DCECS-108/MCA-110/ PGDCA-110/BCA-109/ MCS-102

Uttar Pradesh Rajarshi Tandon Open University

C++ & OBJECT ORIENTED PROGRAMMING

Uttar Pradesh Rajarshi Tandon Open University

DCECS-108/MCA-110/ PGDCA-110/BCA-109/ MCS-102

C++ & OBJECT ORIENTED PROGRAMMING

BLOCK

1

PRINCIPLES OF OOP, OOP SYSTEMS AND ADVANCED CONCEPTS

Course Design Committee

Course Preparation Committee

Dr. Marisha — **Author (Block 1)**
Assistant Professor (Computer Science)
School of Science
UPRTOU, Prayagraj

Er. Pooja Yadav — **Author (Blocks 2, 3 and 4)**
Assistant Professor
Dept. of Computer Science and IT
M.J.P. Rohilkhand University
Bareilly

Prof. Ashutosh Gupta — **Editor**
Director (In-Charge)
School of Computer and Information Science
UPRTOU, Prayagraj

Dr. Marisha — **Coordinator**
Assistant Professor (Computer Science)
School of Science
UPRTOU, Prayagraj

ISBN : 978-93-83328-98-7

Printed and Published by Dr. Arun Kumar Gupta Registrar, Uttar Pradesh Rajarshi Tandon Open University, 2020.

[PYTHON-010]
DCECS-108/4

Printed By : Chandrakala Universal Pvt. 42/7 Jawahar Lal Neharu Road, Prayagraj.

BLOCK INTRODUCTION

In this block, we shall discuss the basic concepts of object oriented programming systems.

Unit-1 : In unit 1, we shall discuss object oriented paradigm. We shall discuss the advantages of object oriented languages over procedural languages. In the end, we shall briefly discuss the advantages of the object oriented language C++.

Unit-2 : In this unit, you will learn in detail about the features of object oriented programming. In particular, we shall discuss about the concepts of class, objects, inheritance, abstraction and polymorphism. These features will be used in programming object oriented systems. The basic concpts of these features will be discussed here. You will learn programming using these features in later units.

Unit-3 : In the last unit, we shall discuss advanced concepts of object oriented programming systems. In particular, we shall discuss about the concept of dynamism, program structuring and reusability. In the end, we shall discuss about how large object oriented projects are organized.

UNIT-I PRINCIPLES OF OBJECT ORIENTED PROGRAMMING

Structure

1.1 INTRODUCTION

This unit gives an introduction of the basic concepts of object oriented programming. We begin with object oriented programming paradigm and discuss the meaning of classes and objects, the concept of abstraction and why it is useful. Then, we will briefly discuss procedural languages and compare object oriented languages with procedural languages. We will then move on to discuss the basic features of object oriented programming languages and also about some commonly used object oriented languages. Later on, we will discuss the advantages of using C++. The last section will provide a summary of the topics learned in the unit which will be followed by review questions.

Objectives :

After learning this unit you should be able to:

1. Understand the object oriented paradigm
2. Differentiate between procedural and object oriented languages
3. Describe the basic concepts of object oriented programming
4. List the advantages of using C++ over C

1.2 OBJECT ORIENTED PARADIGM

Object Oriented Programming systems are based on the concept of classes and objects. Classes and objects form the basic building blocks of any object oriented programming language. Classes are abstract things

while objects form particular instances of classes. The word abstract literally means something which exists in one's mind. Abstraction is not a new concept; we use abstraction in our day-to-day life for almost everything that we deal with. Like a book is an abstract term while "Wings of Fire" and "My experiments with Truth" are particular instances of a book. Similarly, fruit is an abstract term while mango, guava, apple etc. form particular instances of fruit. Abstraction helps us in understanding the overall structure and behavior of an entity without going into finer details. As you might have guessed by looking at the example, there can be different levels of abstraction like mango is a fruit but there can be different varieties of mangoes, each of which forms an instance of the class Mango.

The discussion above makes it clear that we always use abstraction while dealing with real life entities although the level of abstraction may vary depending upon the needs. We hide those details of the objects which are not needed and focus on only those attributes which are useful for us according to our requirements. This reduces the complexity of the real world system and makes it easier for us to deal with so many things in an organized manner and use them for our advantage.

