with a ***tilde ~*** sign. Destructor takes no arguments and has no return type. The general syntax of a destructor is as follows:

```
class classname
{
    private :           //Private members
        ........ ;
    public :
        ~classname( ); //Destructor declaration
};
classname :: ~classname( )//Destructor definition
{
        //Body of destructor
}
```

In the above, the symbol tilde ~ represents a destructor that precedes the name of the class. Like the default constructor, the compiler always creates a default destructor if we don't create one. Similar to constructors, a destructor must be declared in the public section of a class. Destructor cannot be *overloaded* i.e., a class cannot have more than one destructor.

//**Program 9.8:** Program that contains member function, constructor, and destructor.

```
#include<iostream.h>
class  rectangle
{
        int *width, *height;
  public:
        rectangle(int,int);  //constructor declaration
        ~rectangle();    //destructor declaration
        int findarea() //member function
        {
            return(*width**height);
        }
   };

rectangle::rectangle(int  a, int  b)
{
     width = new int;
     height = new int;
     *width = a;
     *height = b;
}
rectangle::~rectangle()   //destructor defined
{
    delete width;
    delete height;
}
int main()
{
     rectangle r1(5,8),r2(3,6);
     cout<<"Area is :"<<r1.findarea()<<endl;
```

```
    cout<<"Area is:"<<r2.findarea();
    return 0;
}
```

The output of the program will be

Area is 40

Area is18

/***Program9.9:*** Program demonstrating destructor*/

```
#include<iostream.h>
#include<string.h>
class student
{
    private:
        int age;
        char name[25];

    public:
        student(int a, char n[25])
        {
            age=a;
            strcpy(name,n);
        }
        void show()
        {
           cout<<"\nThe name of the student is: "<<name;
           cout<<"\nHis age is: "<<age;
        }
        ~student() //destructor defined
        {
            cout<<"\nObject Destroyed";
            }
    };
    int main()
    {
        student s1(21,"Rahul");
        student s2(23,"Dipankar");
        s1.show();
        s2.show();
    }
```

To see the execution of destructor, we have to press Alt+F5 after compiling and running the program. The output willbe like this:

The name of the student is: Rahul

His age is: 21

The name of the student is: Dipankar

His age is: 23

Object Destroyed

Object Destroyed

9.7.1 Constraints on Destructor

The following points should be kept in mind while defining and writing the syntax for the destructor.

- A destructor must be declared with the same name as that of the class to which it belongs. But the destructor name should be preceded by at ilde(~).
- A destructor should be declared with no return type.
- A destructor must have public access in the class declaration.

9.8 DYNAMIC INITIALIZATION OF OBJECTS

We can dynamically initialize objects through constructors. Object's datamembers can be initialized dynamically at run time even after their creation.

//Program9.10 : Demonstration of dynamic initialization of object

```
#include<iostream.h>
#include<math.h>
class number
{
    public:
       int num;
       number(int n)
       {
          num = n;
```

```
        }
        int sum( )
        {
           num = num+5;
           return(num);
        }
};
int main( )
{
    number obj1(1);  // parameterized constructor invoked
    number obj2(2);
    cout<<"\nValue of object 1 and object 2 are : ";
    cout<<obj1.num<<"\t"<<obj2.num;
    number obj3(obj1.sum( ));//dynamic initialization of object
    cout<<"\nValue of object 1 after calling sum() :"<<obj1.num;
    cout<<"\nValue of object 3 is :"<<obj3.num;
    return 0;
}
```

The output of the above program will be:

Value of object 1 and object 2 are	:	1 2
Value of object 1 after calling sum()	:	6
Value of object 3 is	:	6

The statements,'**number obj1(1);'** and '**number obj2(2);'** will initialize obj1 and obj2 by 1 and 2 respectively by invoking the constructor number(intn).**obj1.sum( )** will return the value 6. This is passed as an argument in the statement

```
number  obj3(obj1.sum());
```

where obj3 is initialized dynamically with the value returned by **obj1.sum()**.



CHECK YOUR PROGRESS

Q 5: State whether the following statements are true (T) or false (F):

(i) A class may have more than one destructor.

(ii) When an object is destroyed, destructor is automatically called.

(iii) The presence of many destructors in a class is called destructor overloading.

(iv) A destructor can have are turn type.

(v) A destructor can have arguments like constructor.

(vi) We can pass arguments to a constructor.

(vii) Destructors cannot take arguments.

(viii) Destructors can be overloaded.

Q 6: Choose the appropriate option:

(i) Returntypeofadestructoris

(a) int (b) tilde

(c) void (d) nothing, destructor has no return value

(ii) A class may have _____ destructor

(a) two (b) only one

(c) many (d) none of these

(iii) Which sign precedes the name of the destructor?

(a) * (b) #

(c) ~ (d) -

Q 7: Distinguish between the following two statements:

```
timet1(14,10,30);          //statement1
timet1=time(14,10,30);     //statement2
```



9.9 LET US SUM UP

Based on the discussion so far, the key points to be kept in mind from this unit are:

- A constructor is a special member function for the automatic initialization of an object. Whenever an object is created, the constructor will be called.
- A constructor should have the same name as that of the class name to which it belongs.
- Constructor has no return type.

- The constructor without argument is called a default constructor.
- Constructors maybe overloaded to provide different ways of initializing an object.
- We may have more than one constructor where each has a different signature or arguments list.
- A constructor can accept are ference to its own class as a parameter. Such a constructor having are ference to an instance of its own class as an argument is known as copy constructor.
- C++ also provides another member function called destructor that releases memory by destroying objects when they are no longer required.
- A destructor never takes any argument nor does it return any value.



9.10 FURTHER READING

1. "*Mastering C++*", by Venugopal, Rajkumar, Ravisankar, Tata McGraw Hill publication.
2. "*Object-Oriented Programming with ANS Iand Turbo C++",* Ashok N.Kamthane, Pearson Education.



9.11 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q No 1:

(i) False (ii) True (iii) True (iv) False

(v) False

Ans to Q No 2:

(i) (a)constructor (ii) (b)defaultconstructor

(iii) (d)Shape(){.....}

Ans to Q No 3:

//Programforfindingfactorialofanumber

```
#include<iostream.h>
class factorial
{
    private:
        long n;
    public:
    factorial();
};
factorial::factorial()
{
    cout<<"\nEnter the number to find factorial: ";
    cin>>n;
    long fact =1;
    while(n>1)
    {
        fact =fact *n;
        n=n-1;
    }
    cout<<"\n\nThe factorial is : "<<fact;
}
int main()
{
    factorial f;
    return 0;
}
```

Ans to Q No 4:

/*Program for finding the area of a rectangle of length 10.0 and breadth 6.0*/

```
#include<iostream.h>class rectangle
{
     private:
          float length, breadth;
    public:
rectangle( )
{
     length = 10.0;
     breadth=6.0;
```

```
}
float area( )
{
     return(length*breadth);
}
};
int main( )
{
     rectangle r;
cout<<"\n The area of the rectangle is: "<<r.area()<<" square units";
return 0;
}
```

Ans to Q No 5:

(i) False (ii)True (iii)False (iv)False

(v) False (vi)True (vii)True (viii)False

Ans to Q No 6:

(i) (d) nothing, destructor has no return value,

(ii) (b)only one

(iii) (c)~

Ans to Q No 7: The first statement creates the object *t1* by calling the *time* constructor implicitly. On the other hand, the second statement creates the object *t1* by calling the *time* constructor explicitly.



9.12 MODEL QUESTIONS

Q1. What are constructor and destructor? Explain how they differ from normal member functions. Is it mandatory to use a constructor in a class?

Q2. What are the characteristics of a constructor?

Q3. Write short notes on

(i) Default Constructor

(ii) Copy Constructor

Q4. Differentiate between constructors and destructors.

Q5. Give the rules governing the declaration of a constructor and destructor.

Q6. What is a parameterized constructor?

Q7. Write a C++ program to show overloading of constructors?

Q8. Define a class 'complex_no' which has two data members, one for representing the real part and the other for complex part. Define a constructor to initialize the object.

Q9. Define suitable constructor(s) for the STRING class defined below;

```
STRING
{
        int length; chars[50];
        public:
        //definesuitableconstructors
};
```

Q10. How is dynamic initialization of object achieved?

Q11. Distinguish between the following two statements: employee e2(e1); employee e2=e1; e1 and e2 are objects of **employee** class.

Q12. Mention the importance of destructors.

*** ***** ***

UNIT 10: OPERATOR OVERLOADING

UNIT STRUCTURE

10.1 Learning Objectives

10.2 Introduction

10.3 Definition of Operator Overloading

10.4 Syntax for Operator Overloading

10.5 Implementing Operator Overloading

10.6 Overloading Unary Operators

- 10.6.1 Overloading Unary Operators using Member Function of a Class
- 10.6.2 Overloading Unary Operators using Friend Function

10.7 Overloading Binary Operators

- 10.7.1 Overloading Binary Operators using Member Function of a Class
- 10.7.2 Overloading Binary Operators using Friend Function

10.8 Manipulation of Strings using Operators

10.9 Rules for Overloading Operators

10.10 Type Conversions

- 10.10.1 Conversion from Basic to Class Type
- 10.10.2 Conversion from Class to Basic Type
- 10.10.3 Conversion from One Class to Another Class Type

10.11 Let Us Sum Up

10.12 Further Reading

10.13 Answer to Check Your Progress

10.14 Possible Questions

10.1 LEARNING OBJECTIVES

After going through this unit, you will be able to –

- learn about the definition and meaning of operator overloading
- describe the concept of overloading unary operators using member function

- learn about overloading unary operators using friend function
- get the concept of overloading binary operators using member function
- learn the concept of overloading binary operators using friend function
- learn how to manipulate strings using operator overloading
- define the various rules of operator overloading
- implement various data type conversions techniques such as basic type to class type, class type to basic type, and class type to another class type

10.2 INTRODUCTION

We are already familiar with various operators and their expressions used in C++. We are also acquainted with the ideas of classes, objects, constructors, and destructors in C++.

The term 'Operator Overloading' refers to assign multiple definitions or meanings of an operator. In this unit, we will learn about the concept of overloading of operators such as overloading of unary operators, binary operators, overloading binary operators using friend function, and manipulation of strings using operators.We will also learn about how type conversion is done in C++ such as the conversion of basic to class type, class to basic type, and one class to another class type.In the next unit, we will explore the concept of inheritance and its various types, implementations available in C++.

10.3 DEFINITION OF OPERATOR OVERLOADING

Operator overloading in C++ is an exciting feature in which an operator in one scope can be specified with more than one special user-defined meaning to use it on user-defined data types. For example, + operator can be overloaded to perform the following two tasks:

1. + operator is used to add two numbers.
2. + operator is also used to concatenate two strings.

The majority of C++ operators can be overloaded i.e. redefined. There are a few operators in C++ which cannot be overloaded such as scope resolution operator (::), sizeof operator, pointer-to-member Operator (.*), member access, or dot operator (.), ternary operator (?:), etc.

10.4 SYNTAX FOR OPERATOR OVERLOADING

To define an additional meaning to an operator in C++, we should define a special function inside a class called *operator function.* The syntax of the operator function is:

```
return_typeclassname::operator   symbol
(argument_list)
{
      ... .. ... // function body
}
```

Where *return_type*is the type of the value returned by the specified operation,*symbol* is the overloaded operator and *argument_list* is the list of the arguments passed to the operator function.The overloaded operator is preceded by the keyword **operator**. The operator function must be declared inside the class.

10.5 IMPLEMENTING OPERATOR OVERLOADING

To implement operator overloading, we can define the operator function either as a member function or a friend function.In this process, a friend function has one argument for unary operators and two arguments for binary operators. For example:

```
operator operator_symbol(x);     //unary operator
operator operator_symbol (x,y); // binary operator
```

But, in case of a member function, it has no argument for unary operators and one argument for binary operator.

To perform operator overloading we have to perform the following three steps:

Step-1: A class should be created that defines the data type that will be used in the overloading operation.

Step-2: Declare the overloading function **operator symbol ()** in the public section of the created class either as a member function or a **friend** function.

Step-3: Define the overloading function **operator symbol ()** to implement the required user-defined operations.

10.6 OVERLOADING UNARY OPERATORS

The Operators which work on a single operand are called unary operators. Following are some examples of unary operators:

- Increment operator (++)
- Decrement operator (- -),
- Unary minus operator (-)
- Logical not operator (!)

A unary operator can be overloaded using a friend function or a member function.

10.6.1 Overloding Unary Operators Using Member Function of a Class

The syntax for overloading unary operator using a member function is:

```
return_type  operator  symbol()
{
        // Operator Function's Body
}
```

Let us see, how to overload the unary minus (-) operator. We know that, if we apply the unary minus (-) operator to an integer number it changes the sign of that integer number, e.g. 7 is changed to -7. Now, we overload this operator so that it can be applied to an object in the same way as it is applied to the basic data item. The following example shows the overloading of *minus (-)* operator.

Program10.1 :C++ program to show the overloading of the unary minus (-) operator.

```
#include<iostream.h>
class  ChangeSign
{
      int num1, num2;
      public:
         void read_data()
         {
cout<<"\nEnter the first Number: ";
cin>>num1;
cout<<"\nEnter the second Number: ";
cin>>num2;
         }
      void operator-() //Overload Unary minus operator
         {
                  num1=-num1;
                  num2=-num2;
         }
         void result()
         {
cout<<"\n Number-1: "<<num1;
cout<<"\n Number-2: "<<num2;
         }
};
int main()
{
ChangeSignC;
C.read_data();
cout<<"\nBefore  Overloading:";
C.result();
         -C;      //activate operator -() function
cout<<"\n\n After Overloading:";
```

```
C.result();
        return 0;
}
```

Output:

Enter the first Number: 8

Enter the second Number: 9

Before Overloading:

Number-1: 8

Number-2: 9

After Overloading:

Number-1: -8

Number-2: -9

In the above example, the operator overloading function *operator-()* takes no argument and it does not return any value. Since the operator overloading function *operator-()* is a member function of the class *ChangeSign*, it can be accessed by applying the minus (-) operator on an object of the class *ChangeSign.* Since, the function *operator-()* does not return any value, a statement like `C2=-C1;` will not work.To use a statement like this, we have to assign return type other than void to the operator overloading function *operator-()*. The following program shows the example of operator overloading function *operator-()* modified to return a value.

Program 10.2:C++ program to show unary operator overloading where operator function returns a value.

```
#include<iostream.h>
class ChangeSign
{
     int num1, num2;
public:
void read_data()
{
```

```
            cout<<"\nEnter the first Number: ";
      cin>>num1;
      cout<<"\nEnter the second Number: ";
      cin>>num2;
       }
      ChangeSign operator-()//Overload Unary minus
      operator
       {
      ChangeSign temp;
            temp.num1=-num1;
            temp.num2=-num2;
             return temp;
        }
        void result()
        {
      cout<<"\n Number-1: "<<num1;
      cout<<"\n Number-2: "<<num2;
        }
};
int main()
{
      ChangeSign C1,C2;
        C1.read_data();
      cout<<"\nBefore Overloading:";
        C1.result();
        C2=-C1; //activate operator-() function
      cout<<"\n\n After Overloading:";
        C2.result();
        return 0;
}
```

Output:

Enter the first Number: 8

Enter the second Number: 9

Before Overloading:

Number-1: 8

Number-2: 9

After Overloading:

Number-1: -8

Number-2: -9

10.6.2 Overloding Unary Operators Using Friend Function

The syntax for declaring a friend function to overload unary operator is:

```
friend return_type operator symbol(class_nameobj);
```

Where `return_type` is the return type of the overloading function `operator symbol()`, `symbol` is the operator which is overloaded, `obj`is the object of the class `class_name`. Here, the overloading function is declared as *friend* to the class`class_name`. The following program performs overloading of the same unary *minus (-)* operator like in example 10.1 but in this case, *friend* function is used.

Program 10.3: C++ program to show unary operator overloading using friend function.

```
#include<iostream.h>
class  ChangeSign
{
int num1, num2;
public:
void  read_data()
{
      cout<<"\nEnter the first Number: ";
      cin>>num1;
      cout<<"\nEnter the second Number: ";
      cin>>num2;
 }
```

```
     friend void operator-(ChangeSign&x);
     //friend function declared
void result()
{
cout<<"\n Number-1: "<<num1;
cout<<"\n Number-2: "<<num2;
}
};
void operator-(ChangeSign&x)
//Overload Unary minus operator
{
x.num1=-x.num1;
x.num2=-x.num2;
}
int main()
{
ChangeSign C1;
C1.read_data();
cout<<"\nBefore Overloading:";
C1.result();
-C1; //invokig overloading function
cout<<"\n\n After Overloading:";
C1.result();
 return 0;
}
```

Output:

Enter the first Number: 4

Enter the second Number: 5

Before Overloading:

Number-1: 4

Number-2: 5

After Overloading:

Number-1: -4

Number-2: -5



EXERCISE

1. Modify exercise 10.2 to perform the overloading of unary minus (-) operator using friend function

10.7 OVERLOADING BINARY OPERATORS

The Operators which work on two operands are called binary operators. Following are some examples of binary operators:

- Addition operator (+)
- Subtraction operator (–),
- Division operator (/)
- Logical AND operator (&&)
- Logical OR operator (||)

Like unary operators, binary operators can also be overloaded using a friend function or a member function of a class.

10.7.1 Overloading Binary Operators Using Member Function of a Class

The syntax for overloading binary operator using a member function is:

```
return_type  operator  symbol(arg)
{
      // Operator function's body
}
```

Where, `return_type` is the return type of the overloading function `operator symbol()`, `symbol` is the operator which is overloaded, `arg` is the argument of any type that will be used in the function. In this overloading, the first operand i.e. the left-hand operand which invokes the overloading function is taken implicitly. The second operand i.e, right-hand operand is passed as an argument to the

overloading function. For example, if *S1* and *S2* are two objects of a class, then overloaded operator + is called as '`S3=S1 + S2;`' which is equivalent to '`S3=S1.operator + (S2);`'. The binary operators must explicitly return a value. The following example shows the overloading of - operator to subtract two complex numbers.

Program 10.4: C++ program to show the binary operator overloading using member function of a class.

```
#include<iostream.h>
class ComplexNumber
{
int r, i;
public:
void read_data()
{
     cout<<"\nreal part=";
     cin>>r;
     cout<<"\nimaginary part=";
     cin>>i;
 }
ComplexNumber operator-(ComplexNumber);

void result()
{
     cout<<r<<" + "<<i<<"j";
}
};
ComplexNumberComplexNumber:: operator-
(ComplexNumber X)
{
ComplexNumber temp;
temp.r=r-X.r;
temp.i=i-X.i;
```

```
return temp;
}
int main()
{
ComplexNumber a, b,c;
cout<<"\nEnter the 1st Complex number:\n";
a.read_data();
cout<<"\nEnter the 1st Complex number:\n";
b.read_data();
c=a-b;     //equivalent to c=a.operator-(b);
cout<<"\n1st Complex number is: ";
a.result();
cout<<"\n2nd Complex number is: ";
b.result();
cout<<"\nSubraction of two complex numbers is :";
c.result();
return 0;
}
```

Output:

Enter the 1st Complex number:
real part=7
imaginary part=3
Enter the 1st Complex number:

real part=2
imaginary part=1
1st Complex number is: 7 + 3j
2nd Complex number is: 2 + 1j
Subtraction of two complex numbers is :5 + 2j

In the above example, the data members of the object, are accessed directly in the operator function operator-(). The data members of the object, b are accessed using the dot (.) operator.



EXERCISE

2. Modify exercise 10.4 to perform the overloading of unary Plus (+) operator for the addition of two complex numbers using member function.

10.7.2 Overloading Binary Operators Using Friend Function

The overloading of binary operator using *friend* function is similar to the overloading by using *member* function. A *friend function* requires two arguments to be passed explicitly in contrast to one argument that is to be passed to member function while performing binary operator overloading. The syntax for declaring a friend function to overload binary operator is:

```
friend return_type operator symbol(arg_1, arg_2);
```

Where *friend* is the keyword to make the operator function as a friend to the operator class, `return_type` is the return type of overloading function. Here, `arg_1` and `arg_2` are arguments of any type. For example, if *S1* and *S2* are two objects of a class, then overloaded binary operator, + is called as '`S3=S1 + S2;`' which is equivalent to '`S3=operator+(S1,S2);`'.

The following example shows the overloading of the - operator to subtract two complex numbers using friend function.

Program 10.5:C++ program to show binary operator overloading using friend function.

```
#include<iostream.h>

class ComplexNumber
{
      int r, i;
      public:
      void read_data()
```

```
      {
            cout<<"\nreal part=";
            cin>>r;
            cout<<"\nimaginary part=";
            cin>>i;
       }
friend ComplexNumber operator-(ComplexNumber,ComplexNumber);
      //friend function declared
      void result()
      {
            cout<<r<<" + "<<i<<"j";
      }
};
ComplexNumber operator-(ComplexNumber X, ComplexNumber Y)
{
      ComplexNumber temp;
      temp.r=X.r-Y.r;
      temp.i=X.i-Y.i;
      return temp;
}
int main()
{
      ComplexNumber a, b,c;
      cout<<"\nEnter the 1st Complex number:\n";
      a.read_data();
      cout<<"\nEnter the 1st Complex number:\n";
      b.read_data();
      c=a-b;     //equivalent to c=operator(a,b);
      cout<<"\n1st Complex number is: ";
      a.result();
      cout<<"\n2nd Complex number is: ";
      b.result();
      cout<<"\nSubtraction of two complex numbers is :";
      c.result();
      return 0;
}
```

Output:

Enter the 1st Complex number:

real part=7

imaginary part=3

Enter the 1st Complex number:

real part=2

imaginary part=1

1st Complex number is: 7 + 3j

2nd Complex number is: 2 + 1j

Subtraction of two complex numbers is :5 + 2j

In the above example, the operator function, operator-() is declared as friend to the class `ComplexNumber`. So, the data members of aand b are accessed using the dot (.) operator.



CHECK YOUR PROGRESS

Q 1: Binary operator overloading using friend function takes _____ explicit arguments.