Any abstract entity is characterized by certain features and behaviours. For example, when we talk about the class book, there are certain features that come to mind, say it would be something of the shape of a rectangle with a number of pages bound together, it will have a cover page with a picture and a name describing what is written inside. It will have an author; a publisher etc. and it will be used for the purpose of reading or studying. These features and behaviours form the characteristics of the class and are used for defining a class. Any object oriented system involves thinking of the system in terms of classes and objects as instances belonging to different classes which interact with each other and produce the whole system as a result.

As you know, we use programming to solve the real world problems, the object oriented languages give us the freedom to map the real life entities into program objects and thus depict the real life system as closely as possible in our programs. Before going into deeper details of object oriented programming languages and the advantages of using OOPS (Object Oriented Programming Systems), let us first discuss procedural languages and some of their limitations.

Concept of Procedural Programming

In procedural programming, we divide the problem into a number of sub-problems or tasks and solving the problem involves writing procedures for doing the individual tasks. Thus we think of the problem as being composed of different modules and we write code for each module to solve the whole problem.

The programming languages which support procedural programming are called procedural programming languages. Some

examples of procedural programming languages are C, Pascal, Fortran etc. In fact, initially we had only procedural programming languages and no Object Oriented languages. Object oriented programming was developed later on as we started facing difficulty in solving real world problems while sticking to procedural programming. We will discuss the limitations of procedural programming languages in greater detail in later sections.

In procedural programming, as discussed earlier we write procedures for doing specific tasks. For example we may write an area function to calculate the area of say a rectangle. The code for doing this in C is given below:

```
#include<stdio.h>

int calcarea(int, int);

void main( )

{

        int side1, side2, area;

        printf("enter the lengths of two sides\n");

        scanf("%d%d",&side1,&side2);

        area = calcarea(side1, side2);

        printf("area = %d", area);

}

int calcarea(int side1, int side2)

{

        int area;

        area = side1 * side2;

        return(area);

}
```

The above program has a function named calcarea which takes as input the two sides of the rectangle and produces the area of the rectangle as output. Inside the main function the calcarea function is called with sides as arguments. The return value is stored in the variable area. This example shows you how procedures are written to solve problems in procedural programming. Let us move a step further and suppose your requirement changes now and you want calcarea function to take float values also as input. So, your rectangle can have non-integer values also as side lengths. For doing this, you will have to write a new code with sides defined as floats instead of integers.

Similarly, suppose you want to calculate the area of different geometrical figures not necessarily rectangle, then you need to write separate codes for calculating the area of each geometrical figure and each procedure will have to be given a separate name. This will reduce the readability of program and increase its complexity thus, making the program difficult to understand. It is not possible to reuse the same function name for doing similar type of tasks.

When procedural programming is used for designing large systems, it is common practice to make the data variables that will be used by many programs public so as to reduce the overhead of parameter passing in the function call. This however makes the data vulnerable to inadvertent changes by program functions. The changes made to the global data are difficult to track.

The important characteristics of Procedural Programming are given below:

1. The prime focus is on functions, which operate on data.
2. Data moves freely among different functions
3. Constant data values and the data that are to be shared among many functions are usually made global.
4. It follows top-down approach of program design.

1.3 LIMITATIONS OF PROCEDURAL PROGRAMMING

Some of the limitations of procedural programming languages are as follows:

Data is vulnerable to inadvertent changes : Since, this type of programming focuses mainly on functions and data moves freely among different functions, the data is vulnerable to inadvertent changes by the functions. Also, it makes the task of debugging difficult as it is hard to track the changes made to the data when it is accessed by many functions. Tracking changes in global data is even more difficult as global data can be changed by virtually any function if it is not specifically declared as a constant.

Real Life Entities cannot be represented properly : As already discussed, procedural programming does not naturally represent how we perceive real life systems. This poses difficulty in solving real world problems especially when the system is very large and complex.

Lack of Reusability : In procedural programming, if we want to do little modification in an existing function, we will have to write a new function. The same function cannot be reused for a slightly different scenario. Another related shortcoming is that one cannot have two functions with same name in a single program. This forces the programmer to choose

different names for functions that actually do very similar job. This reduces the clarity and understandability of the code.

Difficulty in Testing and Debugging : This is the outcome of the limitations that we have already discussed. If we let the data move freely among different functions, tracking changes in the data values will become difficult. Tracking the changes in the data gets even more difficult for large systems (which is the case for most real world systems). Large systems naturally have more modules resulting in a lot of data and thus posing difficulty in tracking the changes made in the data by different modules.

Lack of reusability also increases the burden of testing, as every new module will have to be tested in totality even when it is only slightly different from an already written and tested module. This increases the work of the programmer and also increases the time required for delivering the system.

To overcome these limitations of procedural programming languages, object oriented programming was introduced. We will now briefly discuss the basic concepts of object oriented programming. Detailed explanation of these concepts will follow in subsequent units.

1.4 BASICS OF OBJECT ORIENTED PROGRAMMING

Any object oriented programming system involves the following concepts:

1. **Objects**
2. **Classes**
3. **Abstraction**
4. **Encapsulation and Information Hiding**
5. **Polymorphism**

Objects : Objects form the basic runtime entities in an object oriented programming environment. They might represent real life entities or program objects. Objects contain data and methods that operate on the data. They form instance of classes and occupy space in memory.

Classes : Classes are the user-defined data types which represent the objects. The data and methods that operate on data are tied together to form a user-defined data type that we call as class. We can define any number of instances of a class and each such instance will be called an object of that class.

Abstraction : Abstraction as we have already discussed, implies representing the important features of a system while avoiding finer details. This reduces the complexity of very large systems. We consider

program entities and model their interaction without paying attention to the internal structure of an entity.

Encapsulation and Information Hiding : In object oriented programming we define program entities as objects which contain data as well as the methods that operate on the data. This idea of representing the data and the operations together as one entity is called **encapsulation**. Any entity thus formed is isolated from rest of the system and can only be accessed through the public interfaces of the class i.e. the functions.

Since, the data is accessible to other entities in the system only through public interfaces of a class; the data is protected from any inadvertent changes. This forms the basic idea of **information hiding**, as any information that is stored in the system is hidden from other entities of the system. All the interactions among different entities are governed by the functions that are defined for an entity which also act as the interfaces of the class.

Polymorphism : Polymorphism literally means having more than one form. Polymorphism allows us to use the same function to perform similar tasks. For example, using polymorphism you don't have to write two different functions for adding two numbers and for adding three numbers. Similarly, the same function which is used for calculating the sum of two real numbers can be used for calculating the sum of two complex numbers. It is the property of polymorphism which allows the same function to exhibit multiple forms.

The important features of any object oriented system may be listed as follows:

1. It follows bottom-up approach of program design.
2. Emphasis is laid on data rather than functions.
3. Data are hidden and can be accessed only by the functions that operate on the data.
4. The data and the functions that operate on the data are tied together in the form of objects and are thus isolated from all other entities of the system.
5. It allows reusability of the already written and tested modules and thus reduces the time required to build a system.

1.5 OBJECT ORIENTED LANGUAGES

Object oriented programming started for the first time in 1960s. Simula67 was the first programming language which used objects. An object oriented programming language called Smalltalk was developed to program the Dynabook, a proposed personal computer for children. Though, Smalltalk was used independently of Dynabook in a variety of applications. In the early 1980s Bjarne Stroustrup integrated object

oriented features in C programming language and the resulting language was called C++ (drawing inspiration from the increment operator "++'" in C). C++ was widely used commercially. Gradually many object oriented programming languages were developed. The most commonly used object oriented programming languages used nowadays include Java, Python, Ruby, Perl, .NET, C# etc.

1.6 ADVANTAGES OF C++

As discussed in previous sections, C++ has many advantages over C. The important ones are given below:

1. It provides protection to the data.
2. Facilitates reusability of code
3. Complex systems can be designed easily and conveniently
4. Increases clarity of code
5. Reduces the time and cost of debugging
6. Better error-handling and type-checking
7. It is easy to learn and the C++ also accepts C code.