(a) 0 (b) 1 (c) 2 (d) 3

Q 2: Unary operator overloading using member function of a class takes _____ explicit arguments.

(a) 0 (b) 3 (c) 2 (d) 1

Q 3: Which of the following is the correct example of a binary operator?

(a) ++ (b) - - (c) : : (d) +

Q 4: Which of the following operator cannot be overloaded?

(a) . (b) : : (c) + (d) *

Q 5: An operator function is created using _____ keyword.

(a) destructor (b) constructor

(c) operator (d) manipulator

Q 6: In operator overloading, the *operator function* can be a

(a) Member function (b) Friend function

(c) Both (a) & (b) (d) None of the above

Q 7: Fill in the blanks:

(a) Operators can be redefined using _____ techniques.

(b) _____ is equivalent to writing `C1+ C2` while overloading the binary + operator using member function.

(c) _____ operators work on one operand.

(d) Binary operators work on _____ operand.

10.8 MANIPULATION OF STRINGS USING OPERATORS

C++ provides convenient and powerful tools to manipulate strings.In C++, users are permitted to create their own definitions of operators which can be used to manipulate the strings similar to the basic built-in data types in C++. There is a class called *string* is available in the C++ class library that supports all string manipulations using operator overloading. For example, the statement '`str3=str1+str2;`' joins two strings,`str1` and `str2`, and produces the new string,`str3`. Here, `str1`, `str2`, and `str3`are declared as objects of the string class.

In C++, we can also overload operators like +, <= and == for string manipulations. The following example shows the overloading of the + operator for concatenation of two strings.

Program 10.6:C++ program to overload the binary operator + for concatenation of two strings.

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

class String
{
      char str[20]; //member variable for string input
      int len;
      public:
      void read_string(); //member function
```

```
        void display_string(); //member function
        String operator+(String s); //overloading
};
void String :: read_string()
{
        cout<< "\nEnter the string : " ;
        cin>> str;
        len=strlen(str) ;
}
void String::display_string() //member function for
output
{
        cout<<str;
}
String String :: operator+(String s) //overloading +
{
        String temp;
        strcpy(temp.str,str);
        strcat(temp.str,s.str);
        temp.len = len + s.len;
        return(temp);
}
int main()
{
        String str1,str2,str3; //creating three object
        str1.read_string();
        str2.read_string();
        str3=str1+str2; //overloading of + operator
        cout<<"\nConcatenated String: ";
        str3.display_string();
        return 0;
}
```

Output:

Enter the string :Newyork

Enter the string : America

Concatenated String: NewyorkAmerica

10.9 RULES FOR OVERLOADING OPERATORS

In C++, Operator overloading is not simple as it looks. There are some limitations and restrictions for performing operator overloading. The rules for operator overloading in C++ are:

- Only built-in operators can be overloaded.
- At least one operand of the overloaded operator should be of a user-defined type.
- The basic meaning of an operator cannot be changed.
- The binary arithmetic operators (`+, -, *, /`) must explicitly return a value.
- There are some operators that cannot be overloaded. **(S*ection 10.3*)**
- The operators,= (assignment), `[ ]`(subscription), `()` (function call) and `->` (class member access) operators cannot be overloaded using friend function. However they can be overloaded using as member functions of a class.
- Some operators like = (assignment) and`&`(address) are by default overloaded.
- Unary operators take no explicit arguments and return no explicit values when overloaded as a member function. But, it takes one reference argument (the object of the relevant class) when it is overloaded using friend function.
- Binary operators take one explicit argument when overloaded as a member function. Here, the left-hand operator must be an object of the relevant class.
- Binary operators take two explicit arguments when it is overloaded using friend function.

10.10 TYPE CONVERSIONS

We know that constants and variables of different basic data types are used in C++ programs. When these constants and variables are mixed together to represent a single expression, automatic type conversion takes place. The assignment operation also causes automatic type conversions between the operands as per the rules. The data type to the right of an assignment operator is automatically converted to the data type of the variable on the left. This process of data conversion is known as *Type Conversion.* For example, consider the following statements:

```
int m;
float n = 2.741;
m = n;
```

Here, float variable n is converted to an integer before its value assigned to m.

So, when data types are built-in types then the type conversion has automatically happened. But, when the data types involved are user-defined types then the automatic type conversion is not always carried out smoothly by the complier. For example, let us consider the statement,'S3 = S1 + S2;'.Here, if all three objects, (S1, S2 and S3)are from same class then the type conversion is executed smoothly. But, if any one of the above operands is an object from a different class or a built-in type then the programmer must design rules for this data conversion. There are three different possibilities that arise when the conversion of data type occurred between incompatible data types. These are:

- *Conversion from basic type to class type*
- *Conversion from class type to basic type*
- *Conversion from one class type to another class type*

10.10.1 Conversion from Basic to Class Type

In this conversion, type conversion is performed from a variable of basic type to an object of a user-defined class. This conversion is easy to carry out and performed by using the constructor of a

class.The following example shows the conversion of basic data type in to a user-defined class.

Program 10.7:C++ program to show type conversion from the basic data type int to a class type.

```
#include <iostream.h>
// Length Class
class Length
{
      int meter;
      int centi_meter;
      public:
      // Default Constructor
      Length()
      {
      meter = 0;
      centi_meter = 0;
      }
      // Parameterized Constructor
      Length(int L)
      {
      meter = L / 100;
      centi_meter = L % 100;
      }
      void Display()
      {
      cout<< "Length = " << meter;
cout<<"meters and "<<centi_meter<<"centimetres\n";
      }
};
int main()
{
// Object of Time class
```

```
Length L1;
int distance = 395;
// Conversion of int type to class type
L1 = distance;
L1.Display();
return 0;
}
```

Output:

Length = 3 meters and 95 centimetres

In the above example when object `L1` is created and the default constructor is called. The variable `distance` of integer data type is converted into class type `Length` by invoking the parameterized constructor at statement `L1 = distance`.

10.10.2 Conversion from Class to Basic Type

In this conversion, type conversion is performed from an object of a class to a variable of basic data type. This conversion is not supported by constructor function. This type of conversion is performed by defining an *overloaded casting operator* provided in C++. The syntax of an *overload casting operator* is given below:

```
operator type_name()
     {
        // Program code
        //.....
     }
```

While defining the overloaded casting operator function following conditions should be satisfied:

- The overloaded casting operator function must be a class member.
- The operator function converts the class type into the type_name wheretype_name is a basic data type.
- The operator function must not return any value.

- The operator function must not have any argument.

  The following example shows the conversion of a `class type` into a basic data type `int`.

  Program 10.8: C++ program to show type conversion from a class type to a basic type.

```
#include <iostream.h>
// Length Class
class Length
{
      int meter;
      int centi_meter;
     public:
      // Parameterized Constructor
      Length(int,int);
      operator int();  // casting operator function
};
Length::Length (int m, int c)
{
    meter = m;
centi_meter = c;
}
Length :: operator int()
{
int d;
cout<<"Class Type to Basic Type Conversion"<<endl;
d=meter*100+centi_meter;
return(d);
}
int main()
{
int m,cm,distance;
m=3;
cm=95;
```

```
        Length L1(m,cm); // construct object
distance = L1.operator int();
cout<<"\nTotal centimetres are "<<distance;
        return 0;
}
```

Output:

Class Type to Basic Type Conversion

Total centimetres are 395

In the above example,type conversion takes place at statement, ‘`distance = L1.operator int();`’.

10.10.3 Conversion from One Class to Another Class Type

In this type conversion, the object of a source class is converted into the object of a destination class.This type conversion is done in two ways namely *conversion in the source class* or *conversion in the destination class* as shown in the Fig. 10.1.

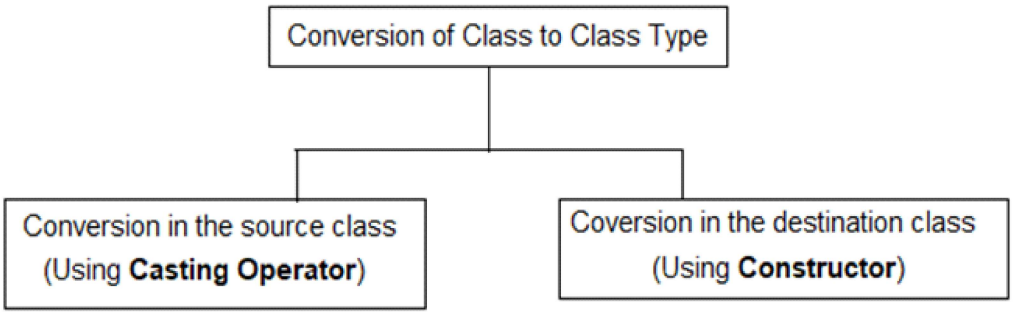


Fig.10.1. Type Conversion of Class to Class Type

Conversion in the source class is performed by using *casting operator* and conversion in the destination class is performed by using a single parameter *constructor*.

- Declaration syntax of *casting operator function:*

```
operator dst_Class();
```

Where `operator` is the keyword and `dst_class` is the name of the destination class.

- Declaration syntax of *constructor function for the type conversion:*

```
Dst_Class(Src_ClassObj);
```

Where Dst_Class is the destination class, Src_Class is the source class and Obj is the object of the class Src_Class.

The following examples show the conversion of the object of class `Length` to the object of class `Centimetre` using *overloaded casting operator* function `operator Centimetre()` in the source class.

Program 10.9: C++ program to show type conversion from one class to another class type using casting operator function.

```
#include <iostream.h>
//Centimetre class
class Centimetre
{
public:
int cm;
// Constructors
Centimetre()
{
     cm = 0;
}
     // Function to print the value of total
Centimetres
void show_centimetres()
{
     cout<< "\nTotalCentimetres : " << cm<<endl;
}
};
// Length Class
class Length
{
```

```
int meter;
int centi_meter;
public:
// Parameterized Constructor
Length(int,int);
operator Centimetre()// casting operator function
{
Centimetre c;
c.cm = (meter * 100) + centi_meter;
return c;
}
     // Function to print the value of meter and
centimetres
     void Display_distance();
};
void Length::Display_distance()
{
     cout<< "Meter : " << meter <<endl;
     cout<< "Centimetre : " <<centi_meter<<endl;
}
Length::Length (int m, int c)
{
     meter = m;
     centi_meter = c;
}
int main()
{
Length L1(3,95);
Centimetre cm;
cm=L1; //Centimetre class is destination and Length
class is source class
L1.Display_distance();
```

```
cm.show_centimetres();
return 0;
}
```

Output:

Meter : 3

Centimetre : 95

Total Centimetres : 395

The following examples show the conversion of the object of class`Length` to the object of class `Centimetre` using a single parameter *constructor* in the destination class `Centimetre`.

Program 10.10: C++ program to show type conversion from one class to another class type using a single parameter *constructor* in the destination class.

```
#include <iostream.h>
// Length Class
class Length
{
public:
int meter;
int centi_meter;
void get_distance();
void Display_distance();
};
//Centimetre class
class Centimetre
{
public:
int cm;
// Constructors
Centimetre()
{
```

```
        cm = 0;
}
Centimetre(Length  L1)
{     // Constructor at the desination class
      cm=L1.meter*100 + L1.centi_meter;
}
      // Function to print the value of total
Centimetres
void  show_centimetres()
{
      cout<< "\nTotalCentimetres : " << cm<<endl;
}
};
void  Length::Display_distance()
{
cout<< "Meter :" << meter <<endl;
cout<< "Centimetre : " <<centi_meter<<endl;
}
//int  Length::get_metre()
void  Length::get_distance()
{
meter = 3;
centi_meter=95;
}
int main()
{
Length L1;
Centimetre  cm;
L1.get_distance();
L1.Display_distance();
cm =L1;
cm.show_centimetres();
return 0;
}
```

Output:

Meter : 3

Centimetre : 95

Total Centimetres : 395



CHECK YOUR PROGRESS

Q 8: Which of the following operator is by default overloaded?

(a) +(b) –(c) =(d) None of the above.

Q 9: The number of arguments in the typecasting operator is—

(a) 0(b) 1(c) 2(d) 3

Q 10: What is the return type of the typecasting operator function?

(a) int(b) void(c) char(d)No return type

Q 11: The conversion from a class type to a basic is done by using _____.

(a) Built-in conversion function

(b) An overloaded = operator

(c) A conversion function that's a member of the class.

(d) Single argument constructor

Q 12: Which one of the following about casting operator is incorrect?

(a) It should be a class member

(b) It has no return type

(c) It has no argument

(d) It is a binary operator

Q 13: State True/False:

(a) The typecasting operator has only one argument.

(b) A friend function is used to convert basic data type to class type.

(c) Type casting is done automatically between built-in data types.

(d) The casting operator should be declared in the source class.



10.11 LET US SUM UP

- Operator overloading refers to the assignment of multiple definitions or meanings for an operator.
- A special function is used to perform operator overloading which is called as *operator function.*
- The *operator function* should be defined within the class.
- The casting operator function has no argument and no return type.
- The operator function should be either member function or friend function of a class.
- The compiler does not support automatic type conversions between user-defined data types. It can be done by defining casting operator functions.
- There are three possibilities of data conversions. These are: basic data type to class type, class type to basic type, and class type to another class type.
- The type conversion of one class to another class type can be done either by using casting operator function (in source class) or single argument constructor (in destination class).



10.12 FURTHER READING

1. E. Balaguruswami, *Object Oriented Programming with C++*, Tata McGraw Hill, Fifth Edition
2. Reema Thareja, *Object-Oriented Programming with C++*, Oxford Publication, Fifth Edition



10.13 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q No 1: (c)

Ans to Q No 2: (a)

Ans to Q No 3: (d)

Ans to Q No 4: (b)

Ans to Q No 5: (c)

Ans to Q No 6: (c)

Ans to Q No 7: (a) Operator overloading

(b) C1.operator+(C2)

(c) Unary

(d) two

Ans to Q No 8: (c)

Ans to Q No 9: (a)

Ans to Q No 10: (d)

Ans to Q No 11: (c)

Ans to Q No 12: (d)

Ans to Q No 13: (a) False (b) False (c) True (d) True



10.14 MODEL QUESTIONS

Q 1. What is operator overloading? Explain its importance.

Q 2. Write down the various steps followed in the implementation of operator overloading.

Q 3. Write down the syntax of the operator overloading function.

Q 4. What are the operators that cannot be overloaded?

Q 5. What are the two techniques that are used in operator overloading?

Q 6. What is type casting?

Q 7. Write down the syntax of type casting operator.

Q 8. Explain how the conversion of data from one class to another class is done?

Q 9. Explain how the conversion of data from basic type to a class is done?

Q 10. Illustrate how the conversion of data from class type to basic type is performed?

Q 11. Write a C++ program to show unary operator overloading using a friend function.

Q 12. Write a C++ program to show unary operator overloading using a member function.

Q 13. Write a C++ program to show binary operator overloading using a member function.

Q 14. Write a C++ program to show binary operator overloading using a friend function.

*** ***** ***

UNIT 11: INHERITANCE

UNIT STRUCTURE

11.1 Learning Objectives

11.2 Introduction

11.3 Inheritance

- 11.3.1 Defining Derived Classes
- 11.3.2 Accessing Base Class Members

11.4 Types of Inheritance

- 11.4.1 Single Inheritance
- 11.4.2 Multiple Inheritance
- 11.4.3 Hierarchical Inheritance
- 11.4.4 Multilevel Inheritance
- 11.4.5 Hybrid Inheritance
- 11.4.6 Multipath Inheritance
- 11.5 Virtual Base Classes
- 11.6 Abstract Class

11.7 Let Us Sum Up

11.8 Further Reading

11.9 Answers to Check Your Progress

11.10 Model Questions

11.1 LEARNING OBJECTIVES

After going through this unit, you will be able to –

- learn about the concept of inheritance in C++
- create new classes by reusing the members and properties of existing classes
- learn about the advantages and disadvantages of inheritance in programming
- learn how to use base class access specifiers *public*, *private* and *protected*
- learn about different types of inheritance in C++
- learn about the use of virtual base class and abstract class

11.2 INTRODUCTION

We have already been acquainted with the concept of operator overloading in the previous unit.

In this unit, we shall discuss one important and useful feature of Object-Oriented Programming (OOP) which is called ***inheritance***. C++ supports the concept of inheritance. We have already discussed the concept of classes and objects in our previous unit. The knowledge of class and objects is a prerequisite for this unit.

With the help of inheritance, we can reuse (or inherit) the properties of a previously written class in a new class. There are different types of inheritance that will be discussed in this unit. The concept of abstract and virtual base class will also be covered in this unit.

11.3 INHERITANCE

In Biological term, inheritance is the transformation of the hereditary characters from parents or ancestors to their descendent. In the context of Object-Oriented Programming (OOP), the meaning is almost the same. We have also learnt that reusability is one of the important features of OOP. It is always convenient if we could reuse something that already exists rather than create the same all over again.

The process of creating a new class from existing class or classes is called ***inheritance***. The newly created class is called ***derived class*** and the existing class is called the ***base class***. The derived class inherits some or all of the characteristics of the base class. Derived class may also possess some other characteristics which are not in the base class. Let us first describe the advantages and disadvantages of inheritance.

Advantages of Inheritance:

- ***Reusability***

  Reusability is an important feature of Object-Oriented Programming. We can reuse code in many situations with the help of inheritance. The base class is defined and once it is compiled, it need not be re-written. Using the concept of inheritance, the programmer can create

as many derived classes from the base class as needed. New features can also be added to each derived class when required.

- **Reliability and Cost**

  Reusability would not only save time but also increase reliability and decrease maintenance cost.

- **Saves Time and Effort**

  The reuse of class that has already been tested, debugged, and used many times can save us the effort of developing and testing the same again.

Disadvantages of Inheritance:

- In appropriate use of inheritance makes a program more complicated.
- In the class hierarchy, various data elements remain unused; the memory allocated to them is not utilized.

11.3.1 Defining Derived Classes

A derived class is defined by specifying its relationship with the base class in addition to its own details. The general form of deriving a new class from an existing class is as follows:

```
class Derived Class Name:access specifier Base Class Name
     {
      ..........
      .......... //members of derived class
      ..........
     };
```

The derived class name and the base class name are separated by a colon "**:**". We can derive classes using any of the three base class access specifiers: ***public***, ***private,*** or ***protected***. If we do not specify the access specifier, then by default it will be *private.* Generally, it is convenient to specify the access specifier while deriving a class. Access specifiers are sometime called *visibility mode*. It determines the access control of the base class members inside the derived class. In the previous unit, we have already learnt about *private* and

public access specifier while declaring classes. ***Protected*** specifier has a significant role in inheritance. For example, let us consider two classes namely, "**employee"** and "**manager**". What ever information is present in "employee" class, the same will be present in "manager" class also. Apart from that, there will be some extra information in "manager" class due to other responsibilities assigned to managers. Due to the feature of inheritance in C++, it is enough only to indicate those pieces of information which are specific to manager in its class. In addition, the "manager" class will inherit the information of "employee" class.

11.3.2 Accessing Base Class Members

There are three types of base class derivation:

- Public derivation
- Private derivation
- Protected derivation

- **Public derivation**

When the base class is inherited by using ***public access specifier*** then all public members of the base class become public members of the derived class, and all protected members of the base class become protected members of the derived class. The private members of the base class remain private and are not accessible by members of a derived class. Let us examine this with the following example:

//**Program 11.1:**Inherit base class with public access specifier.