1.7 SUMMARY

In this unit we have discussed about the basics of object oriented systems. Object oriented systems are based on the concepts of classes and objects. Class is an abstract entity while object forms an instance of the class. Object oriented systems provide many features such as abstraction, data hiding, encapsulation inheritance and polymorphism. We discussed briefly about each of these features. We also discussed about procedural programming, its limitations and advantages of object oriented programming over procedural programming. Then we discussed about some of the object oriented programming languages. We have concluded by taking the specific example of C and C++ and discussing the advantages of using C++ over C.

1.8 REVIEW QUESTIONS

1. Briefly explain the basic features of object oriented systems.
2. Discuss the limitations of procedural programming.
3. List three object oriented languages other than those already mentioned in the text.

UNIT-II
OBJECT ORIENTED PROGRAMMING SYSTEMS

Structure :

2.1 INTRODUCTION

In this unit we will discuss in detail the features of object oriented programming systems. This will include the concepts of class, inheritance, abstraction, encapsulation and information hiding, polymorphism and overloading. Each section will have one concept explained with the help of examples. The last section will provide a summary of the concepts learned which will be followed by review questions.

Objectives

After learning this unit you should be able to:

1. Understand the basic features of object oriented programming languages
2. Understand the importance and utility of each of the features

2.2 CLASS

As discussed earlier, classes are abstract entities which bind the data and the methods together. Thus, a class defines the characteristics of the objects and also their behavior. Objects form particular instances of a class. Let us understand the concept of classes and objects with the help of an example. Suppose we want to create a class triangle to calculate the area of a given triangle. Then, the data members of this class will be the lengths of each of the edges of the triangle while the methods will include the function to calculate the area of the triangle. Also if we give particular values to all the edges of the triangle then it will form an instance i.e. an

object of the class. A schematic diagram of the class Triangle is given below.

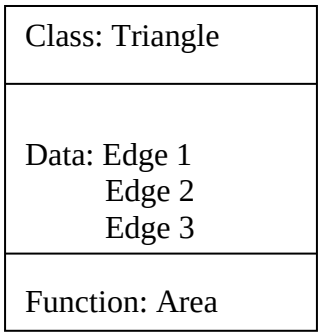


Fig. 1 : Description of the class triangle

Class is a user defined data type and can represent any real life entity or a program object. For example, a class may represent a person, a company, a geometrical figure or it might represent time, a vector or any other thing. Classes are the basic program entities in an object oriented programming system i.e. any object oriented programming system can be viewed as a system of classes interacting with each other through passing messages. The diagram showing an object oriented programming system is given in Fig 2:

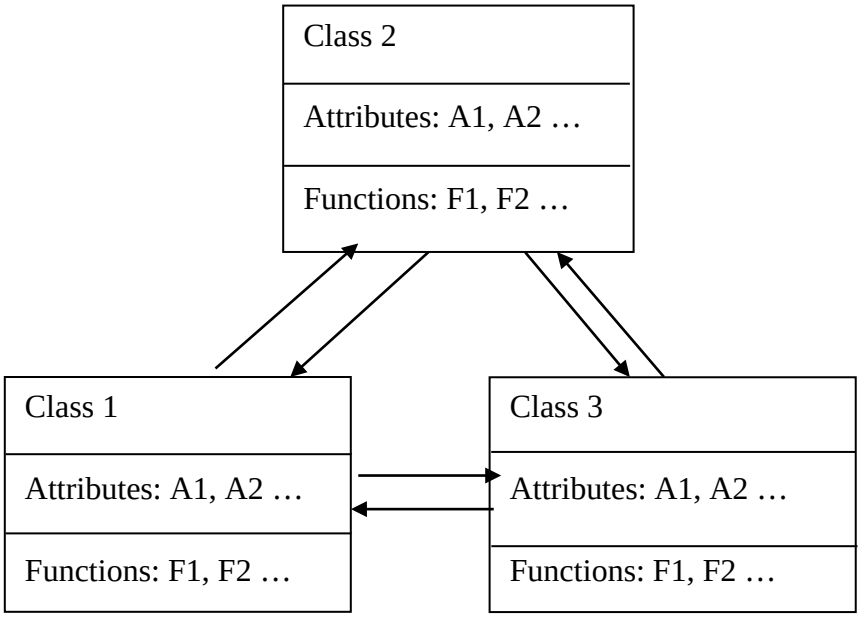


Fig. 2: An Object Oriented System

2.3 INHERITANCE

The concept of inheritance has been taken from real life. As children inherit genetic traits from their parents, similarly new class can be derived from an existing class. Thus, inheritance is the process by which we can create new classes using an existing already written class. Inheritance allows **reusability** of the code. Thus, if we write a class and at a later date we need to modify it slightly then we can simply create a new class which inherits the existing class instead of creating a new class from scratch. Using inheritance we can also combine the features of two or more classes and make a new class.