```
#include<iostream.h>
class base
{
    private:
      int num1,num2;
    public:
      void input(int n1, int n2 )
       {
```

```
        num1=n1;
        num2=n2;
    }
   void display( )
   {
       cout<<"Number 1 is: "<<num1<<endl;
       cout<<"Number 2 is: "<<num2;
   }
};//end of base class

class derived : public base
{
    private:
        int num3;
    public:
        void enter(int n3)
        {
            num3=n3;
        }
        void show()
        {
            cout<<"\nNumber 3 is: "<<num3;
        }
}; //end of derived class
int main()
{
    derived d;     // d is an object of derived class
    d.enter(15); //enter() function is called by object d
    d.input(5,10);       /* accessing base class member. input() is
                    a public member of base class */
    d.display(); /* accessing base class member display() is a
                    public member of base class */
    d.show();
    return 0;
}
```

The output will be like this:

Number1 is: 5

Number2 is: 10

Number3 is: 15

In **Program 11.1**,***d*** is a derived class object. With the statement, '***d.input(5,15);',*** we have made a call to the function *input()* of base class by the derived class object. Similarly, we have also called *display()* member function of base class using derived class object d.

- **Private derivation**

When the base class is inherited by using ***private access*** specifier then all the public and protected members of the base class become private members of the derived class. Therefore, public members of base class canonly be accessed by the member functions of the derived class and they arenot accessible outside the derived class using objects of the derived class.

For example, in **Program 11.1,** if the base class is inherited using ***private access*** specifier by the derived class then it will give following two error messages while compiling.

Error:base::input(int,int) is not accessible

Error: base::display() is not accessible

As the base class is privately inherited, *input()* and *display()* become private to the derived class althogh they were public in the base class. So other functions like *main()* cannot access them. Statement like '**d.input(5,10);'** and '**d.display();'** will be invalid in that case.

Protected Members and Inheritance: Protected members provide greater flexibility incase of inheritance. By using ***protected*** instead of private declaration, we can create class members that are private to their class but that can still be inherited and accessed by derived class. A member declared as protected is accessible by the member functions with in its class and any class immediately derived from it.

When a ***protected*** member is inherited in ***public*** mode, it becomes ***protected*** in the derived class too and there fore is accessible by the member functions of the derived class. It also becomes ready for further inheritance. If a base class is inherited as ***private***, then the ***protected*** member of base class becomes ***private*** in the derived class. Although it is available to the member function of the derived class, it is not available for further inheritance since ***private*** members cannot be inherited.

/***Program11.2:*** Program showing protected members inherited in public mode*/

```
#include<iostream.h>
class base
{
     protected:
     int num1,num2;     /*private to base, protected to derived and accessible
                                by derived class member function*/
     public:
     void input(int n1, int n2 )
     {
          num1=n1;
          num2=n2;
     }
     void display( )
     {
          cout<<"Number 1 is: "<<num1<<endl;
          cout<<"Number 2 is: "<<num2;
     }
};   //end of base class
class derived : public base//base class is publicly inherited
{
     private:
     int s;
     public:
     void add( )
```

```
    {
        s=num1+num2 ;      /* derived class accessing base class pro-
                               tected member num1, num2 */
    }
    void show( )
    {
        cout<<"\nSummation is : "<<s;
    }
};  //end of derived class
int main()
{
    derived d;     // d is an object of derived class
    d.input(10,20);   /* accessing base class member. input() is a func-
                         tion of base class */
    d.display(); /* accessing base class member. display() is a function of
                    base class */
    d.add();
    d.show();
    return 0;
}
```

The above program will give the output as:

Number1 is: 10

Number 2 is: 20

Summation is : 30

The derived class member function *voidadd()* can access *num1* and *num2* of the base class because *num1* and *num2* are declared as *protected* and the base class accesss pecifier is *public*.

- **Protectedderivation**

When the base class is inherited by using ***protected access specifier***, then all protected and public members of base class become protected members of the derived class. Let us consider the following example:

//**Program 11.3**: Base class derived as protected

```
#include<iostream.h>
class base
{
    protected:
        int num1,num2;
    public:
        void input(int n1, int n2 )
        {
            num1=n1;
            num2=n2;
        }
        void display( )
        {
            cout<<"Number 1 is: "<<num1<<endl;
            cout<<"Number 2 is: "<<num2;
        }
};  //end of base class
class derived : protected base
{
    private:
        int s;
    public:
        void add( )
        {
            input(30,60);       /*member function add() of derived
                                class can access input() as it is inher-
                                ited as protected */
            s=num1+num2; // num1,num2 are inherited as
                                // protected, so add() can access
            }
            void showall( )
            {
                display();      //display() is inherited as protected
                cout<<"\nSummation is : "<<s;
```

```
};   //end of derived class
int main()
{
     derived d;
     /* d.input(10,20); [Invalid statement because
     input() is inherited as protected member of derived.
     main() can't access it] */
     d.add(); /* accessing base class member display()
                 is a function of base class */
     d.showall(); // public member of derived
     /* d.display(); [Invalid statement because
     display() can be accessible by derived class member
     function only */
     return 0;
}
```

The above program will give the following output:

Number1 is: 30

Number 2 is: 60

Summation is: 90

In the above program, we can observe that *input()*, *display()*, *num1*, *num2* of baseclass are inherited as protected to its derive class. The member functions *add()*, *show all()* of derived class can use them; as protected members are accessible by derive class members. But main() is not a member function and it cannot access *input()* and *display()*. Although *num1*, *num2* are protected to derived class, but they behave as private to base class.



CHECK YOUR PROGRESS

Q 1: Answer the following by selecting the appropriate option:

(i) By using protected, one can create class members that
- (a) cannot be inherited and accessed by a derived class
- (b) can be accessed by a derived class
- (c) can be public
- (d) none of these

(ii) Class members are by default
- (a) protected (b) public
- (c) private (d) none of these

(iii) When base class access specifier is protected, then public members of base class can be accessible by
- (a) member function of derived class
- (b) main() function
- (c) objects of derived class
- (d) none of these

(iv) Privated at a members of base class can be inherited by declaring them as
- (a) private (b) public
- (c) protected (d) none of these

(v) If we don't specify the visibility mode in base class derivation then by default it will be
- (a) protected (b) private
- (c) public (d) none of these

(vi) Privated at a members can be accessed
- (a) from derive class
- (b) only from the base class itself
- (c) both from the base class and its derived class
- (d) None of these

11.4 TYPES OF INHERITANCE

A program can use one or more base classes to derive a single class. It is also possible that one derived class can be used as base class for another class. Depending on the number of base classes and levels of derivation inheritance is classified into the following forms:

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance
- Multipath Inheritance

11.4.1 Single Inheritance

The programs discussed so far in this unit are examples of single inheritance. In ***single inheritance,*** the derived class has only one base class and the derived class is not used as base class. The pictorial representation of single inheritance is given in Figure11.1.

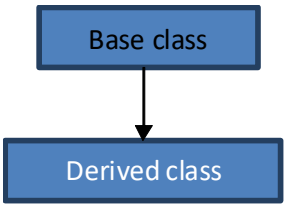


Figure11.1:Single Inheritance

The arrow directed from base class towards the derived class indicates that the features of base class are inherited to the derived class. In the following program, we have derived '*employee*' class from '*person'* class. Data member '*name*' and '*age*' are common to both of the two classes. '*name*' and '*age*' are declared as protected so that derive class can inherit 'name' and '*age*' from the base class '*person*'. Base class is inherited in public mode. Employee may have some other data like designation, salary. So the other data members of '*employee*' class are '*desig*' and '*salary*'.

/***Program11.4***: Single inheritance with protected data member and public inheritance of base class*/

```
#include<iostream.h>

#include<string.h>
class person //base class "person"
{
    protected:
    char name[30];   //protected to derived class 'employee'
    float age;
    public:
    void enter(char *nm, float a)/* base class member function */
    {
        strcpy(name,nm);
        age=a;
    }
    void display()   //base class member function
    {
        cout<<"Name: "<<name<<endl;
        cout<<"Age: "<<age<<endl;
    }
};
class employee : public person    /*base class "person" is
    publicly inherited by derived class "employee" */
 {
    private:
       float salary;
       char desig[20];
    public:
    void enter_data(char *n,char *d,float ag,float s)
    {
        strcpy(name,n);  /*"name" of base class can be accessible
                           by derived class member function*/
```

```
            strcpy(desig,d);
            salary=s;
                age=ag;  /*age can be accessible by enter_data() of derived class*/
                }

                void display_all()      //derived class member function
                {

                    display();    //can be used here as publicly inherited
                    cout<<"Designation: "<<desig<<endl;
                    cout<<"Salary:  "<<salary<<endl;


                }
};
int main()
{
    employee e1, e2;    // e1,e2 are objects of derived class "employee"
    person p;    // p is an object of base class "person"
    e1.enter_data("Raktim","Clerk",32,5000);
    cout<<"Employee Details....."<<endl;
    e1.display_all();
    e2.enter("Vaskar",41);     /*Derived class object e1
    accessing public member of base, enter() */
    e2.display();     /*derived class object e2 accessing
    public member of base, display() */
    cout<<endl<<"Person  Details ...."<<endl;
    p.enter("Pragyan",24);
    p.display();
    return 0;
}
```

Here, the derived class '***employee'*** uses '***name'*** and '***age'*** of base class '***person'*** with the help of derived class member function ***enter_data()***.

Two different classes may have member functions with the same name as well as the same set of arguments. But in the case of

inheritance, an ambiguous situation arises when base class and derived classes contain member functions with the same name. In main(), if we call member function of that particular name of base class with derived class object, then it will always make a call to the derived class member function of that name. This is because the function in the derived class *over rides* the inherited function. However, we can invoke the function defined in base class by using the ***scope resolution operator (::)*** to specify the class. For example, let usconsider the following program.

/***Program11.5**: Program where base and derived class has member functions with same name*/

```
#include<iostream.h>
class B
{
    protected:
    int p;
    public:
    void enter()
    {
      cout<<"\nEnter an integer:";
      cin>>p;
    }
    void show()
    {
      cout<<"\n\nThe number in Base Class is: "<<p;
    }
};
class D : public B
{
    private:
    int q,r;
    public:
      void enter() //overrides enter() of "B"
      {
          B::enter();
```

```
        cout<<"\nEnter an integer:";
        cin>>q;

}
        void show()
        {
            r=p*q;
            cout<<"\nEntered numbers in Base and
                Derived class are:"<<p<<"\t"<<q;
            cout<<"\n\nThe product is :"<<r;
        }
};
int main()
{
    D d;                    //d is an object of class derived class "D"
    d.enter();    //invokes enter() of "D"
    d.show();     //invokes show() of "D"
    d.B::show();  //invokes show() of "B"
    return 0;
}
```

In the above program, the function name *show()* is same in both base “B” and derived class “D”. To call show() of base class “B”, we have used the statement, ‘**d.B::show();’.** If we uses imply ‘**d.show();’** then it will invoke show() of derived class”D”.

When a derive class implements a function that has the same name aswell as the same set of arguments as the function in the base class, it is called ***function over riding***. When such a function is called through an object of derived class, then the derived class function would be invoked. However, that function in base class would remain hidden. But there are certain situations where function over riding plays an important role.

11.4.2 Multiple Inheritance

When one class is derived from two or more base classes then it is called ***multiple inheritance***. This type of inheritance allows us to combine the properties of more than one existing classes in a new class. Figure 11.2 depicts the multiple inheritance.

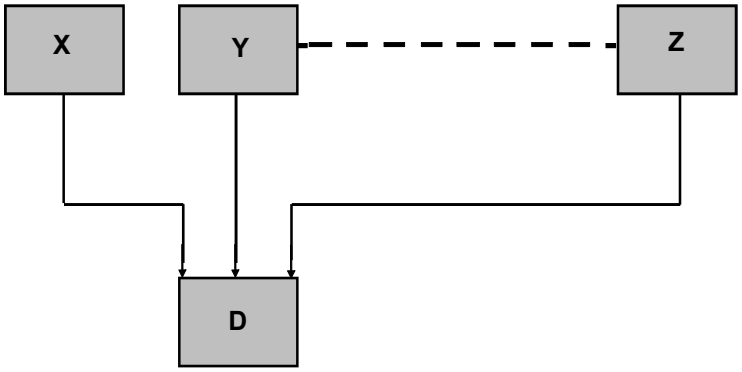


Figure 11.2: Multiple inheritance

We have to specify the base classes one by one separated by commas with their access specifiers to derive a new class from multiple base classes. The general form of deriving a derived class from multiple base classes is as follows:

```
class D : public X, public Y, public Z
{
     ......  //body of the derived class
};
```

In the above syntax, X,Y, Z are base classes and D is the derived class. There may be numbers of base classes in multiple inheritance which is indicated by the dotted line in the figure.

Fora demonstration of multiple inheritance, let us consider the following program. There are three base classes and one derived class. The derived class CHARACTER has one private member “n” and two public member functions “enter()” and “show()”. The function “enter()” is used to read anumber, a vowel, a consonant, and a symbol from the keyboard, and the “show()” function is used to display the contents on the screen. The class members of all three base classes are publicly derived.

//**Program11.6**: Example of Multiple inheritance

```
#include<iostream.h>
class V //base class
{
    protected:
        char v;
};
class C //base class
{
    protected:
        char c;
};
class S //base class
{
    protected:
        char s;
};
class CHARACTER : public V, public C, public S
{
    private:
        int n;
    public:
        void enter() //derived class member function
        {
            cout<<"\nEnter a  vowel:";
            cin>>v;//accessing protected member v of class "VOWEL"
            cout<<"\nEnter a consonent:";
            cin>>c; //accessing c of "CONSONENT" class
            cout<<"\nEnter  a  symbol:";
            cin>>s; //accessing s of "SYMBOL" class
            cout<<"\nEnter a number:";
            cin>>n; //accessing n of "NUMBER" class
    }
    void show()
    {   cout<<"\nThe entered characters are :\n\n";
        cout<<"\nVowel: "<<v;
        cout<<"\nConsonent: "<<c;
```

```
    cout<<"\nSymbol: "<<s;
    cout<<"\nNumber:  "<<n;
}
};
int main()
{
    CHARACTER  o; //o is an object of derived class "character"
    clrscr();
    o.enter();
    o.show();
    return 0;
}
```

One more suitable example of the implementation of multiple inheritance is shown in the program below:

/***Program11.7***: Program showing multiple inheritance with two base classes (practical, theory) and one derived class (result)/*

```
#include<iostream.h>
class practical  //base class "practical"
{
    protected:
    float p1_marks, p2_marks,total;
    public:
    void practical_marks()
    {
        cout<<"Enter marks of practical paper1 and
                paper 2: ";
        cin>>p1_marks>> p2_marks;
    }
    float add()  //returns the total of practical
    {
        total=p1_marks+p2_marks;
        return total;
    }
    void display_practical()
    {
```

```
            cout<<endl<<"Total Practical marks:"<<total;
        }
};      //end of "practical" class
class theory // base class "theory"
{
    protected:
    float phy, chem, math, total_marks;
    public:
    void theory_marks(){
    cout<<"Enter marks of Physics, Chem and Maths:";
    cin>>phy>>chem>>math;
}
float sum()
{
    total_marks=phy+chem+math;
    return total_marks;
}
void display_theory()
{
    cout<<endl<<"Total Theory marks:"<<total_marks;
}
};  //end of base class "theory"
class result : public practical, public theory
{
    protected:
    int roll;
    float grand_total,t,p;
    public:
    void enter() {
    cout<<"ENTER STUDENT INFORMATION....."<<endl;
    cout<<"Enter Roll no.:";
    cin>>roll;
    theory_marks();   //inherited publicly from base class "theory"
    practical_marks();     //inherited from base class "practical"
}
void theory_practical()
{
```

```
        t=sum();
        p=add();
        grand_total=t+p;
        cout<<"\nThe  total  marks  of  the  student  is:"
                    <<grand_total;
    }
    };  //end of derived class "result"
    int main()
    {
        result s1;   //object of derived class
        s1.enter();
              s1.theory_practical();// accessing derived class member
          s1.display_theory(); //s1 accessing "display_theory()" of"theory"
        s1.display_practical();
        return 0;
    }
```

When we execute the program then following output is displayed.

ENTER STUDENT INFORMATION.....

Enter Roll no. : 1

Enter marks of Physics, Chemistry and Mathematics: 657281

Enter marks of practical paper1 and paper2: 2526

The total marks of the student is: 269

Total Theory marks: 218

Total Practical marks: 51

The above program consists of three classes: two base classes ("practical"and "theory") and one derived class("result"). The member function "enter()" of derive class inherits member functions "practical_mark()" and "theory_ marks()" of base class "practical" and "theory" respectively. Similarly, member function "theory_practical() "uses" sum()" and "add()"of base class to calculate the "*grand_total*" marks of student. Thus, in the derived class we need not have to write functions for entering practical and theory marks. We just inherit them from the base classes.

11.4.3 Hierarchical Inheritance

Derivation of more than one class from a single base class is termed as ***hierarchical inheritance***. This is a very common form of inheritance inpractice. The rules for defining such classes are the same as in single inheritance. The pictorial representation of hierarchical inheritance is shown in Figure 11.3.

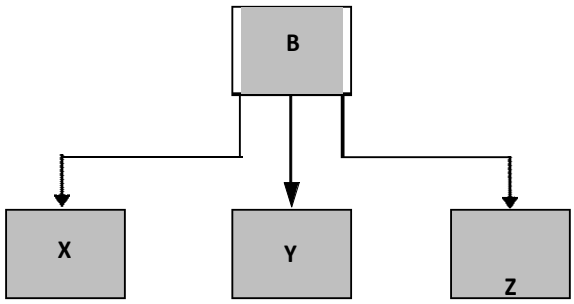


Figure11.3: Hierarchical Inheritance

For demonstration of hierarchical inheritance, let us consider a program with one base class (“student”) and three derived classes (“arts”, “science” and “commerce”).

//**Program11.8**: Demonstration of hierarchical inheritance

```
#include<iostream.h>
class student     // base class "student"
{
     protected:
     char  fname[20],lname[20];
     int  age,roll;
     public:
     void  student_info()
     {
          cout<<"Enter the first name and last name: ";
          cin>>fname>>lname;

          cout<<"\nEnter the Roll no.and Age: ";
         cin>>roll>>age ;

     }
```

```
    void display()
    {
        cout<<"\nRoll Number = "<<roll;
        cout<<"\nFirst and Last Name = "<<fname<<"\t"<<lname;
        cout <<"\nAge = " << age;
    }
};
class arts : public student   //derived class arts
{
    private:
    char asub1[20], asub2[20], asub3[20] ;
    public:
    void enter_arts()
    {
        student_info(); //base class member function
        cout <<"\n Enter the subject1 of the arts student:";
        cin >> asub1 ;
        cout<<"\nEnter the subject2 of the arts student:";
        cin >> asub2 ;
        cout<<"\nEnter the subject3 of the arts student:";
        cin >> asub3 ;
    }
    void display_arts()
    {
        display();//base class member function
        cout<<"\n\t Subject1 of the arts student="<< asub1;
        cout<<"\n\t Subject2 of the arts student="<< asub2;
        cout<<"\n\t Subject3 of the arts student="<< asub3;
    }
};

class commerce : public student   //derived class "commerce"
{

    private:
    char csub1[20], csub2[20], csub3[20] ;
    public:
```

```
void enter_com(void)
{
student_info();  //base class member function
cout<<"\tEnter the subject1 of the commerce student:";
cin>> csub1;
cout<<"\tEnter the subject2 of the commerce student:";
cin>>csub2 ;
cout<<"\tEnter the subject3 of the commerce student:";
cin>> csub3 ;
}
void display_com()
{
display();   //base class member function
cout<<"\nSubject1 of the commerce student=" <<csub1;
cout<<"\nSubject2 of the commerce student=" <<csub2;
cout<<"\nSubject3 of the commerce student="<< csub3
}
};
class science : public student  //derived class "science
{
private:
char ssub1[20], ssub2[20], ssub3[20] ;
public:
void enter_sc(void)
{
student_info(); //base class member function
cout<<"\nEnter the subject1 of science student:";
cin>>ssub1;
cout<<"\nEnter the subject2 of science student:";
cin>>ssub2 ;
cout<<"\nEnter subject3 of the science student:";
cin>>ssub3 ;
}
void display_sc()
{
display();   //base class member function
```

```
        cout<<"\nSubject1 of the science student="<<ssub1;
        cout<<"\nSubject2 of the science student="<<ssub2;
        cout<<"\nSubject3 of the science student="<<ssub3;
        }
};
int main()
{
  arts a ; //a is an object of derived class "arts"
  cout << "\n Entering details of the arts student\n";
  a.enter_arts( );
  cout <<"\nDisplaying the details of arts student\n";
  a.display_arts( );
  science s; //s is an object of derived class "science"
  cout <<"\n\n Entering details of science student\n";
  s.enter_sc( );
  cout<<"\n Display details of the science student\n";
  s.display_sc( );
  commerce c ;  //c is an object of derived class "commerce"
  cout<<"\n\nEnter details of commerce student\n";
  c.enter_com( );
  cout << "\n Display details of commerce student\n";
  c.display_com() ;
  return 0;
  }
```

11.4.4 Multilevel Inheritance

C++ also provides the facility of ***multilevel inheritance***, according to which the derived class can also be derived by another class, which in turn can further be inherited by another and soon. For instance, class X serves as a base class for class Y which in turn serves as base class for another class Z. The class Y which forms the link between classes X and Z is known as the ***intermediate base class***. Further, Z can also be used as a base class for another new class. The following figure depicts multi level inheritance.

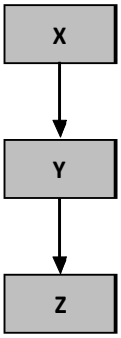


Figure 11.4: Multi level Inheritance

/***Program11.9:*** C++ program to demonstrate example of multi level inheritance.*/

```
#include<iostream.h>
class A    // A is a base class
{
    private:
     int a;
    public:
      void insert_a(int ia)
      {
         a = ia;
      }

      void display_a(void)
      {
        cout <<"Value of a is :" << a << endl;
      }
  };

class B: public A      // Class B is derived from class A
{
    private:
     int b;
    public:
void insert_b(int ia, int ib)
{
insert_a(ia); //assign value of a by calling function of class A
```

```
b = ib;
}
void display_b(void)
{
display_a();  //display value of a
cout << "Value of b is : " << b << endl;
}
};
class C: public B //class C is derived from class B
{
  private:
    int c;
  public:
    void insert_c(int ia, int ib,int ic)
    {
      /*Multilevel Inheritance Implementation: Assign
        value of a , b by calling function of class B
        and Class A. Here Class A is inherited by
        Class B, and Class B is inherited by Class C
        */
       insert_b(ia,ib);
       c = ic;
    }
      void display_c(void)
      {
        display_b(); //display value of a and b
        cout << "Value of c is : " << c << endl;
      }
  };



  int main()
  {
```

```
    C objC;     //create object of the final class, C
    objC.insert_c(10,20,30);
    objC.display_c();
    return 0;
}
```

11.4.5 Hybrid Inheritance

It is possible to derive a class involving more than one type of inheritance. ***Hybrid inheritance*** is that type of inheritance where several forms of in heritance are used to derive a class. There could be situations where we need to apply two or more types of inheritance to design a particular program. For example, let us assume that we are to design a program that will select players for a particular competition. For this purpose, we could consider four classes PLAYER, GAME, RESULT, and PHYSIQUE. PLAYER class contains the player details including name, address, location etc.GAME class can be derived from PLAYER class. Again, if weight age for physical test should be added before finalizing the result, then we can inherit that from PHYSIQUE. RESULT class is derived from two base classes GAME and PHYSIQUE. The following diagram gives us the inheritance relationship between various classes.

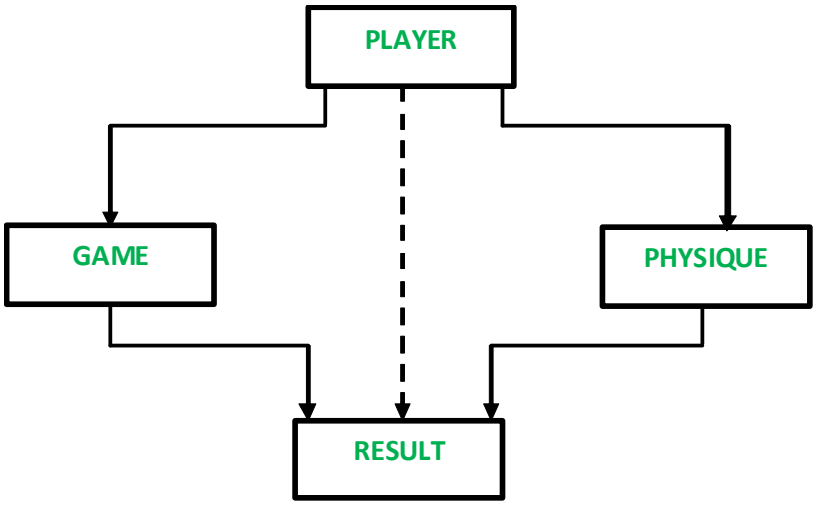


Figure11.5: Hybrid Inheritance

In the diagram, RESULT has two base classes,GAME, and PHYSIQUE. GAME is not only a base class but also a derived class. Here we can see that two types of inheritance multiple and multilevel are combined to create the RESULT class.

11.4.6 Multipath Inheritance

The inheritance where a class is derived from two or more classes, which are in turn derived from the same base class, is known as multipath inheritance. There may be many types of inheritance such as multiple, multilevel, hierarchical, etc. in multipath inheritance. Certain difficulties may arise in this type of inheritance.

For example, suppose we have two derived classes D and E that have a common base class B, and we have another class F that inherits from D and E.In that case it is considered as a multipath inheritance. The hybrid inheritance shown in figure 11.5 is also an example of multipath inheritance.For better illustration let us consider the following program.

//**Program11.10:** Demonstration of multipath inheritance