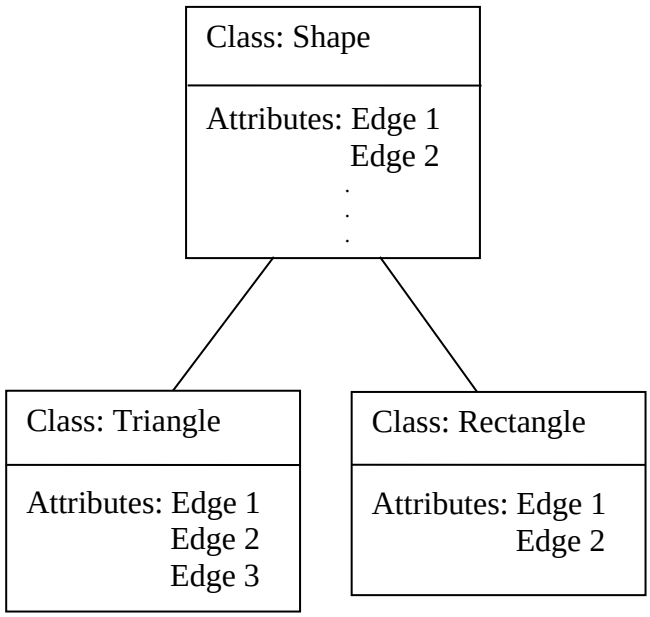
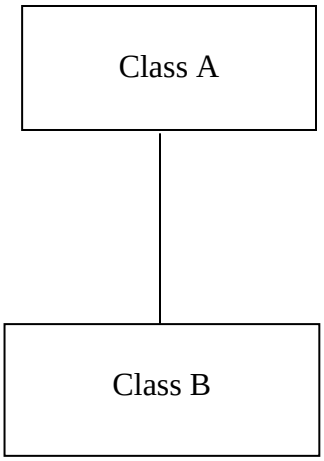


Fig. 3 : Example showing Inheritance

Suppose we define a class Shape to calculate the area of a geometric shape. This class contains the edge lengths of the given geometric shape as data members and a method to calculate the area of the shape as shown in Fig 2. Now, we can use this class to define new classes for different shapes such as triangle, rectangle etc., each of which will inherit the Shape class and can define their number of edges in according to the shape. The class which acts as the parent class is called base class while the new class which inherits the properties of the base class is called the derived class.

There are two broad types of inheritance, single inheritance and multiple inheritance based upon how the child class derives the properties and behaviors from the parent class. In single inheritance the derived class inherits from a single parent class while in multiple inheritance, the derived class has more than one parent as illustrated in Fig4.

Single Inheritance :



Multiple Inheritance :

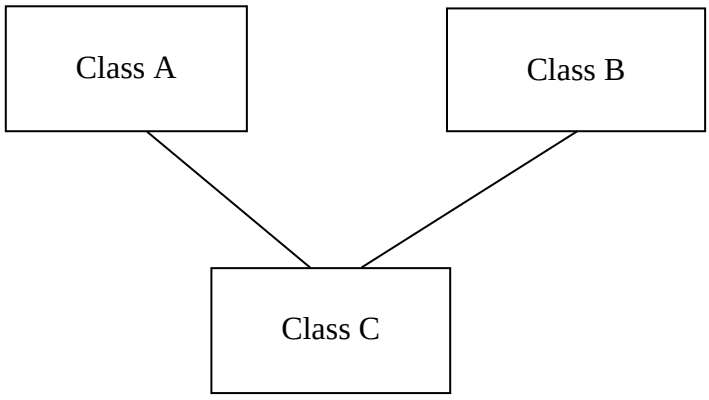


Fig. 4 : Figure showing single and multiple inheritances

2.4 ABSTRACTION

In object oriented programming we achieve abstraction through classes. A class is used to describe what an entity does without detailing about how it does. For example the Triangle class in our previous example calculates the area of a given triangle without describing how the area is calculated. The methods of the class act as public interfaces of the class which can be used to perform the functionalities provided by the class.