```
#include<iostream.h>

class B
{
    protected:
     int b;
};
class D : public B //B is publicly inherited by D
{
    protected:
      int d;
};
class E : public B //B is publicly inherited by E
{
```

```
    protected:
        int e;
};
class F : public D, public E //D,E are publicly inherited by F
{
    protected:
        int f;
    public:
    void enter_number()
    {
        cout<<"Enter some integer values for b,d,e,f:";
        cin>>b>>d>>e>>f;
    }

    void display()
    {
        cout<<"\nEntered numbers are:\n";
        cout<<"\nb="<<b<<"\nd= "<<d<<"\ne= "<<e<<"\nf= "<<f;
    }
};
int main()
{
    F obj;    //instantiaton of class F
    obj.enter_number(); //enter_number() of F is called
   obj.display();
    return 0;
}
```

Now, if we instantiate class F and call the functions ***enter_number()*** and ***display()***, then the compiler shows the following types error messages: Error M.cpp 28: Member is ambiguous: 'B ::b' and 'B::b' Error M.cpp33: Member is ambiguous: 'B::b' and 'B::b'

This is due to the duplication of members of class B in F. The member **b** of class B is inherited twice to class F: one throughclass D and

another through class E. This leads to ambiguity. To avoid such type of situation, ***virtual base class*** is introduced.

11.5 VIRTUAL BASE CLASSES

C++ provides the concept of ***virtual base class*** to over come the ambiguity occurring due to *multipath inheritance*. While discussing multipath inheritance, we have faced a situation that may lead to duplication of inherited members in the derived class F (Program11.10). This can be avoided by making the common base class(i.e.,B) as *virtual base class* while derivation. We can declare the base class B as ***virtual*** to ensure that D and E share the same data members of B.

This is shown in the following program which is the modification of the previous program 11.10.

/***Program11.11:** Virtual base class and removal of ambiguity occurred in multipath inheritance*/

```
#include<iostream.h>
#include<conio.h>
class B
{
    protected:
    int b;
};
class D:public virtual B
//B is publicly inherited by D and made virtual
{
    protected:
    int d;
};
class E:public virtual B
//B is publicly inherited by E and made virtual
{
    protected:
    int e;
```

```
};
class F : public D, public E
//D,E are publicly inherited by F
{
     protected:

     int f; public:
     void enter_number()
     {

          cout<<"Enter some integer values for b,d,e,f:";
          cin>>b>>d>>e>>f;
     }
     void display()
     {
          cout<<"\nEntered numbers are:\n";
          cout<<"\nb= "<<b;
          cout<<"\nd= "<<d;
          cout<<"\ne= "<<e;
          cout<<"\nf= "<<f;

     }
};
int main()
{
     F obj;  //instantiaton of class F
     clrscr();
     obj.enter_number();
     obj.display();
     getch();
     return 0;
}
```

Here we have used the key word virtual infront of the base class specifiers to indicate that only one object of type B, shared by class D and class E, exists. When a class is made as ***virtual base class***, C++ takes the necessary action so that only one copy of that class is inherited, regardless of how many paths exist between the virtual base class and a derived class.

11.6 ABSTRACT CLASS

The objects created often are the instances of a derived class but not of the base class. The base class just becomes a structure or foundation with the help of which other classes are built and hence such classes are called ***abstract class*** or ***abstract base class***. In other words, when aclass is not used for creating objects then it is called *abstract class*. In **Program11.10**, B is an abstract class since it was not used forcreating any object.



LET US KNOW

Inheritance and Constructors, Destructors

Although constructors are suitable for initializing objects, we have not used them in any program in this unit for the sake of simplicity. But if we use constructors in program, then we must follow certain definite rules while inheriting derive classes. If the base class contains no argument constructor, then the derived class does not require a constructor. If any base class contains parameterized constructor, then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructors. In case of inheritance, normally derived classes are used to declare objects. Hence it is necessary to define constructor in the derived class. When both the derived and base classes contain constructors, the base constructor is executed first and then the constructor in derived class is executed. Destructors are executed in the reverse order of constructor execution.



CHECK YOUR PROGRESS

Q 2: Select whether the following statements are True (T) or False (F):

(i) A class can serve as base class for many derived classes.

(ii) When one class is derived from another derived class then that is called *multiple inheritance*.

(iii) When one class is derived from more than one base class then that is called *multiple inheritance*.

(iv) When more than one form of inheritance is used in designing a class then that type is called *hybrid inheritance*.

Q 3: Answer the following by selecting the appropriate option:

(i) In multi level inheritance, the middle class acts as

(a) only derived class (b) only base class

(c) base class as well as derived class

(d) none of the above

(ii) A class is declared "virtual" when

(a) more than one class is derived

(b) two or more classes have common base class

(c) there are more than one base classes

(d) none of the above

(iii) When a class is not used for creating objects, it is called

(a) abstract class (b) virtual base class

(c) derived class (d) none of these

(iv) Intermediate base class is present in case of

(a) single inheritance (b) multiple inheritance

(c) multilevel inheritance (d) hierarchical inheritance



11.7 LET US SUM UP

The key points to keep in mind in this unit are:

- ***Inheritance*** is one of the most useful and essential characteristics of object-oriented programming language. If we have developed a class and we want a new class that is almost similar, but slightly different, the principles of inheritance come in handy. The existing class is known as ***base*** class and the newly formed class is known as ***derived*** class. The derived class can have some other characteristics which are not in base class.

- Private members of a class can not be inherited either in public mode or in private mode.
- When a public member is inherited in public, protected, and private mode, then in derived class it remains with the same access specifiers as in base class i.e., public, protected, and private respectively.
- A protected member inherited in public mode becomes protected, where as inherited in private mode becomes private inderived class.
- In ***single inheritance***, one new class is derived from a single base class.
- When a class is derived using the properties of several base classes, then it is called ***multiple inheritance.***
- The process of deriving a class from another derived class is called multilevel inheritance***.***
- More than one class can be derived from only one base class i.e., characteristics of one class can be inherited by more than one class. This is called ***hierarchical inheritance***.
- When different types of inheritance are involved in a single class then it is termed as ***hybrid inheritance***.
- When a class is derived from two or more classes, which are derived from the same base class, such type of inheritance is known as ***multipath inheritance***.
- We can 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 classes are declared as *virtual*, the compiler takes necessary caution to avoid the duplication of the member variables.
- When a class is not used for creating objects, it is called an ***abstract class***.



11.8 FURTHER READING

1. Venugopal, K.R. Rajkumar and Ravishankar T. *Mastering C++*. Tata McGraw Hill Publishing Co. Ltd., New Delhi.
2. Balagurusamy, E. (2001). *Object Oriented Programming with C++*,. 6e. Tata McGraw-Hill Education.
3. Kamthane, A.(2003). *Object-Oriented Programming with ANS Iand Turbo C++.* Pearson Education India.



11.9 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q.No.1:

(i) (b) can be accessed by a derived class

(ii) (c) private

(iii) (a) member function of derived class

(iv) (c) protected

(v) (b) private

(vi) (b) only from the base class itself

Ans. to Q.No.2: (i) True (ii) False (iii) True (iv) True

Ans. to Q.No.3:

(i) (c) base class as well as derived class

(ii) (b) two or more classes have common base class

(iii) (a) abstract class

(iv) (c) multi level inheritance



11.10 MODEL QUESTIONS

Q 1. What does inheritance mean in C++?

Q 2. What are the types of inheritance? Explain any three of them with examples.

Q 3. What are the different types of visibility modes of base class?

Q 4. Write a program to derive a class from multiple base classes.

Q 5. When do we make a class virtual?

Q 6. What are abstract base classes?

Q 7. Explain multipath inheritance.

Q 8. Write a C++ program involving appropriate type of inheritance which will inherit two classes *triangle* and *rectangle* from *polygon* class. Use member functions for entering appropriate parameters like *width*, *height* etc. and to calculate the area of triangle and rectangle.

Q 9. Consider a case of University having the disciplines of Engineering, Management, Science, Arts, and Commerce. There are many colleges in the University. Assuming a college can run a course pertaining to only one discipline, draw the class diagram. To which type of inheritance does this structure belong?

Q 10. Write a C++ program where class D is derived from class B. B has two public member functions *getdata()* and *showdata()* and D has two public functions *read data()* and *display all()*. Define the classes such that both function *getdata()* and *showdata()* should be accessible in the *main()* function.

*** ***** ***

UNIT 12: VIRTUAL FUNCTIONS AND POLYMORPHISM

UNIT STRUCTURE

12.1 Learning Objectives

12.2 Introduction

12.3 Virtual Functions

12.3.1 Pure Virtual Functions

12.4 Polymorphism

12.5 Categorization of Polymorphism Techniques

12.5.1 Compile-time Polymorphism

12.5.2 Run time Polymorphism

12.6 Let Us Sum Up

12.7 Further Reading

12.8 Answers to Check Your Progress

12.9 Model Questions

12.1 LEARNING OBJECTIVES

After going through this unit, you will be able to –

- learn the concept of virtual functions in OOP
- learn how virtual functions and dynamic binding can be implemented in C++
- define pure virtual functions to create abstract classes
- learn the concepts of polymorphism and its categories in OOP.

12.2 INTRODUCTION

In object-oriented programming, polymorphism is an important concept.Virtual function in object-oriented programming (OOP) is a part of the runtime polymorphism.A virtual function is an inheritable and overridable function.The compiler binds the virtual function at runtime. Binding a function means that wherever there is a function call, the compiler needs to know which function definition it should choose. The binding of a function can be early binding or late binding.

Early binding is a phenomenon wherein the decision to match various function calls to its definition happens at the compile time itself and the compiler directly associates the link with addresses.It is also known as Static Binding or Compile-time Binding.On the other hand, in late binding, the compiler identifies the required objects at runtime and then matches the function call with the correct functiondefinition.It is also known as Dynamic Binding or Runtime Binding. Late binding may be implemented by using the keyword, 'virtual' in the base class.

12.3 VIRTUAL FUNCTIONS

A virtual function is a member function that is declared within a base class and redefined in a derived class.A virtual function is a method that is declared and defined by using the keyword, '**virtual**'.When we try torefer a derived class object using a reference to the base class then we can call a virtual function for that object and execute the derived class's version of the function.Virtual functions ensure that the correct function is called for an object, regardless of the expression used to make the function call. This feature in OOP is called ***Function Overriding.***

The concept of function overriding allows defining functions with the same name and the same function signature (number of parameters and their data types) in both a base class and its derived class.But in this process, function definitions will be different. So, according to this concept, a derived class function is allowed to override a base class function by declaring and defining the base class function as virtual function by using the keyword 'virtual'.Function overriding is an implementation of the run-time polymorphism, which we would study in later sections of this unit. So, it overrides the function at the run-time of the program.

Let us consider the following C++ program to understand the concept of virtual function.

Program 12.1:

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

```
{
      public:
            void display()
            {
                  cout<< "\nDisplay Base";
            }
            virtual void show()
            {
                  cout<< "\nShow Base";
            }
};
classB: public A
{
      public:
            void display()
            {
                  cout<< "\nDisplay Derived";
            }
            void show()
            {
                  cout<< "Show Derived";
            }
};

int main ()
{
      A b; // Object b of the base class A declared
      B d; // Object d of the derived class B declared
      A *bptr;   // Pointer to base class A
cout<< "\nbptr points to object of the base class A:";
bptr=&b;
bptr->display();
bptr->show();
cout<< "\nbptr points to object of the derived class
B:";
```

```
bptr=&d;
bptr->display();
bptr->show();
return 0;
}
```

In the above example, class A is the base class and class B is derived from class A.Both the base class and the derived class have two functions with the same function names and prototypes. These functions are 'void display()' and 'void show()'. In the base class, the function, 'void show()' is declared and defined as virtual. Now in this type of situation, if we use a pointer to the base class to hold the address of a derived class object and try to use the pointer to call a derived class function thenby default the base class function will be executed if the function is not declared as virtual in the base class. So if we execute the above program, the following output will be displayed.

Output:

bptr points to object of the base class A:
Display Base
Show Base
bptr points to object of the derived class B:
Display Base
Show Derived

From the above output, it is observed that when pointer bptr points to the derived class object d and it is used to call the function 'void display()' then the base class function 'void display()'is executed. It is happened because the function 'void display()'is not declared as virtual in the base class A and as a result, it cannot be overridden by the corresponding derived class function.

Characteristics of a virtual function

- If the function is made virtual, then the compiler will determine which function is to execute at the run time based on the assigned address to the pointer of the base class.
- The classes which contain virtual functions are not abstract classes.

12.3.1 Pure Virtual Functions

Pure virtual functions are a specific type of virtual functions.A virtual function with no function body is called a pure virtual function.Such functions exist in the base class only to be overridden by the derived class functions.The base class only specifies that there is a function with this name and signature which would be implemented in the derived class.Such classes are also called as abstract base classes.In simple words, an abstract class is defined as a class with one or more pure virtual functions.Thus, an abstract class is one that is not used to create any object of its own but it solely exists to act as a base class for the other classes, which means the abstract class must be a part of some inheritance hierarchy.

So, a pure virtual function has no implementation. We only declare it. There are two ways of creating a pure virtual function as shown below.

```
virtual void display( ) = 0;
        Or,
virtual void display( ) {    }
```

A pure virtual function declared in a class must be defined in its derived class. However, if the derived class does not contain the definition of any pure virtual function then it will not show any error but the derived class becomes an abstract class.Let us consider the following program to implement pure virtual function concept.

Program 12.2:

```
#include<iostream.h>
class A
{   public:
     virtual void display() = 0; //Pure virtual function
};
class B : public A
{
```

```
      public:
void display () //Overriding the pure virtual function
{
      cout<< "\nThe display is from derived class B"
}
};
class C : public A
{
      public:
            void display()        // Overriding the pure
virtual function
{
      cout<< "\nThe display is from derived class C"
}
};

int main()
{
      B objB;    //Object of B class created
      C objC;    //Object of C class created
      objB. display();
      objC.display();
      return 0;
}
```

Output:

```
The display is from derived class B
The display is from derived class C
```

Characteristics of a pure virtual function

- Pure virtual functions need to be redefined in the derived class as they have no definitions in the base class.
- A class having a pure virtual function cannot be used to create objects of its own.



CHECK YOUR PROGRESS

Q 1: Fill In the blanks

i. A virtual function is a member function within the base class that we redefine in a _____ class.

ii. A virtual function is declared using the _____ keyword.

iii. Virtual Function is a member function of the base class which is _____ in the derived class.

iv. Virtual functions ensure that the _____ function is called for an object, regardless of the expression used to make the function call.

v. The phenomenon in which the decision to match various function calls happens at the compile time itself is called _____.

vi. Late binding may be implemented by using _____ keyword in the base class.

vii. A class with one or more pure virtual functions is called an _____ class.

viii. A pure virtual function in C++ is a virtual function for which we don't have an implementation, we only _____ it.

12.4 POLYMORPHISM

The term "Polymorphism" is a Greek word. It is a combination of two words, "poly", which means many, and "morphs" which means forms. Thus, the word polymorphism means having many forms.

It is one of the OOPs features that allow us to perform a single action in different ways.Polymorphism is the ability of an object to take on many forms. A real-life example of polymorphism is that a person at the same time can have different characteristics. A woman at the same time can be a mother, a wife, a daughter, and an employee. So the same person possesses different behavior in different situations. This is called polymorphism.

The behavior of an operator may depend upon the types of data used in the operation. For example, consider the operation of addition. If the operands are two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation. The process of making an operator to exhibit different behaviors in different instances is known as operator overloading.

A single function name can be used to define multiple functions with different numbers of argumentsor different types of arguments or both.This is like a particular word that may have several different meanings depending upon the context.Using a single function name to perform multiple tasks is known as function overloading. This can be illustrated by the following figure.

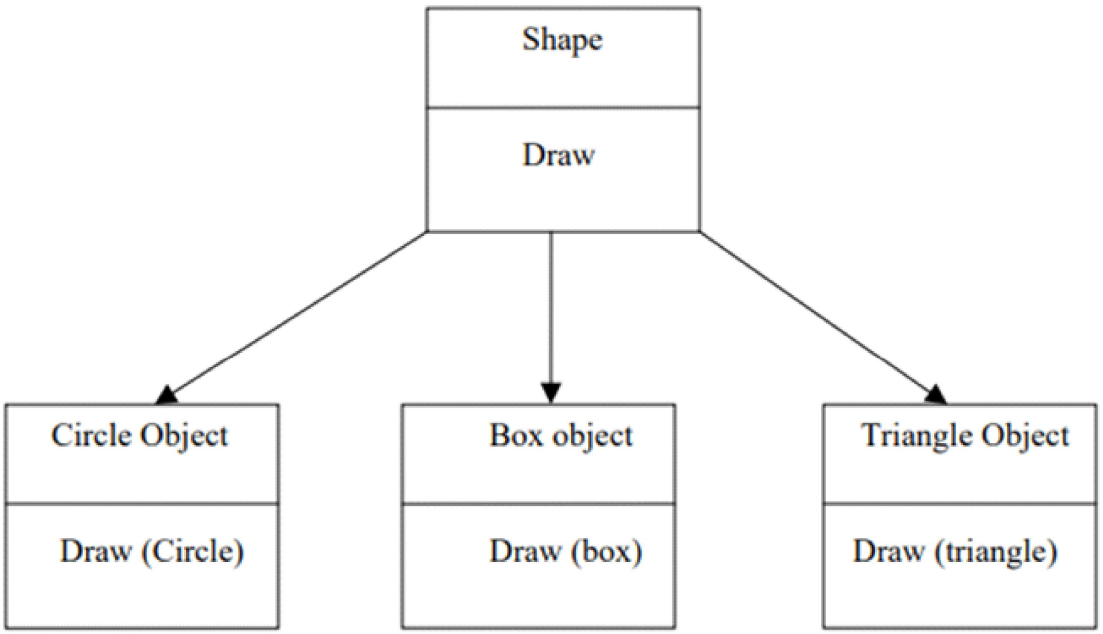


Figure 12.1: Example of Function Overloading

A base class and it's derive class may contain functions with same name, number of parameters and type of parameters. In this situation, the base class function is overridden by the corresponding derive class function. This is another form of polymorphism.

12.5 CATEGORIZATION OF POLYMORPHISM TECHNIQUES

There are two types of polymorphism:

1) Compile time Polymorphism – This is also known as static (or early) binding.
2) Runtime Polymorphism – This is also known as dynamic (or late) binding.

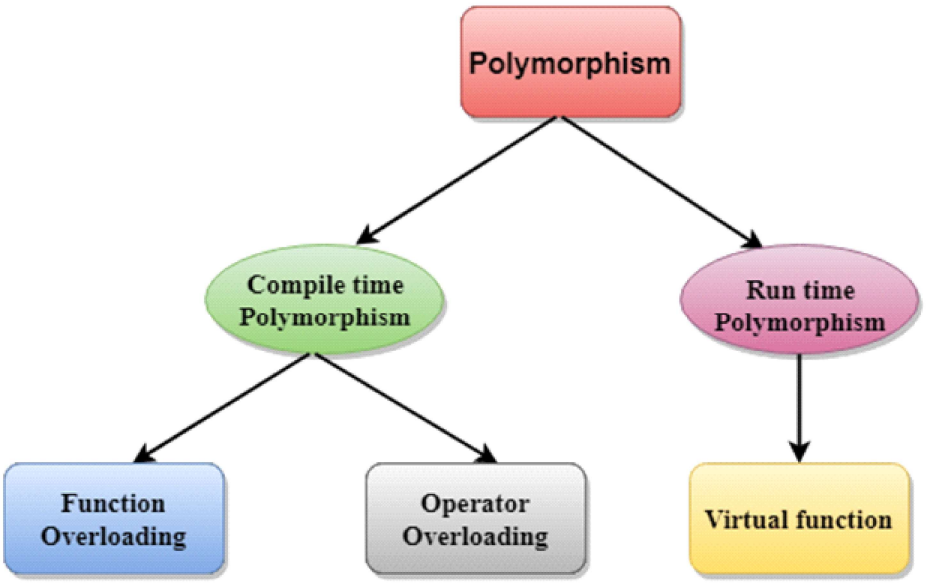


Figure 12.2: Classification of Polymorphism

Compile-time polymorphism can be achieved by function overloading and operator overloading.Run time polymorphism can be achieved by virtual functions. Virtual functions have been discussed in the previous sections of this unit. Compile-time and run time polymorphism will be discussed in the following sub-sections.

12.5.1 Compile-Time Polymorphism

Compile-time polymorphism is achieved by static binding. We have already learnt that function overloading and operator overloading are the two types of compile-time polymorphism.So, in case of function overloading and operator overloading, linking or binding of

a proper function definition to its function call is implemented at the time of compilation. Compile-time polymorphism provides fast execution because necessary things are known early at compile time.

Function Overloading:

When there is more than one function with the same function name but different functionalities then these functions are said to be overloaded. Functions can be overloaded by making changes in the number of arguments or/and in the type of arguments. So, during a function call for an overloaded function, the correct or proper function definition is linked with the call at the compile-time depending upon the matching type and number of arguments provided in the call.It means that based on the type of parameters and number of parameters provided in a function call determines which overloaded function is to be called at the compile-time.This is why it is considered as a compile-time polymorphism. So, in case of function overloading, different output may be obtained in different conditions. Let us consider the following simple C++ program to understand the concept of function overloading.

Program 12.3:

```
#include<iostream.h>
class Addition
{
      public:
            int sum (int num1, int num2)
             {
                  return num1+num2;
             }
            int sum(int num1, int num2, int num3)
            {
                  return num1+num2+num3;
            }
```

```
            float sum(float num1,float num2)
            {
                  return num1+num2;
            }
};
int main()
{
      Addition obj;
      float num1 = 4.5,num2 = 7.8;
      cout<<"\nOutput = "<<obj.sum(10, 20);    //Call the
first function
      cout<<"\nOutput = "<<obj.sum(11, 22, 33);
//Call the second function
      cout<<"\nOutput = "<<obj.sum(num1,num2);   // Call
the third function
return 0;
}
```

Output:

Output = 30

Output = 66

Output = 12.3

In the above example, a single function name, '*sum'* acts differently in the three different situations.

Let us consider another example of function overloading presented as follows.

Program 12.4:

```
#include<iostream.h>
class Swap
{

      public:
            void interchange(int num1,int num2)
             {
            int temp;
            temp = num1;
```

```
                num1 = num2;
                num2 = temp;
            cout<<"\nAfter swapping integer numbers:";
            cout<<"\nnum1="<<num1;
            cout<<"\nnum2="<<num2;
             }
            void interchange(float fnum1,float fnum2)
             {
            float temp;
            temp = fnum1;
                fnum1 = fnum2;
                fnum2 = temp;
            cout<<"\nAfter   swapping   floating   point
numbers:";
            cout<<"\nfnum1="<<fnum1;
            cout<<"\nfnum2="<<fnum2;
             }
            void interchange(char chr1, char chr2)
             {
            char temp;
            temp = chr1;
                chr1 = chr2;
                chr2 = temp;
            cout<<"\nAfter swapping character values:";
            cout<<"\nchr1="<<chr1;
            cout<<"\nchr2="<<chr2;
             }
};

int main()
{
   Swap obj;
int num1 = 10, num2 = 20;
char chr1 = 'W', chr2 = 'S';
float fnum1 = 3.7, fnum2 = 8.9;
```

```
cout<<"\nBefore swapping integer numbers:";
cout<<"\nnum1="<<num1;
cout<<"\nnum2="<<num2;
obj.interchange(num1,num2);  //  Integer  numbers  are
swapped

cout<<"\nBefore swapping floating point numbers:";
cout<<"\nfnum1="<<fnum1;
cout<<"\nfnum2="<<fnum2;
obj.interchange(fnum1,fnum2); // Floating point numbers
are swapped

cout<<"\nBefore swapping character values:";
cout<<"\nchr1="<<chr1;
cout<<"\nchr2="<<chr2;
obj.interchange(chr1,chr2);  //  Character  values  are
swapped
return 0;
}
```

Output:

Before swapping integer numbers:

num1=10

num2=20

After swapping integer numbers:

num1=20

num2=10

Before swapping floating point numbers:

fnum1=3.7

fnum2=8.9

After swapping floating point numbers:

fnum1=8.9

fnum2=3.7

Before swapping character values:

chr1=W

chr2=S

After swapping character values:

chr1=S

chr2=W

Operator Overloading:

Object-oriented programming also provides an option to overload operators.For example, the addition operator ('+') is used to add two numbers and it can also be used to concatenate two strings by operator overloading technique. In operator overloading, new operation can be defined for an operator. At compile-time, the defined operation of the operator will be called depending upon the operand or operands applied to it.Both arithmetic and non-arithmetic operators can be overloaded. The precedence and associativity of operators remain the same after and before operator overloading. Also, operator overloading does not change the basic meaning and arity of any operator. Arity means the number of operands required by an operator to perform its action.

In unit 10, operator overloading is already discussed in detail.

12.5.2 Run-Time Polymorphism

Run time polymorphism is also known as dynamic binding or late binding. It is achieved when the object's method is invoked at the run time instead of compile time. This type of polymorphism is achieved by Function Overriding.

We have already learnt in the earlier sections that function overriding is implemented through pointers and virtual functions. Function overriding is a run time polymorphism where more than one function is having the same name, number of parameters, and type of parameters. In this process, a member function of a base class can be redefined or overridden in its derived class.So, only

the function definition of the derived class function is different from the corresponding base class function. Example of runtime polymorphism or function overriding is provided in section 12.3.



CHECK YOUR PROGRESS

Q 2: Fill In the blanks

i. The ability of an object to take on many forms is called _____.

ii. The two types of polymorphism are _____ and _____.

iii. Compile time polymorphism can be achieved by _____ and _____.

iv. Run time polymorphism can be achieved with the help of _____.

v. In function overloading, names of the functions can be same, but their parameters are _____.

vi. The precedence and associativity of operators remains the same after and before _____.

vii. The number of operands, an operator requires to perform its action is called _____.

viii. The run time polymorphism where more than one function is having the same name, number of parameters and the type of the parameters is called as_____.



12.6 LET US SUM UP

- Binding a function means that wherever there is a function call, the compiler needs to know which function definition it should choose. The binding of a function can be early binding or late binding.
- A virtual function is a member function that must be declared inside the base class. It is declared by using the keyword '*virtual*' in the class definition.

- Late binding occurs only with virtual functions, and only when a pointer of the base class where those virtual functions exist is used to hold the address of its derived class object.
- The redefinition of a virtual function in a derived class is usually called function overriding.
- The concept of function overriding allows defining functions with same name and the same function signature (parameters and their data types) in both a base class and its derived class with different function definitions.
- Virtual functions ensure that the correct function is called for an object.
- The classes which are containing virtual functions are not abstract classes.
- Pure virtual functions are a specific type of virtual function. A virtual function with no function body is called a pure virtual function.
- A pure virtual function has no implementation. It is only declared.
- An abstract class is one that is not used to create any object of its own but it solely exists to act as a base class for the other classes.
- The word polymorphism means ‘having many forms’. It is the ability of an object to take on many forms.
- Polymorphism can be achieved in Object-Oriented Programming by overloading functions and operators.
- The process of making an operator to exhibit different behaviors in different instances is known as operator overloading.
- The precedence and associativity of operators remain the same after and before operator overloading.
- Operator overloading does not change the basic meaning of any operator.
- When there is more than one function with the same name but different parameters then these functions are said to be overloaded.
- There are two types of polymorphism: Compile time Polymorphism and Runtime Polymorphism

- Compile-time polymorphism can be achieved by function overloading and operator overloading.
- Run time polymorphism can be achieved by virtual functions.
- Run time polymorphism is achieved when the object's method is invoked at the run time instead of compile time. This is achieved by Function Overriding..



12.7 FURTHER READING

1. Bajarane, S. (1997). The C++ programming language.
2. Schildt, H. (2004). *C++: The complete reference*. New York: McGraw-Hill.
3. Venugopal, K. R., & Rajkumar, T. (2004). Ravishankar: Mastering C++.
4. Balagurusamy, E. (2001). *Object-Oriented Programming with C++, 7e*. McGraw-Hill Education.
5. SAMANTA, D. (2006). *Object-oriented Programming with C++ and Java*. PHI Learning Pvt. Ltd.
6. Jana, D. (2014). *C++ and Object-Oriented Programming Paradigm*. PHI Learning Pvt. Ltd..
7. www.learncpp.com/cpp-tutorial



12.8 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q.No.1:

i. Derived	ii. Virtual	iii. Overridden
iv. Correct	v. Early binding	vi. Virtual
vii. Abstract	viii. declare	

Ans to Q.No.2:

i. Polymorphism

ii. Compile-time Polymorphism, Runtime Polymorphism

iii. function overloading, operator overloading

iv. virtual functions

v. Different

vi. operator overloading

vii. Arity

viii. Function overriding



12.9 MODEL QUESTIONS

Q1. Explain briefly virtual function and pure virtual function.

Q2. Describe the concept of polymorphism. Explain its different types.

Q3. What are early binding and dynamic binding?

Q4. What is an abstract class?

*** ***** ***

UNIT 13: FILE HANDLING

UNIT STRUCTURE

13.1 Learning Objectives

13.2 Introduction

13.3 File Classes

13.4 Opening and Closing a File

13.5 Functions for I/O operations

- 13.5.1 get() and put()
- 13.5.2 read() and write()
- 13.5.3 getline()
- 13.5.4 eof()

13.6 Manipulation of File Pointers

- 13.6.1 seekg()
- 13.6.2 seekp()
- 13.6.3 tellg()
- 13.6.4 tellp()

13.7 Let Us Sum Up

13.8 Further Reading

13.9 Answers to Check Your Progress

13.10 Model Questions

13.1 LEARNING OBJECTIVES

After going through this unit, you will be able to –

- describe the different file classes
- write programs for opening and closing files
- use functions like get(), put(), read(), write() for I/O operations
- use functions like getline() and eof() in I/O operations
- use functions like seekp(), seekg(), tellp(), tellg() for manipulation of file pointers

13.2 INTRODUCTION

In the previous unit, we have learnt about virtual functions and pure virtual functions. We have also learnt about polymorphism and the different types of polymorphism in detail.

In this unit, we will learn about the different types of file classes. We will also learn how to open and close a file. Different functions like likeget(), put(), read(), write(), getline() and eof() for performing I/O operations are discussed in detail in this unit. Functions like seekg(), seekp(), tellg() and tellp() which are used to manipulate file pointers are also discussed in this unit. We will learn about the concept of templates in detail in the next unit.

13.3 FILE CLASSES

File handling in C++ refers to the use of files and performing various operations on the files. To perform file input-output, we must include the header <*fstream*> in our program. There are several file classes: *ifstream*,*ofstream* and *fstream*. Input from a keyboard is managed by the *istream* object *cin* and the output to a screen is managed by the *ostream* object *cout*. Similarly, input from a file is managed by the *ifstream* object and the output of a file is managed by the *ofstream*. So just in earlier programs where we have included the header file <*iostream*> to use functions like *cin* and *cout*, we will now use the header file<*fstream*> to read from and write to files. Since these classes are declared in the *fstream*, we will need to include the <*fstream*> header to perform any operations on files.

Table 13.1: File Stream Classes

ata Type	Description
ifstream	Represents the input file stream and is used to perform file input operations
ofstream	Represents the output file stream and is used to perform file output operations
fstream	Represents the file stream and is used to perform both file input and output operations

13.4 OPENING AND CLOSING A FILE

In C++, a file is opened after linking it to a stream. There are three types of streams: input, output, and input/output. To create an input stream we must declare the stream to be of class **ifstream**. To create an output stream we must declare the stream to be of class **ofstream**. To create both an input and output stream together, we must declare the stream to be of class **fstream**. The following lines of code create one input stream, one output stream, and one stream capable of both input and output.

```
ifstream in;     //input
ofstream out;    //output
fstream io;      //both input and output
```

File Modes

Once the stream has been created, we need to associate a file to the stream by using the **open()** function. The open() function is a member of all the three streams. The prototype for each stream is shown below:

```
void ifstream::open(const char *filename,
ios::openmode mode = ios::in);
void ofstream::open(const char *filename,
ios::openmode mode = ios::out | ios::trunc);
void fstream::open(const char *filename,
ios::openmode mode = ios::in | ios::out);
```

Here, filename is the name of the file. The value of mode determines how the file is to be opened. It must be one or more of the given values defined by openmode. The different file modes and their meaning are listed in the table below.

Table 13.2: File modes

Constant	Meaning
ios :: in	Opens file for reading and specifies that the file is capable of input
ios :: out	Opens file for writing and specifies that the file is capable of output
ios :: app	Causes all output to that file to be appended to the end and can be used only with files capable of output
ios :: ate	Causes an initial seek to the end of file but I/O operations can still occur anywhere within the file
ios :: binary	Opens the file in binary mode.
ios :: nocreate	Causes the open() function to fail if the file does not already exist. It will not create a new file with that name.
ios :: noreplace	Causes the open() function to fail if the file already exists. This is used when you want to create a new file at the same time
ios :: trunk	Causes the contents of a pre-existing file by the same name to be destroyed and the file is truncated to zero length

The *fstream* class does not provide a mode by default and so we must specify the mode explicitly when we use an object of the *fstream* class. We can also combine two or more file modes constants by ORing these modes together. For example the following statements:

```
ofstreamout;
out.open("File1", ios::app |ios::nocreate);
```

The above statements will open a file in the append mode if the file exists and will abandon the file opening operation if the file does not exist.

By default, all files are open in text mode. By using *ios::binary*, we can open files in binary mode. For example, the following statements:

```
ofstream out;
out.open("File1", ios::binary);
```

Files contain both formatted text and raw data and can be opened either in text mode or binary mode. But when we open a file in text mode, various

character translations like carriage return being converted to newlines might take place. Whereas we when open in binary such character translations do not occur.

Using Constructor to open a file

Constructors of a class initialize their object when it is created. Likewise, constructors of the stream classes (ifstream, ofstream, fstream) can be used to initialize file stream objects with filenames that are provided to them. To open a file say 'file1' as an input file, we can create an input file stream like below:

```
ifstreamfin("file1", ios::in);
```

The above statement creates an object ***fin*** of input file stream and the file named "file1" is attached to the input stream ***fin***.

Similarly, opening an output file can be done by creating output file stream like below:

```
ofstreamfout("file1", ios::out);
```

The above statement creates an object ***fout*** of output file stream and the file named "file1" is attached to the output stream ***fout***.

Closing a file

The **close()**function is used to close the connection to a file. The general syntax for closing a file is:

```
stream_object.close();
```

For example we can close the file linked to a stream say named "fin" by using a statement like:

```
fin.close();
```

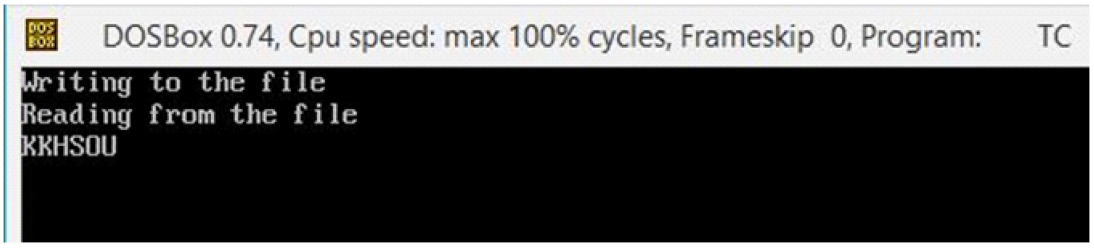
The close() function does not take any parameter and also returns no value.

Program 13.1: Program to open and close a file

```
#include<fstream.h>
#include<iostream.h>
int main () {
char write_data[10] = "KKHSOU";
```

```
char read_data[10];
//***Writing into a File***
ofstreamoutfile; //open file in write mode
outfile.open("file_to_open_close.txt");
cout<< "Writing to the file" <<endl;
outfile<<write_data<<endl; //write into the file
outfile.close();//close the opened fil
//***Reading from a File***
ifstreaminfile; //open a file in read mode
infile.open("file_to_open_close.txt");
cout<< "Reading from the file" <<endl;
infile>>read_data;
cout<<read_data<<endl; //write the data
                                //at the screen
infile.close(); //close the opened file
return 0;
}
```

Output:



In program 13.1 it is shown how to open a file and close a file. The file is first opened using "ofstream" and the text "KKHSOU" is written into the file and thenthe file is closed. The same file is then once again opened using "ifstream", the contents of the file are read and the file is then closed. Messages about writing and reading from the file are displayed on the screen when the said operations are being performed.

13.5 FUNCTIONS FOR I/O OPERATIONS

We are already familiar with the different file stream classes and member functions that are used to perform input and output operations on files. One way to read and write data is by using the member functions get() and put().Another way to read and write blocks of binary data is to use read() and write() functions.

13.5.1 get() and put()

The get() function will read a character and the put() function will write a character to a file. Let us look at the member functions get() and put() in detail.

get()

The **get()** function reads a single character from the input stream. The syntax of the **get()** is shown below:

```
istream&get(char  &ch);
```

The get() function reads a single character from the input stream and puts that value in *ch*.

put()

The **put()** function writes a single character to the output stream. The syntax of the **put()** is shown below:

```
ostream&put(char  ch);
```

The put() function writes *ch* to the stream and returns a reference to the stream.

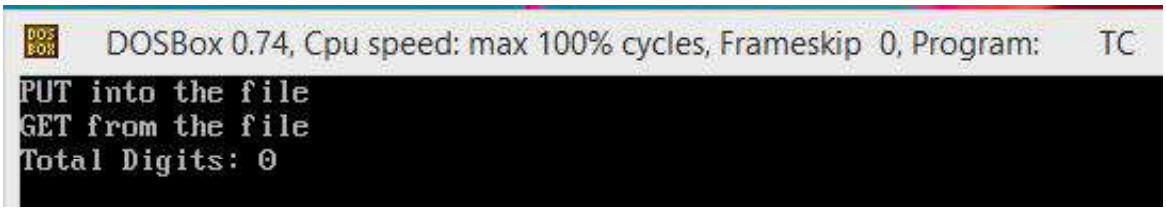
Program 13.2: Program to read and write characters from a file using get() and put()

```
#include<fstream.h>
#include<iostream.h>
#include<conio.h>
#include<ctype.h>
int main () {
      char put_data[10] = "KKHSOU";
      char ch;
```

```
        int count=0;
        clrscr();

        //***PUT into a File***
        fstream file; //open file to put
        file.open("file_to_put_get.txt",ios::app);
        cout<< "PUT into the file" <<endl;
        for(int  i=0;put_data[i]!='\0';i++)
        {
              file.put(put_data[i]); //put into the file
        }
        file.close();
        //***GET from a File***
        file.open("file_to_put_get.txt",ios::in);
        cout<< "GET from the file" <<endl;
        while(file)
        {
              file.get(ch);
              if(isdigit(ch))
              count++;
        }
        file.close(); //close the opened file
        cout<< "Total Digits: " << count <<endl;
        getch();
        return 0;
}
```

Output:



In the above Program 13.2, put() and get() functions are used to write and read characters into and from afile. The word “KKHSOU” is written into thefile “file_to_put_get.txt” one character at a time, by using the put() function. The get() function is then used to read the text from the file character by character. Since only text is written in the file, so when we check for the total digits in the file it is zero.

13.5.2 read() and write()

The read() function is used to read blocks of binary data from a file and the write() function is used to write blocks of binary data to a file in C++. Let us look at the member functions read() and write() in detail.

read()

The syntax of the **read()** function is shown below:

```
istream&read(char *buf, streamsizenum);
```

The read() function reads num characters from the invoking stream and puts them in the buffer pointed to by *buf*. Streamsize is a type defined in the C++ library and is capable of holding the largest number of characters that can be transferred in any I/O operation.

write()

The syntax of the **write()** function is shown below:

```
ostream&write(const char *buf, streamsizenum);
```

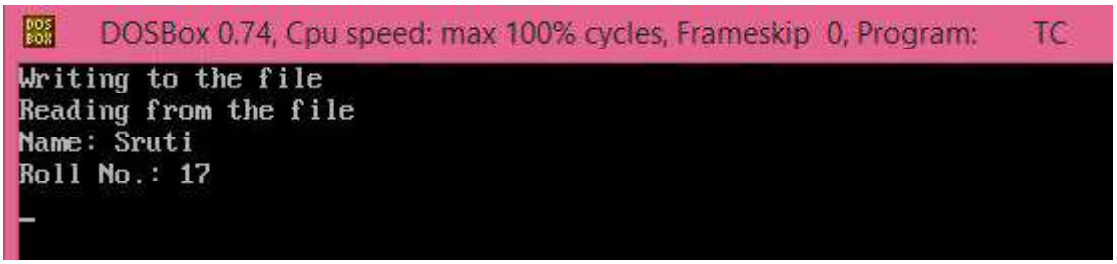
The write() function writes num characters to the invoking stream from the buffer pointed to by *buf*.

Program 13.3: Program to read and write data toand from a file using read() and write()functions

```
#include <fstream.h>
#include <iostream.h>
class student
{
     private:
```

```
            char *name;
            int roll_no;
      public:
            void putdata();
            void getdata();
};
void student :: putdata(){
      name="Sruti";
      roll_no=17;
}
void student :: getdata(){
      cout<< "Name: " << name <<endl;
      cout<< "Roll No.: " <<roll_no<<endl;
}
int main () {
//***Writing into a File***
ofstreamoutfile; //open file in write mode
outfile.open("file_to_read_write.txt", ios::out);
cout<< "Writing to the file" <<endl;
student std1;
std1.putdata();
//write data into the file
outfile.write((char*) & std1, sizeof(std1));
outfile.close();//close the opened file
//***Reading from a File***
ifstreaminfile; //open a file in read mode
infile.open("file_to_read_write.txt", ios::in);
student std2;
cout<< "Reading from the file" <<endl;
            //read data from the file
infile.read((char*) & std2, sizeof(std2));
std2.getdata(); //write the data at the screen
infile.close(); //close the opened file
return 0;
}
```

Output:



In program 13.3, we have defined a class “student” that has two data members “name” and “roll_no” and two member functions “putdata” where we assign student information and “getdata” where we display the student information to the screen.The file “file_to_read_write.txt” is opened using “ofstream” and the student information is written into the file by using the write() function and then the file is closed. The file “file_to_read_write.txt” is then opened again using “ifstream” and the student information is read from the file using read() function and then the file is closed.

13.5.3 getline()

Another function to read data is **getline()** function. It is a member function of each input stream class. The syntax for **getline()** is shown below:

```
istream&getline(char *buf, streamsizenum);
istream&getline(char *buf, streamsizenum, char delim);
```

The first syntax form reads characters intothe array pointed by *buf* until *num*-1 characters have been read, a newline or end of file is encountered.

The second syntax form reads characters into the array pointed by *buf* until *num*-1 characters have been read, the character specified by *delim* has been found or end of file is encountered.

13.5.4 eof()

In C++, whether the end of the file has been reached or not can be detected using the function **eof().** The syntax of the **eof()** function is shown below:

```
bool eof();
```

The **eof()**function returns true when the end of file has been reached; otherwise it returns false.



CHECK YOUR PROGRESS

Q 1: Whatare the three types of streams for file handling?

Q 2: What is the importance of using binary mode in open() function?

13.6 MANIPULATION OF FILE POINTERS

C++ uses the concepts of **get** pointer and **put** pointer. The get pointer for a stream indicates the position in the stream from which characters are extracted. And the put pointer for a stream indicates the position in the stream where characters are inserted. The get and put pointers reference positions between the characters of the stream and not the characters themselves. For example, consider the following characters as a stream where the positions of get and put pointers are marked in figure 13.1 below:

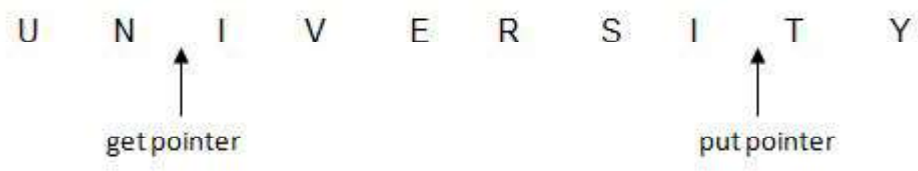


Figure 13.1: Positions of get and put pointers

In the example above, the next character extracted from the stream is 'I', and the next character inserted into the stream replaces the 'T'.

We can use member functions of the various stream classes to move the get or put pointer. Some of the member functions which can move the positions of the get and put pointer are:

- seekg()
- seekp()
- tellg()
- tellp()

Let us look at these functions in detail.

13.6.1 seekg()

The **seekg()** function is used to move or set the **get** pointer at a particular location. It is used in file handling to set the position of the next character that is to be extracted from the input stream from a file. The syntax of seekg() is given below:

```
istream&seekg(off_type offset, seekdir origin);
```

Here, *off_type* is an integer type that can contain a large value that offset can have; *seekdir* is the seeking direction. And, o*rigin* is an object of type ios_base::seekdir that can take any of the following three values:

ios::beg	Beginning of file
ios::cur	Current location
ios::end	End of file

13.6.2 seekp()

The syntax of seekp() is given below:

```
ostream&seekp(off_type offset, seekdir origin);
```

Here, *off_type* is an integer type that can contain a large value that offset can have; *seekdir* is the seeking direction. And, o*rigin* can take any of the three values as specified above in the syntax for seekg().

The **seekp()** function moves the associated file's current **put** pointer *offset* number of characters from the specified *origin,* which must be one of the three specified values (beg, cur, or end). The seekp() function allows us to change a specific character in a file.

13.6.3 tellg()

The **tellg()** function is used with input streams, and returns the current position of the **get** pointer in the stream. The syntax of tellg() is shown below:

```
pos_typetellg();
```

It does not have any parameters but returns a value of the member type pos_type, which is an integer data type representing the current position of the get pointer.

13.6.4 tellp()

The **tellp()** function is used with output streams, and returns the current position of the **put** pointer in the stream. The syntax of tellp() is shown below:

```
pos_typetellp();
```

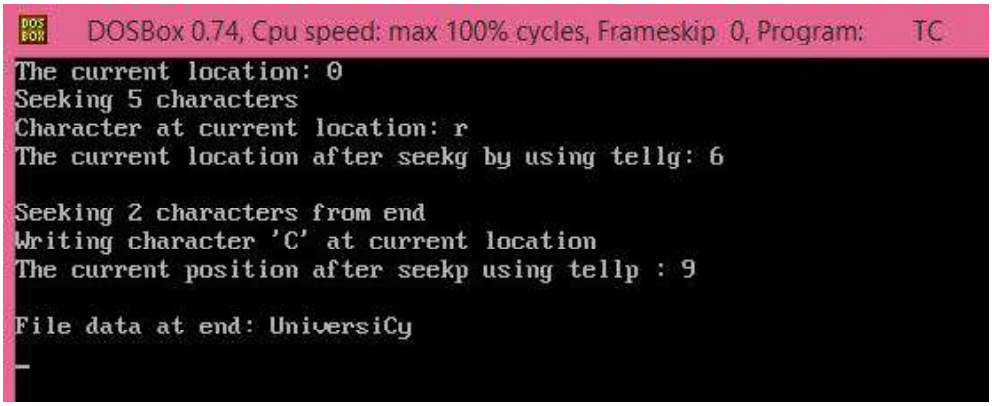
It does not have any parameters but returns a value of the member type pos_type, which is an integer data type representing the current position of the put pointer.

Program 13.4: Program to show use of seekp(), seekg(), tellp() and tellg() functions

```
#include <iostream.h>
#include <fstream.h>
int main()
{
      char put_data[]="University";
      char *read_data;
      char *data;
      fstream file; //open file to put
      file.open("file.txt",ios::app);
      for(int i=0;put_data[i]!='\0';i++)
      {
            file.put(put_data[i]); //put into the file
      }
      file.close();
      //opening file in input and output mode
      file.open("file.txt", ios::in | ios::out);
```

```
        cout<< "The current location: ";
//getting current location
cout<<file.tellg() <<endl;
        cout<< "Seeking 5 characters" <<endl;
        file.seekg(5, ios::beg); //seeking 5 characters
        //extracting one character from current location
        char c = file.get();
cout<< "Character at current location: " << c <<endl;
//printing the character
        cout<< "The current location after seekg by
using tellg: ";
//getting current location
cout<<file.tellg() <<endl<<endl;
        cout<< "Seeking 2 characters from end" <<endl;
        //seeking 2 characters from end
        file.seekp(-2, ios::end);
        cout<< "Writing character 'C' at current
location" <<endl;
       //writing a character 'C' at current location
        file.put('C');
        cout<< "The current position after seekp using
tellp : ";
cout<<file.tellp() <<endl<<endl;
        //printing thr current position in file
        file.seekg(0, ios::beg);
        file >> data;
        cout<< "File data at end: " << data <<endl;
        file.close(); //closing the file
        return 0;
}
```

Output:



In program 13.4, the use of seekg(), seekp(), tellg() and tellp() functions are demonstrated using a file "file.txt" where a word "University" is written. The seekg() pointer is first used to move the get pointer to five places from the beginning. The character at the current position is now read and displayed. So, "r" is displayed on the screen. The tellg() is then used to get the current location of get pointer and to display this value which is "6". The seekp() is used next to move the put pointer to minus two (-2) places from the end which means the put pointer is now between "i" and "t". Then, the character "C" is written into the file "file.txt" at the current position which replaces "t". The tellp() is then used to get the current location of put pointer and to display this value which is "9". The contents of the file are then finally displayed on the screen.



CHECK YOUR PROGRESS

Q 3: Why do we use tellg() function?

Q 4: Why do we use seekg() function?



13.7 LET US SUM UP

The key points to keep in mind in this unit are:

- To create an input stream we must declare the stream to be of class **ifstream**.

- To create an output stream we must declare the stream to be of class **ofstream**.
- To create both an input and output stream together, we must declare the stream to be of class **fstream**.
- Once the stream has been created, we need to associate a file to the stream by using the **open()** function.
- **The open()** function is a member of all three streams.
- The **close()** function is used to close the connection to a file.
- The **get()** function reads a single character from the input stream.
- The **put()** function writes a single character to the output stream.
- The **read()** function is used to read blocks of binary data from a file and the **write()** function is used to write blocks of binary data to a file in C++.
- In C++, whether the end of the file has been reached or not can be detected using the function **eof().**
- The get pointer for a stream indicates the position in the stream from which characters are extracted and the put pointer for a stream indicates the position in the stream where characters are inserted.
- The **seekg()** function allows us to seek an arbitrary position in a file. It is used in file handling to set the position of the next character that is to be extracted from a file.
- The **seekp()** function moves the associated file's current **put** pointer *offset* number of characters from the specified *origin,* which must be one of the three specified values (beg, cur, or end).
- The **tellg()** function is used with input streams and returns the current position of the **get** pointer in the stream.
- The **tellp()** function is used with output streams and returns the current position of the **put** pointer in the stream.



13.8 FURTHER READING

1. Schildt, H. (2004). *C++: The complete reference*. New York: McGraw-Hill.
2. Stroustrup, B.(2000).*The C++ programming language*. Pearson Education India.
3. https://support.sas.com/documentation/onlinedoc/ccompiler/doc700/html/cpp/z0102251.htm
4. https://www.geeksforgeeks.org/



13.9 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q No 1: The three types of streams are: input, output, and input/ output.

Ans to Q No 2: Files contain both formatted text and raw data and can be opened either in text mode or binary mode. But when we open a file in text mode, various character translations like carriage return being converted to newlines might take place. So we can open in binary mode so that such character translations do not occur.

Ans to Q No 3: The tellg() function is used with input streams, and returns the current position of the get pointer in the stream.

Ans to Q No 4: The seekg() function allows us to seek an arbitrary position in a file. It is used in file handling to set the position of the next character that is to be extracted from a file.



13.10 MODEL QUESTIONS

Q1. What are the different file classes in C++?

Q2. Describe the different file modes in C++.

Q3. Describe opening a file in C++ with an example.

Q4. Differentiate between the put() and get() functions.

Q5. What is the importance of eof() function?

Q6. Describe how read() and write() functions can be used to read and write data from a file.

Q7. Illustrate the usage of seekg(), seekp(), tellg() and tellp() functions with the help of examples.

Q8. Describe the file pointers in C++? How are they used?

*** ***** ***

UNIT 14: TEMPLATES

UNIT STRUCTURE

14.1 Learning Objectives

14.2 Introduction

14.3 Templates

14.4 Function Template

14.5 Function Template Overloading

14.6 Class Template

14.7 Default Template Arguments

14.8 Let Us Sum Up

14.9 Further Reading

14.10 Answers to Check Your Progress

14.11 Model Questions

14.1 LEARNING OBJECTIVES

After going through this unit, you will be able to –

- define templates
- use function templates in programs
- describe how function template overloading works
- use class templates in programs
- describe template arguments

14.2 INTRODUCTION

In the previous unit, we have learnt about file classes and file modes. We have also learnt how to open and close a file, write into a file and read data from a file using functions like read(), write(), get() and put(). Manipulation of file pointers and getting their location by using functions like seekp(), seekg(), tellp() and tellg() were also included in the previous unit.

In this unit, we will learn about templates. We will learn in detail about both function templates and class templates. Function template overloading and default template arguments are also covered in this unit.

We will learn about object-oriented design, its approaches, techniques, and different models in detail in the next unit.

14.3 TEMPLATES

Templates are one of the most powerful features of C++. It is possible to create generic functions and classes using templates. In a generic function or class created using templates, the type of data upon which the function or the class operates is specified as a parameter. So by using templates, a general procedure can be applied to different types of data. For example, the logic behind a sorting function or a printing function is the same whether we use integers or strings or floating point numbers. Only the type of data being sorted or printed will be different. By using a template, we define the nature of the algorithm independent of the data. Once defined, the compiler will automatically generate the correct code for the data type that we have used during the execution of the program. So, basically, we are creating a function or class that can work using different types of data.

14.4 FUNCTION TEMPLATE

A function template is created using the keyword “template”. The meaning of the word template in real-world accurately describes its meaning in C++ also. It is used to create a template (or framework) that describes what the function is supposed to do. The compiler will then fill out the missing details using the given parameter values at the compile time. The general syntax of a function template is given below:

```
template <typename T>return_typefunction_name
(T parameter 1, T parameter 2, ...)
     {
          // body of function
     }
```

Here, “***typename***” is the keyword and ‘*T*’ is the template argument that accepts different data type to be used by the function. This template argument will be replaced with the actual data type by the compiler

automatically when it creates a version of the function. In place of the "*typename"* keyword, we can also use the keyword "***class***".

The function clause of the function template definition does not have to be in the same line as the template definition. Another way to represent the syntax for function template is shown below:

```
template <class T>
T function_name (T parameter1, T parameter 2, ...)
{
      // body of function
}
```

Now, let us look at an example to understand the concept of function templates more clearly.

Program 14.1: Program showing function template

```
#include <iostream.h>
#include <conio.h>
template<class T>
T sum(T number_1, T number_2)
{
      return (number_1 + number_2);
}
int main ()
{
      clrscr();
      /*calling sum function template with
integer paramenters */
      int result_1 = sum(1, 3);
      cout<< "Sum of 1 and 3 is: " << result_1
<<endl;
      /*calling sum function template with
double paramenters */
      double result_2 = sum(1.3, 3.5);
```

```
      cout<< "Sum of 1.3 and 3.5 is: " << result_2
<<endl;


      getch();
      return 0;
}
```

Output:



In Program 14.1, we have defined a template where we have a sum() function. The sum()function takes any two numerical values and displays their summation result. In the main() function, the sum() function is called twice. For the first time, we have passed integer values and for the second time floating point double values are passed both using the same function sum(). This is possible because we have used a template where the compiler fills the missing data type at compile time.

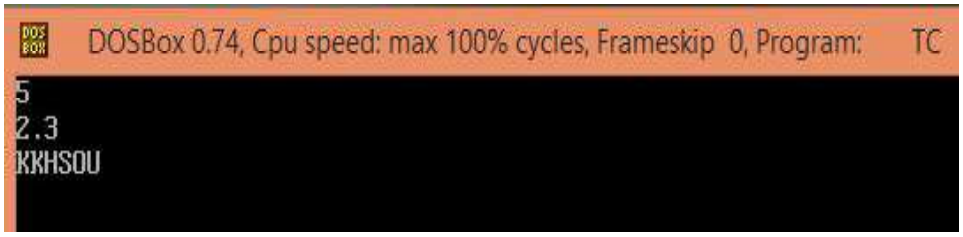
We can use more than one data type in the template statement by using a comma-separated list. For example, program 14.2 creates a template function that can have three data types.

Program 14.2: Program showing use of multiple data types in a function template

```
#include <iostream.h>
#include <conio.h>
template <class T1, class T2, class T3>
void mprint( T1 a, T2 b, T3 c)
{
      cout<< a <<endl;
      cout<< b <<endl;
      cout<< c <<endl;
```

```
}
int main ()
{
      clrscr();
//calling mprint function using multiple data types
      mprint(5, 2.3, "KKHSOU");
      getch();
      return 0;
}
```

Output:



14.5 COMPONENTS OF A FUNCTION

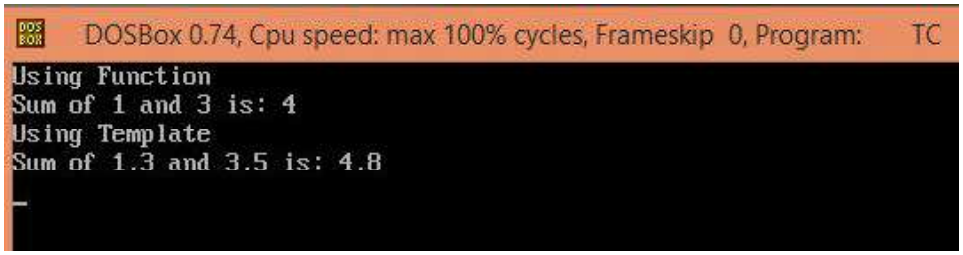
As we have seen in the programs above, a generic function overloads itself as needed. But we can explicitly also overload it. This is called explicit specialization. When we overload a generic function, that overloaded function overrides the generic function relative to that specific version.

Program 14.3: Program showing explicit overloading of a template function

```
#include <iostream.h>
#include <conio.h>
template <class T>
T sum(T number_1, T number_2){
      cout<< "Using Template" <<endl;
      return (number_1 + number_2);
}
int sum(int number_1, int number_2) {
      cout<< "Using Function" <<endl;
      return (number_1 + number_2);
```

```
}
int main ()
{
clrscr();
/*calling sum function template with integer
Paramenters */
int result_1 = sum(1, 3);
cout<< "Sum of 1 and 3 is: " << result_1 <<endl;
/*calling sum function template with
double paramenters  */
double result_2 = sum(1.3, 3.5);
cout<< "Sum of 1.3 and 3.5 is: " << result_2
<<endl;
getch();
return 0;
}
```

Output:



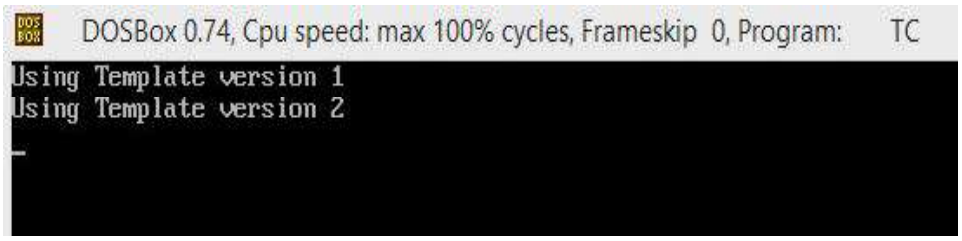
We can observe from the output of program 14.3, when sum(1, 3) is called, it invokes the explicitly overloaded version of sum() function defined in the program. So, the compiler does not generate this version of the generic sum() function. However, when sum(1.3, 3.5) is called, then the generic sum() function defined in the template is called.

Just like we can create explicit overloaded generic function, we can also overload the template specification itself. We can do this by creating another version of the template that differs from any others in its parameter list. An example for overloading a function template that accepts one or more parameters is shown in program 14.4.

Program 14.4: Program showing overloading of a function template

```
#include <iostream.h>
#include <conio.h>
// first version of template
template <class T>
void mprint(T number_1){
      cout<< "Using Template version 1" <<endl;
}
// second version of template
template <class T, class S>
void mprint(T number_1, S number_2){
      cout<< "Using Template version 2" <<endl;
}
int main()
{
      clrscr();
      //calls mprint(T)
      mprint(10);
      //calls mprint(T,S)
      mprint(10,"KKH");
      getch();
      return 0;
}
```

Output:



In the above program, the template for mprint() function is overloaded to accept either one or two parameters.



CHECK YOUR PROGRESS

Q 1: What is a template?

Q 2: Give the syntax for declaring a function template.

14.6 CLASS TEMPLATE

As we have seen in the earlier sections, just like we have created generic functions we can also create generic class. Generic classes are used when the algorithm used in the class can be generalized. For example, the same algorithm used in a queue of integers will also work for a queue of floating point numbers. When we create a generic class, it can perform the operations defined in that class for any type of data. The compiler will then automatically generate the type of object depending upon the type we specify when the object is created.

A class template is created using the keywords “template” and “class”. The general syntax of a class template is given below:

```
template <class T> class class_name {
            // body of class
      };
```

Here, ‘*T*’ is the type name that will be specified when a class is instantiated. Another way to represent the syntax for class template is shown below:

```
template <class T>
class class_name {
            // body of class
      };
```

We can create a specific instance of the class by using the following general form:

```
class_name<type>ob;
```

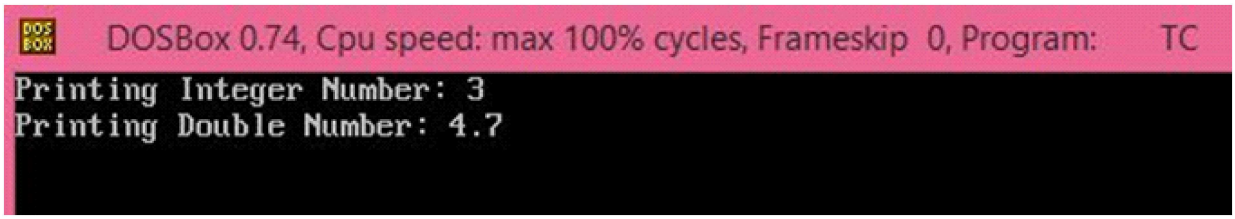
Here, ‘type’ is the type name of the data that the class will be operating upon. The member functions of a class are automatically generic and we do not need to use template to explicitly specify these member functions.

Program 14.5: Program showing class template

```
#include <iostream.h>
#include <conio.h>
template <class T>
class PrintNumber {
     private:
          T number;
     public:
          PrintNumber(T num) {
               number = num;
          }

          T GetNumber() {
               return number;
          }
};
int main () {
     clrscr();
          //creating object with double datatype
     PrintNumber<int>IntNumber(3);
          //printing number with integer datatype
     cout<< "Printing Integer Number: ";
cout<<IntNumber.GetNumber() <<endl;
          //creating object with double datatype
     PrintNumber<double>DoubleNumber(4.7);
          //printing number with double datatype
     cout<< "Printing Double Number: ";
cout<<DoubleNumber.GetNumber() <<endl;
     getch();
     return 0;
}
```

Output:



From program 14.5, we can observe that the declaration of a generic class is similar to that of a generic function. It is not until an object is created of the class “PrintNumber” is declared that the actual data type is determined. When a specific instance of “PrintNumber” is declared, the compiler automatically generates all the functions and variables associated with that class. In this example, two different types of objects are declared of type “int” and “double”.

14.7 DEFAULT TEMPLATE ARGUMENTS

Default arguments can be associated with a template class. For example, in the code below an”int” type is associated as a default argument with a template class.

```
template <class T = int> class class_name {
            // body of class
      };
```

Here, ‘*T’* is the type name that will be specified when a class is instantiated and the default value for ‘T’ is ‘int’ when no value is specified. The default value is used when no explicit value is specified when the class is instantiated.



CHECK YOUR PROGRESS

Q 3: How are class templates created?

Q 4: How can we create a specific instance of a class object?



14.8 LET US SUM UP

The keypoints to keep in mind in this unit are:

- Templates are one of the most powerful features of C++.
- It is possible to create generic functions and classes using templates.
- In a generic function or class created using templates, the type of data upon which the function or the class operates is specified as a parameter.
- By using a template, we define the nature of the algorithm independent of the data. Once defined, the compiler will automatically generate the correct code at the compile time for the data type that we have used.
- A function template is created using the keyword “template” and “typeame” or “class”.
- When we overload a generic function, that overloaded function overrides the generic function relative to that specific version.
- Just like we can create explicit overloaded generic function, we can also overload the template specification itself. We can do this by creating another version of the template that differs from any others in its parameter list.
- Generic classes are used when the algorithm used in the class can be generalized.
- When we create a generic class, it can perform the operations defined in that class for any type of data. The compiler will then automatically generate the type of object depending upon the type we specify when the object is created.
- A class template is created using the keywords “template” and “class”.
- Default arguments can be associated with a template class.



14.9 FURTHER READING

1. Schildt, H. (2003). C++: *The complete reference*. New York: McGraw-Hill.
2. Stroustrup, B.(2000). *The C++ programming language*. Pearson Education India.



14.10 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q No 1: Templates are one of the most powerful features of C++. By using a template, we can define the nature of the algorithm independent of the data. Once defined, the compiler will automatically generate the correct code at the compile time for the data type that we have used.

Ans to Q No 2: The general syntax of a function template is given below:

```
template <class T>
return_typefunction_name   (T parameter 1,     T
parameter 2, ...)
      {
            // body of function
      }
```

Here, "***class"*** is the keyword and '*T*' is the template argument that accepts different data type to be used by the function. This template argument will be replaced with the actual data type by the compiler automatically when it creates a version of the function.

Ans to Q No 3: A class template is created using the keywords "template" and "class". The general syntax of a class template is given below:

```
template <class T> class class_name {
            // body of class
      };
```

Where '*T*' is the type name that will be specified when a class is instantiated.

Ans to Q No 4: We can create a specific instance of the class by using the following general form:

```
class_name<type>ob;
```

Where 'type' is the type name of the data that the class "class_name" will be operating upon and "ob" is the object of the class.



14.11 MODEL QUESTIONS

Q1. Define template.

Q2. What are function templates and class templates?

Q3. How are function templates declared? Explain using an example.

Q4. How are class templates declared? Explain using an example.

Q5. Show using a program how function template overloading is done.

Q6. Define a function template that swaps two numeric data types.

Q7. Write a sorting function using generic type so that the function can sort any array of numeric type (int, double, float and long).

Q8. Write a linear search function using generic type so that the function can search a specific element in any array of numeric type (int, double, float and long).

*** ***** ***

UNIT 15: OBJECT-ORIENTED DESIGN

UNIT STRUCTURE

15.1 Learning Objectives

15.2 Introduction

15.3 Object-Oriented Design Approaches

- 15.3.1 Elements of Object-Oriented System
- 15.3.2 Features of Object-Oriented System

15.4 Object Modeling Techniques (OMT) Tools

15.5 Class Diagram

- 15.5.1 Components of a Class Diagram
- 15.5.2 Relationships

15.6 Object Diagram

- 15.6.1 Components of an Object diagram
- 15.6.2 Difference between Class Diagram and Object Diagram

15.7 State Diagram

- 15.7.1 Components of a State Diagram

15.8 Data Flow Diagram (DFD)

- 15.8.1 Components of DFD
- 15.8.2 DFD Levels
- 15.8.3 Advantages and Disadvantages of DFD

15.9 Phases of Object-Oriented Development

- 15.9.1 System Analysis
- 15.9.2 System Design
- 15.9.3 Object Design
- 15.9.4 Implementation

15.10 Advantages of Object-Oriented Methodology

15.11 Let Us Sum Up

15.12 Further Reading

15.13 Answers to Check Your Progress

15.14 Model Questions

15.1 LEARNING OBJECTIVES

After going through this unit, you will be able to –

- learn about Object-Oriented Design Approaches
- describe Object Modeling Techniques (OMT) tools
- learn about Class Diagram, Object Diagram, and State Diagram
- describe DFD
- learn about different phases of Object-Oriented Development

15.2 INTRODUCTION

In the previous unit we have discussed about templates in C++. This unit discusses the concept of Object-Oriented Design.

Object-Oriented Design (OOD) is the process of using object-oriented methodology to design a computing system or application. This technique enables the implementation of a software solution based on the concepts of objects.OOD serves as apart of the Object-Oriented Programming (OOP) process or lifecycle.

In object-oriented system design and development, OOD helps in designing the system architecture or layout - usually after the completion of the object-oriented analysis (OOA). The designed system is then created or programmed using object-oriented based techniques and/or an object-oriented programming language (OOPL).

15.3 OBJECT-ORIENTED DESIGN APPROACHES

In the object-oriented approach, the focus is on capturing the structure and behavior of information systems into smaller modules that combine both data and processes. The main aim of Object-Oriented Design (OOD) is to improve the quality and productivity of system analysis and design by making it more usable.

15.3.1 Elements of Object-Oriented System

- **Objects** " An object is something that exists within the problem domain and can be identified by data (attribute) or behavior. All

tangible entities (student, patient, teacher) and some intangible entities (bank account, passport) are modeled as objects.

- **Attributes** " Attributes describe information about the object.
- **Behavior** " Behavior stateswhat the object can do. It defines the operation performed on objects.
- **Class** " A class encapsulates data and its behavior. Objects with similar meaning and purpose are grouped together as a class.
- **Methods** " Methods decide the behavior of a class. They are justactions that an object can perform.
- **Message** " A message is a function or procedure call from one object to another. They are information sent to objects to trigger methods. Practically, a message is a function or a procedure call from one object to another.

15.3.2 Features of Object-Oriented System

An object-oriented system has several features. Some of the important features are described below:

- **Encapsulation**

  Encapsulation is the process of hiding information. It is the combination of process and data into a single entity. Data of an object is hidden from the rest of the system and available only through the services of the class. It allows improvement or modification of methods used by objects without affecting other parts of a system.

- **Abstraction**

  Abstraction is the process of taking or selecting necessary method and attributes to specify the object. It focuses on the essential characteristics of an object relative to the perspective of a user.

- **Relationships**

  All the classes in the system are related to each other. Objects do not exist in isolation;instead they exist in relationships with other objects.

There are three types of object relationships –

1. **Aggregation** " It indicates the relationship between a whole and its parts.
2. **Association** " In this, two classes are related or connected in some way such as one class works with another to perform a task or one class acts upon another class.
3. **Generalization** " The child class is based on the parent class. It shows that two classes are similar but have some differences.

- **Inheritance**

  Inheritance is a feature that allows creation of sub-classes from an existing class by inheriting the attributes and/or operations of existing classes.

- **Polymorphism**

  Polymorphism is the ability to takeonseveral differentforms. It is applicable to both objects and operations. A polymorphic object is one whose true type hides within a super or parent class.

  In polymorphic operations, the operation may be carried out differently by different classes of objects. It allows us to manipulate objects of different classes by knowing only their common properties.

15.4 OBJECT MODELING TECHNIQUES (OMT) TOOLS

Object Modeling Technique (OMT) is a real-world based modeling approach for software modeling and designing. It was originallydeveloped as a method of developing object-oriented systems and to support object-oriented programming. It representsthe static structure of the system. The object modeling technique is a methodology of object-oriented analysis, design, and implementation. It focuses on creating a model of objects from the real world and then using this model to develop object-oriented software. The Object Modeling Technique (OMT) was developed by James Rambaugh.

The purposes of the Object Modeling Technique are:

- To test physical entities before construction of them.
- To make communication easier with the users.
- To reduce the complexity of the software.
- To present information in an alternative way (visualization).
- To solve real-world problems.

There are three main types of models that are proposed by OMT:

1. **Object Model:**
   Object Model encompasses the principles of abstraction, encapsulation, modularity, hierarchy, typing, concurrency, and persistence. The Object Model basically emphasizes the objects and classes. The main concepts related to Object Model are classes and their association with attributes. The predefined relationships in the object model are aggregation and generalization (multiple inheritance).

2. **Dynamic Model:**
   Dynamic Model deals with states, events, and state diagram (transition diagram) on the model. The main concepts related to Dynamic Model are states, the transition between states, and events to trigger the transitions. The predefined relationships in the object model are aggregation and generalization.

3. **Functional Model:**
   The functional model focuses on how data is flowing, where data is stored, and different processes. The main concepts involved in the Functional Model are data, data flow, data store, process, and actors. The Functional Model in OMT describes the whole processes and actions with the help of a data flow diagram (DFD).

15.5 CLASS DIAGRAM

The class diagram depicts a static view of an application. It shows the types of objects residing in the system and the relationships between them.

A class consists of its objects, and it may also inherit from other classes. A class diagram is used to visualize, describe, and document different aspects of the system, and also to construct executable software code.

It shows the attributes, classes, functions, and relationships that gives an overview of the software system. It constitutes of class names, attributes, and functions in a separate compartment that helps in software development. As it is a collection of classes, interfaces, associations, collaborations, and constraints, it is termed as a structural diagram.

The main purpose of class diagrams is to createa static view of an application. It is the only diagram that is generally used for construction, and it can be mapped with object-oriented languages.

Following are the purpose of class diagrams:

- It analyses and designs a static view of an application.
- It describes the primary responsibilities of a system.
- It is a base for the component and deployment diagrams.
- It incorporates forward and reverse engineering.

15.5.1 Components of a Class Diagram

The class diagram is made up of three sections:

1. **Upper Section:** The upper section encloses the name of the class. A class is a depictionof similar objects that shares the same relationships, attributes, operations, and semantics.

   The following rules should be taken into account while representing a class:

   - The initial letter of the class name should be capitalized.
   - Class names should be written in bold format.
   - The class name should be placed at the center of the upper section.
   - Names of abstract classes should be written in italics format.

2. **Middle Section:** The middle section comprisesofthe attributes, which describe the quality of the class. Attributes have the following characteristics:
   - Anattribute is written along with its visibility factors, which are public (+), private (-), protected (#), and package (~).
   - The accessibility of an attribute class is represented by the visibility factors.
   - Attributes should be assigned a meaningful name that will explain its usage inside the class.

3. **Lower Section:** The lower section contains methods or operations. The methods are depicted in the form of a list, where each method is written in a single line. It showshow a class interacts with data.

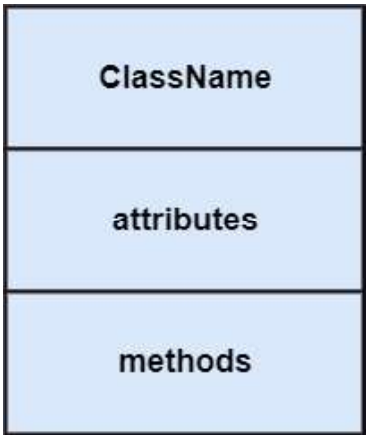


Figure 15.1:Components of a Class Diagram

15.5.2 Relationships

There are three types of relationships:

1. **Dependency:** A dependency is a semantic relationship that existsbetween two or more classes where a change in one class cause changes in the other class. It forms a weaker kind ofrelationship.

   In the following example, Resultis dependent on Student.

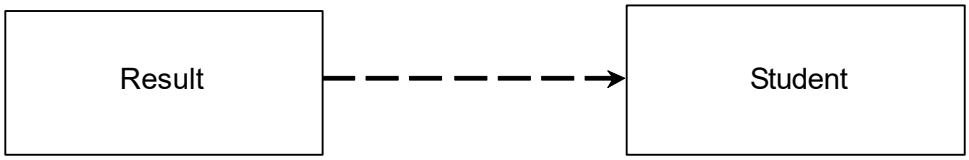


Figure 15.2:Example of Dependency

2. **Generalization:** A generalization is a relationship between a parent class (called assuperclass) and a child class (called assubclass). Here, the child class is inherited from the parent class.

   For example, The Current Account, Saving Account, and Credit Account are the generalized form of Bank Account.

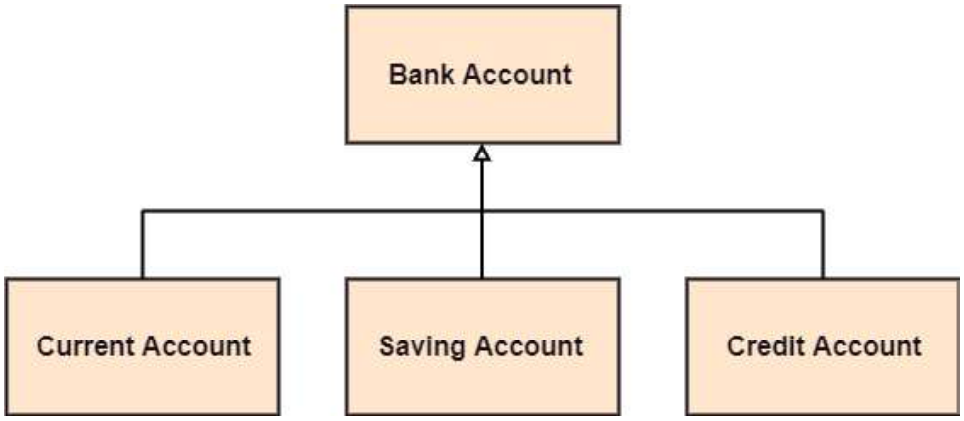


Figure 15.3:Example of Generalization

3. **Association:**Association describes a physical or static connection between two or more objects. It portrays how many objects are there in the relationship.

   For example, a department is associated with the college.



Figure 15.4:Example of Association

Some other kinds of relationships are:

- **Multiplicity:**Multiplicitydefines a specific range of instances of attributesthat are allowed. In the case wherea range is not specified, one is considered as a default multiplicity.

  For example, multiple patients are admitted to one hospital.

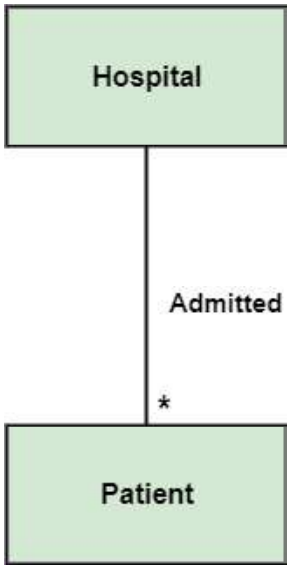


Figure 15.5:Example of Multiplicity

- **Aggregation:** An aggregation is a subset of association that represents “has a” relationship. It is more specific than the association. It represents a “part-whole” or “part-of” relationship. In this kind of relationship, the child class can exist independent of its parent class.

  For example, a company comprisesof a number of employees, and even if one employee resigns, the company still exists.



Figure 15.6:Example of Aggregation

- **Composition:** The composition is a subset of aggregation. It showsthe dependency between the parent and its child, which means if one part is deleted, then the other part also gets scrapped. It represents a “whole-part” relationship.

  For example, a contact book consists of severalcontacts, and if you delete the contact book, all the contacts will be lost.



Figure 15.7:Example of Composition

Some key points that are to keep in mind while drawing a class diagram are as follows:

- In order to describe a complete aspect of the system, the class diagram should be given a meaningful name.
- The objects and their relationships should be defined in advance.
- The attributes and methods of each class must be known.
- Since a large number of the unwanted properties will lead to a complex diagram, just a minimum number of desired properties should be specified.
- The developer can use notes as and when required to describe the aspects of a diagram.
- The diagrams should be redrawn and reworked asmany times as necessaryto make themoptimal before producing its final version.

For example, a class diagram describing the sales order system is given below.

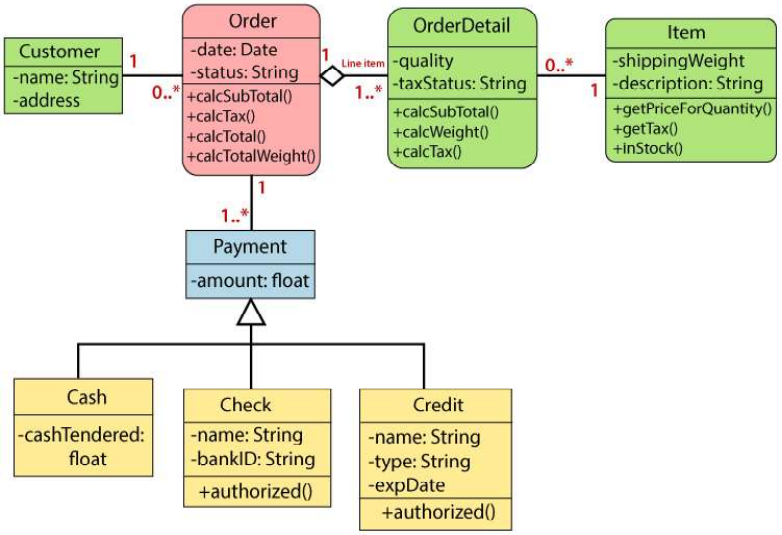


Figure 15.8:Class Diagram Describing the Sales Order System

15.6 OBJECT DIAGRAM

An object diagram represents an instance of a class diagram.They are dependent on the class diagram as they are derived from the class diagram. Objects help in depicting a static view of an object-oriented system at a specific instant.

The notation of an objectis presented as follows:

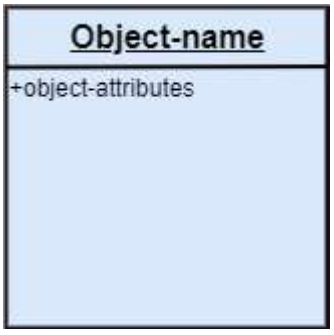


Figure 15.9:Notation of Object

The object diagram has the same purpose as that of a class diagram. While a class diagram provides an abstract view which comprises of classes and their relationships, an object diagram represents an instance of the classat a particular point of time.

The object diagram is actually similar to the actual system behavior. The main purpose is to portray a static view of a system.

15.6.1 Components of an Object diagram

Object diagrams are quitesimple to create.Theyaremade from objects, which are represented by rectangles that arelinked together with lines.

Below are the major elements of an object diagram.

- **Objects**

  Objects are instances of a class. For example, if “car” is a class, then “Maruti Alto”is an object of a class.

- **Class titles**

  Class titles are the specific attributes of a given class.

  For example,in a family tree object diagram, class titles include the name, gender, and age of the family members. You can list class titles as items on the object or even in the properties of the object itself (such as color, size, etc.).

- **Class attributes**

  Class attributes are represented by a rectangle with two tabs that indicates a software element.

- **Links**

  Links are the lines that connect two shapes of an object diagram to each other.

  For example, the corporate object diagram below shows how departments are connected in the traditional organizational chart style.

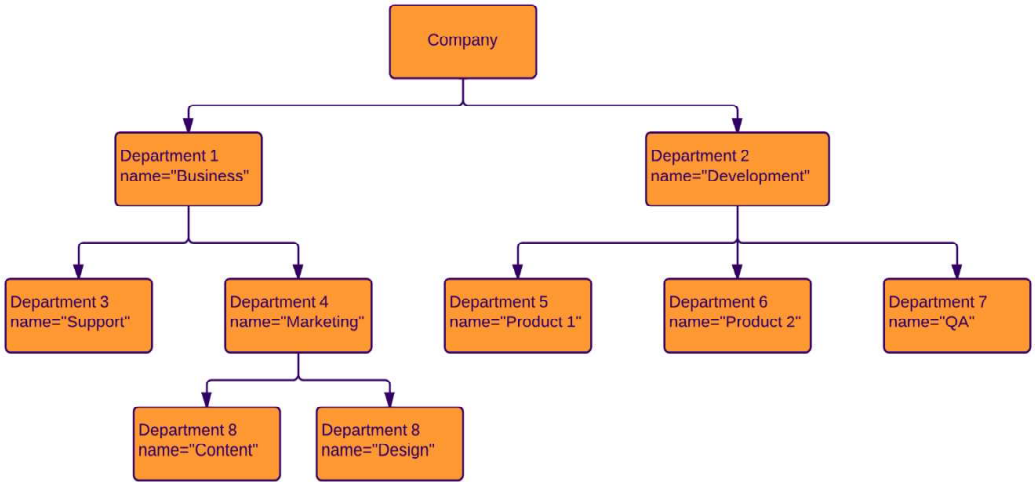


Figure 15.10: Object Diagram of Corporate

While drawing an Object diagram,the following rules should be followed.

1. Before drawing the object diagram, all the objects present in the system should be examined.
2. The relations between the objects must be established beforehand.
3. The association relationship among the entities must be clearly defined.
4. A meaningful name should be assigned to depict the functionality of an object.
5. The objects should be examined to understand their functionality.

15.6.2 Difference between Class Diagram and Object Diagram

To some extent, both object and class diagrams are similar. However there are some characteristic differences as stated below:

Class Diagram	Object Diagram
It shows the static view of a system.	It depicts the real-time behavior of a system.
Dynamic changes are not captured in the class diagram.	Dynamic changes are included in the object diagram.
Here the data values and attributes of an instance are not involved.	It involves data values and attributes of an entity.
In a class diagram, the object behavior is manipulated.	An object is an instance of a class.

Table 15.1:Difference between Class Diagram and Object Diagram

15.7 STATE DIAGRAM

A state diagram is used to portray the condition of the system or part of the system at finite instances of time. It is a behavioral diagram and it depicts the behavior using finite state transitions. State diagrams are also called State machines and State-chart Diagrams.

The basic purpose of a state diagram is to represent various changes in state of the class and not the processes or commands that cause the changes

15.7.1 Components of a State Diagram

1. **Initial state –** We use a black filled circle to represent the initial state of a System or a class.



Figure 15.11:Initial State

2. **State –** A rounded rectangle is used to represent a state. A state shows the conditions or circumstances of an object of a class at a given instant of time.



Figure 15.12:State

3. **Transition –** We use a solid arrow to depict the transition or change of control from one state to another. The arrow is labeled with the event that causes the change in state.

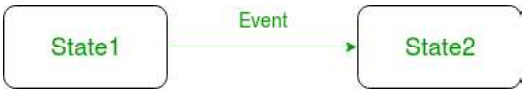


Figure 15.13:Transition

4. **Fork –** A rounded solid rectangular bar is used to represent a Fork notation with an incoming arrow from the parent state and outgoing arrows towards the newly created states. The fork notation is used to represent a state splitting into two or more concurrent states.

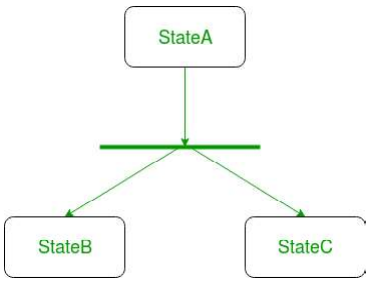


Figure 15.14: Fork Notation

5. **Join –** A rounded solid rectangular bar is used to represent a Join notation with incoming arrows from the joining states and outgoing arrow towards the common goal state. The join notation is used when two or more states concurrently converge into one on the occurrence of an event or events.

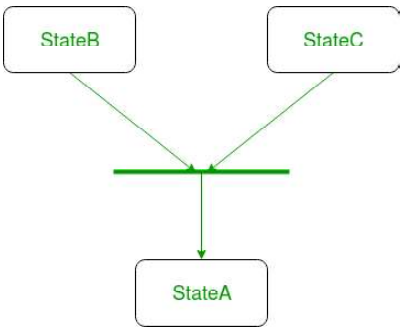


Figure 15.15:Join Notation

6. **Self transition –** Asolid arrow pointing back to the state itself is used to represent a self transition. There might be a scenario when the state of the object does not change upon the occurrence of an event. Self transitions are used to represent such cases.

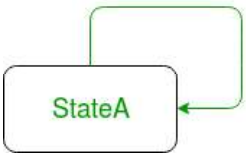


Figure 15.16:Self Transition

7. **Composite state –** A rounded rectangle represent a composite state as well. A state with internal activities is depicted using a composite state.

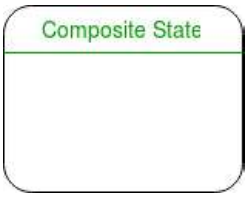


Figure 15.17:Composite State

8. **Final state –** A filled circle within a circle is used to represent the final state in a state machine diagram.



Figure 15.18:Final State

For Example – The state diagram of an online orderis presented as follows.

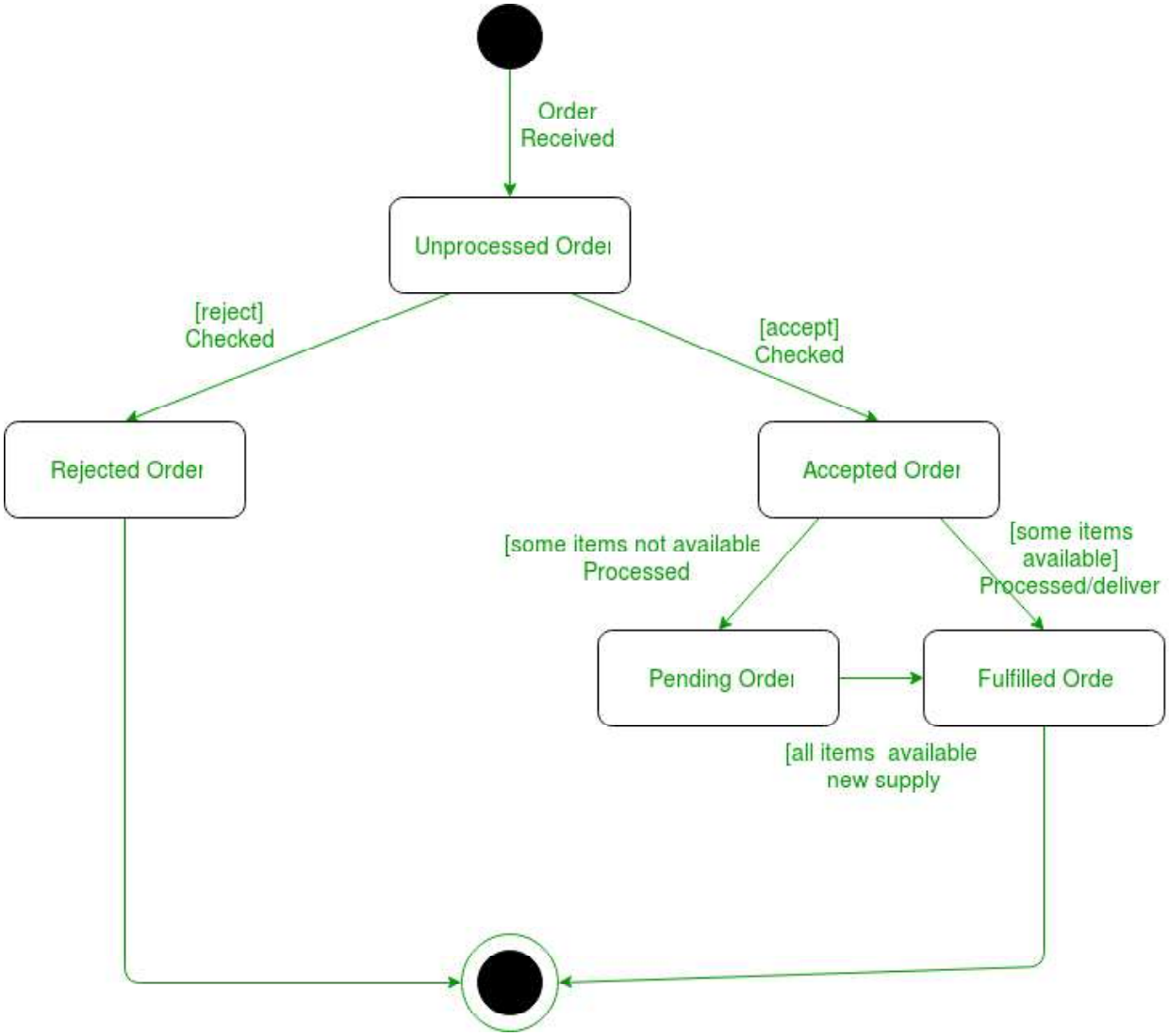


Figure 15.19:State Diagram of an Online Order

Here is an example of how an online ordering system might look like:

- On the event of an order received, we transit from our initial state to Unprocessed order state.
- The unprocessed order is checked.
- If the order is rejected, we go to the Rejected Order state.
- If the order is accepted and we have the items available, we go to the fulfilled order state.
- However if the items are not available, we go to the Pending Order state.

- After the order is fulfilled, we go to the final state. (In this example, Fulfilled order and Rejected order are merged into one final state).



CHECK YOUR PROGRESS

Q 1: In the object-oriented approach, the focus is on capturing the structure and behavior of information systems into small modules that combines both _____and _____.

Q 2: _____ specifies what the object can do.

Q 3: Object Modeling Technique was developed by _____.Q4. _____is the ability to take on many different forms.

Q 5: The class diagram depicts a _____ view of an application. (static/dynamic)

15.8 DATA FLOW DIAGRAM (DFD)

A data flow diagram (DFD) is a diagram that maps out the flow of information for any process or system.It uses symbols like rectangles, circles and arrows, plus short text labels, to show data inputs, outputs, storage points and the routes between the destinations.

A DFD also gives insight into the inputs and outputs of each entity, as well as the process itself.A DFD does not have any control flow and no loops or decision rules are presentin it.

15.8.1 Components of DFD

- **External entity:** It is an outside system that sends or receives data, communicating with the system that is being diagrammed. They are the sources and destinations of information that enter or leave the system. Examples are an outside organization or person, a computer system or a business system. They are also known as terminators, sources, sinks or actors. They are generally drawn on the edges of the diagram.

- **Process:** It is any process that changes the data, and produces an output. It might perform some computation, or sort data based on some logic, or direct the data flow based on certain business rules. The process is described by a short label e,g. “Submit payment.”
- **Data store:** Data stores are files or repositories which hold information for later use. Examples are a database table or a membership form. Every data store has a simple label, such as “Orders.”
- **Data flow:** It is the route that data takes between external entities, processes and data stores. They portray the interface between the other components and are shown with arrows, typically labeled with a short data name, like “Billing details.”

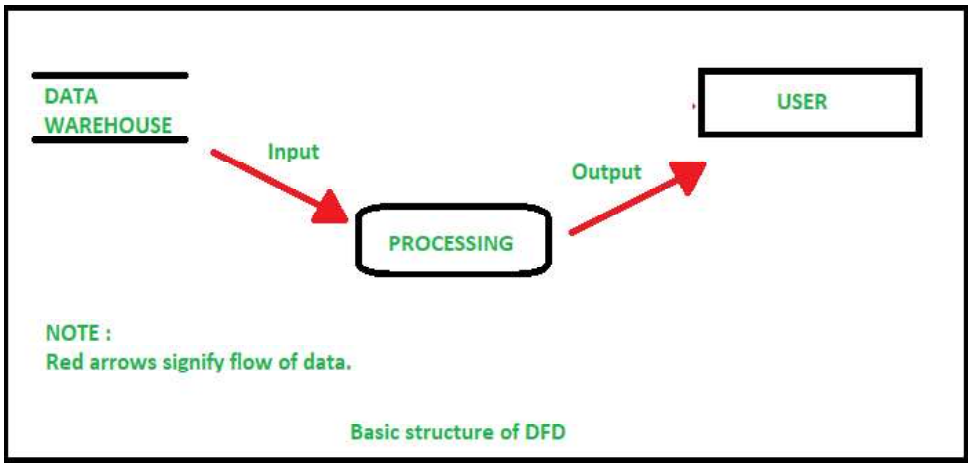


Figure 15.20:Basic Structure of DFD

15.8.2 DFD Levels

A data flow diagram can go into progressively more detail by using levels and layers, zeroing in on a particular piece.DFD levels are numbered as 0, 1 or 2, and occasionally go to even Level 3 or beyond.The level of detail depends on the scope of what you are trying to accomplish by the DFD.

- **DFD Level 0:** Level 0 DFD is also known as **Context Diagram**.It is a basic overview of the whole system or process that is being analyzed or modeled. It is designed to be an "at-a-glance" view that shows the system as a single high-level process with its relationships to various external entities. It should be easily understood by a wide audience, which may include stakeholders, business analysts, data analysts and developers.

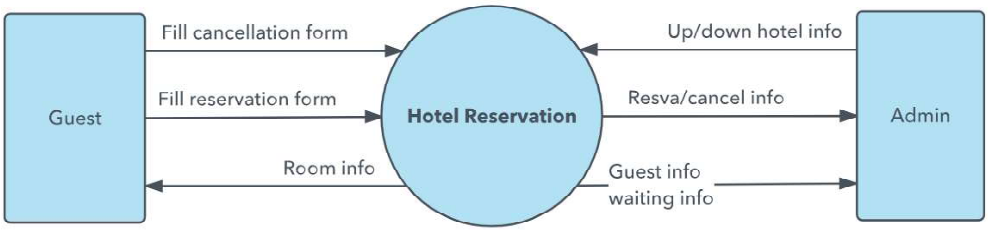


Figure 15.21:Example of Context Diagram

- **DFD Level 1:** Level 1 DFD provides a more detailed breakout of pieces of the Context Level Diagram.Here you will highlight the main functions carried out by the system, while breaking down the high-level process of the Context Diagram into its sub-processes.

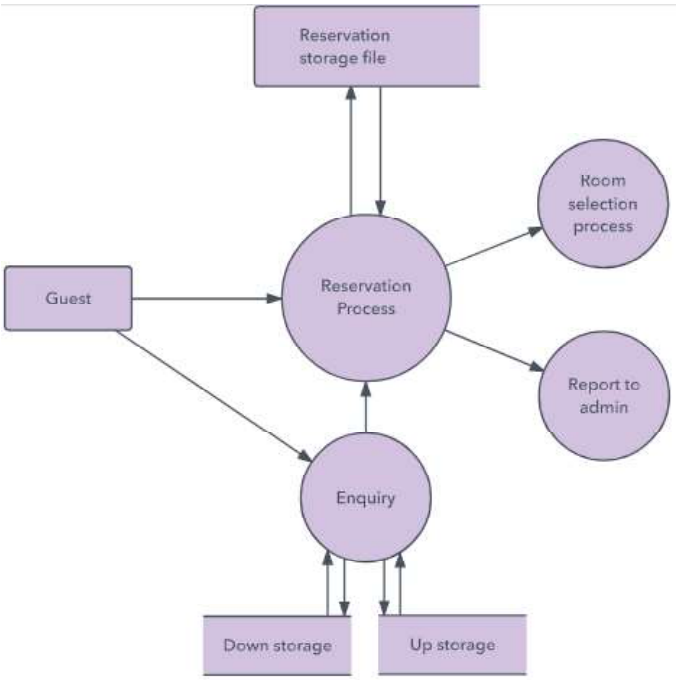


Figure 15.22: Example of DFD Level1

- **DFD Level 2:** Level 2 DFD goes one step deeper into the parts of Level 1. More text might be needed to reach the necessary level of detail about the system's functioning.

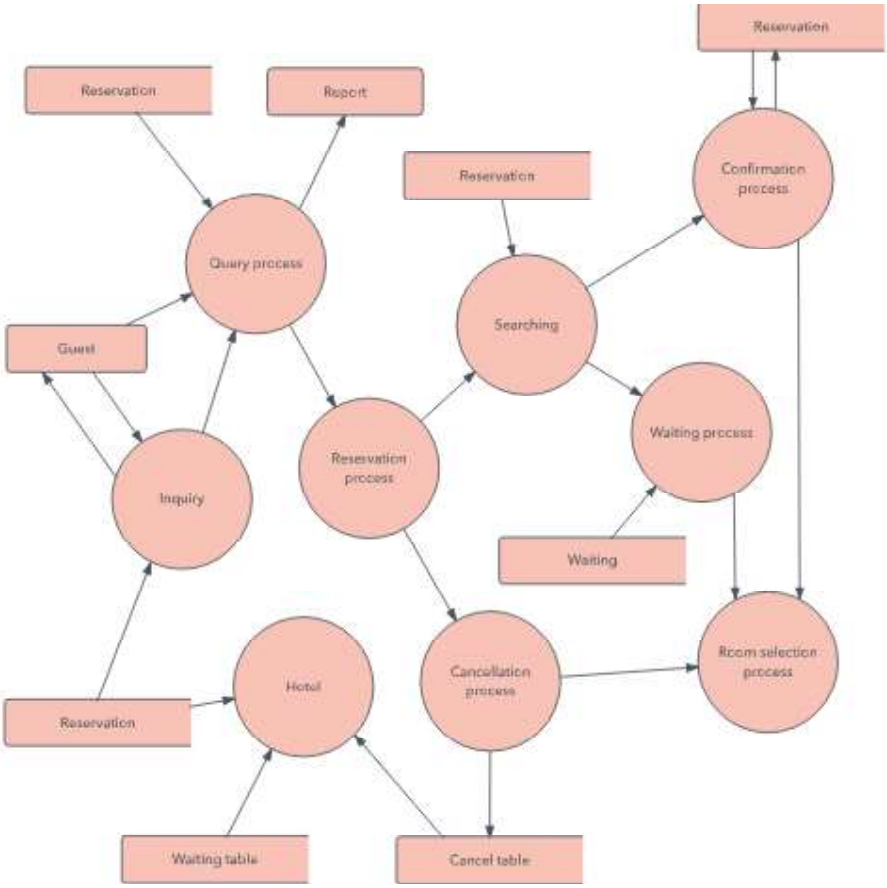


Figure 15.23:Example of DFD Level 2

- Levels 3, 4 and beyond is possible, but going beyond Level 3 is uncommon.Doing so can create complexities that makes it difficult to communicate, compare or model effectively.

15.8.3 Advantages and Disadvantages of DFD

Advantages of DFD

- DFD helps us to understand the functioning and the limits of a system.
- It is a graphical representation that is very easy to understand as it helps to visualize the contents.

- DFD represents a detailed and well explained diagram of the system components.
- DFD is used as the part of system documentation file.
- A DFD can be understood by both technical and nontechnical people.

Disadvantages of DFD

- At times, DFD may confuse programmers while working on the system.
- They take long time to be generated, and many times due to this reason analysts are denied permission to work on it.

15.9 PHASES OF OBJECT-ORIENTED DEVELOPMENT

The Object Oriented Model (OOM) uses objects as the basis for building systems.For this, the system to be developed is first observed and analyzed and the requirements are defined.Once this is done, the objects in the system are identified. For example, in case of a Banking System, a customer, a ledger, a passbook, and even an account is an object.

The Object Oriented Model is somewhat similar to the traditional approach of system designing, in that it also follows a sequential process of system designing but with a different approach.

The basic steps of system designing using Object Oriented Model can be listed as:

- System Analysis
- System Design
- Object Design
- Implementation

15.9.1 System Analysis

As in any other system development models, system analysis is the first phase of OOM as well.In this phase, the developer interacts

with the user of the system to find out the user's requirements and then analyses the system to understand its functioning.

Based on this system study, the analyst then prepares a model of the desired system. This model is purely based on what the activity the system needs to perform. At this stage, the implementation details are not taken into account. Just the model of the system is preparedbased on the idea that the system is made up of a set of interacting objects.The important elements of the system are emphasized here.

15.9.2 System Design

System Design is the next development stage in Object Oriented Methodology. Here the overall architecture of the desired system is decided. The system is organized into a set of sub-systems interacting with each other.While designing the system as a set of interacting subsystems, the analyst takes care of specifications as observed in system analysis as well as what is required from the new system by the end user.

The system analysis perceives the system as a set of interacting objects.A bigger system may also be seen as a set of interacting smaller sub-systems that in turn are composed of another set of interacting objects.While designing the system in this stage, the stress is on the objects comprising the system and not on the processes being carried out in the system.

15.9.3 Object Design

In this stage, the details of the system analysis and system design are implemented. The objects that were identified in the system design phase are designed. Here the implementation of these objects is decided in the form of data structures that are required and the interrelationships between the objects.

For example, we can define a data type called customer and then create several objects of this data type. This concept is known as creating a class.

In this stage of the development process, the designer also decides the classes in the system based on these concepts. He decides on whether the classes need to be created from scratch or any existing classes can be used or new classes can be inherited from them.

15.9.4 Implementation

During this stage, the class objects and the interrelationships of these classes are translated and actually coded by using an object-oriented programming language. The necessary databases are created and the complete system is transformed into an operational one.

15.10 ADVANTAGES OF OBJECT-ORIENTED METHODOLOGY

As compared to the conventional system development techniques, the Object Oriented Methodology provides many benefits:

- Systems that are designed using OOM are closer to the real world as the real world functioning of the system is directly mapped into the system designed using OOM. Due to this, it becomes easier to produce and understand designs.

- The objects in the system are immune to changes in requirement because of data hiding and encapsulation features of object orientation. Here, encapsulation means a technique that allows the programmer to hide the internal functioning of the objects from the users of the objects. Encapsulation separates the internal functioning of the object from the external functioning thereby providing the user flexibility to change the external behavior of the object which makes the programmer code safe against the changes made by the user.

- Object oriented designs encourage reusability. The classes that once defined can be easily used by other applications. This is achieved by defining classes and putting them into a library of classes and maintained for future use. Whenever a new class is needed the programmer first looks into the library of classes and if it is available, it can be used as it is or with some modifications. This reduces the development cost and time. Itincreases quality as well.

- Another way of providing reusability is through the inheritance feature of the object-orientation. The concept of inheritance allows programmers to use existing classes in new applications by making small additions to the existing classes to create new classes quickly.

- As the programmer has to spend less time and effort so he can utilize saved time in concentrating on other aspects of the system.



CHECK YOUR PROGRESS

Q 6: DFD Level 1 is called a Context Diagram. (True/False)

Q 7: In a class diagram, the lower section contains methods or operations. (True/False)

Q 8: A _____ is a semantic relationship between two or more classes where a change in one class cause changes in another class.

Q 9: What is the first phase of OOM?

Q10: The systems designed using OOM are closer to the real world. (True/False)



15.11 LET US SUM UP

- Object-Oriented Design (OOD) is the process of using an object-oriented methodology to design a computing system or application.

- The main aim of Object-Oriented Design (OOD) is to improve the quality and productivity of system analysis and design by making it more usable.
- The elements of an Object-Oriented System are - Objects, Attributes, Behavior, Class, Methods and Message.
- Encapsulation is a process of information hiding.
- Abstraction is a process of taking or selecting necessary method and attributes to specify the object.
- All the classes in the system are related with each other.
- There are three types of object relationships - Aggregation, Association and Generalization.
- Inheritance is a feature that allows creating sub-classes from an existing class by inheriting the attributes and/or operations of existing classes.
- Polymorphism is the ability to take on many different forms.
- There are three main types of models that have been proposed by OMT - Object Model, Dynamic Model and Functional Model.
- The class diagram depicts a static view of an application. It represents the types of objects residing in the system and the relationships between them.
- Object diagrams are dependent on the class diagram as they are derived from the class diagram. It represents an instance of a class diagram.
- A state diagram is used to represent the condition of the system or part of the system at finite instances of time.
- A data flow diagram (DFD) maps out the flow of information for any process or system.
- The basic steps of system designing using OOM may be listed are - System Analysis, System Design, Object Design, and Implementation.



15.12 FURTHER READING

1. Booch, G. (2009). *Object-oriented analysis and design with applications*. Pearson Education India
2. Blaha, M., & Rumbaugh, J. (2005). *Object-oriented modeling and design with UML*. Pearson Education India.



15.13 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q.No.1: data, process

Ans to Q.No.2: Behavior

Ans to Q.No.3: James Rambaugh

Ans to Q.No.4: Polymorphism

Ans to Q.No.5: static

Ans to Q.No.6: False

Ans to Q.No.7: True

Ans to Q.No.8: dependency

Ans to Q.No.9: System Analysis

Ans to Q.No.10: True



15.14 MODEL QUESTIONS

Q1. What is Object Oriented Design? What are the elements of an Object-Oriented System?

Q2. What are the features of Object-Oriented Systems?

Q3. Describe the three main types of models that have been proposed by Object Modeling Technique(OMT).

Q4. Define the elements of a Class Diagram with an example.

Q5. What are the differences between a Class Diagram and an Object Diagram?

Q6. What are the different types of relationships in Class Diagrams?

Q7. What are the components of a State Diagram?

Q8. What are the different DFD Levels?

Q9. Describe the different phases of Object-Oriented Development.