Abstraction is especially helpful if we are designing a large system as with abstraction one can design units which perform definite tasks and interact with other units through clearly defined interfaces thus making it

easier to track the behavior of an entity in different scenarios and also its interaction with other entities.

2.5 ENCAPSULATION AND INFORMATION HIDING

Encapsulation in Object Oriented Programming systems implies binding data and function together. Unlike procedural programming where data is allowed to move freely among different functions making it vulnerable to inadvertent changes, in OOP data is bound together with the function. What this really means is that whatever operations are allowed on a particular data item are described clearly and the methods to perform those operations are written and the method and the data are both tied together in the form of a class.

Thus, after we have defined a class, no function other than the once written in the class can access or modify the data associated with the class. Of course, there are exceptions to this rule and in some cases outside functions may access or modify the data of a class but then these exceptional functionalities of object oriented languages are used only when it is extremely necessary to do so and using them frequently or unnecessarily is against the basic idea of object oriented programming systems and is therefore, considered as a bad programming practice.

2.6 POLYMORPHISM AND OVERLOADING

Polymorphism is a Greek term and it means having more than one form. In Object Oriented Programming Systems, using polymorphism we can create two functions with the same name to perform two different but closely related operations. For example, the Area function in Fig 4 can be used to calculate the area of different kinds of shapes in the Shape class.

Similarly, we might want to define a function sum which can calculate the sum of two real numbers as well as the sum of two complex numbers depending upon whether the input numbers are real or complex. When we are using the same function name to describe more than one function, the program needs to do decide which function to use in a particular case. This decision as to which method is to be used for a given input may be taken at compile time or at run-time resulting into two different types of polymorphisms namely compile-time polymorphism (achieved through function and operator overloading) and run-time polymorphism (achieved through virtual functions). Many object oriented programming languages such as Java support only run-time polymorphism and not compile-time. Usually, pure object oriented languages support only run-time polymorphism.

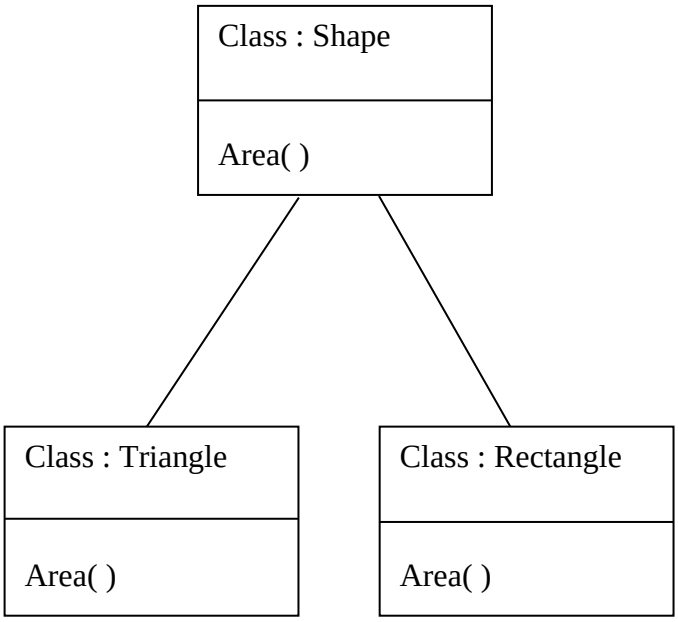


Fig 5 : Example showing polymorphism

When we define a function or an operator to behave differently under different inputs, the method or the operator is said to be overloaded. Both function and operator overloading form example of compile time polymorphism whereas run-time polymorphism is achieved in C++ through virtual functions. We will discuss these concepts in more detail in later sections.

Object oriented systems are represented using UML (Unified Modeling Language) diagrams. There are thirteen different types of UML diagrams which are used to describe any object oriented system. For example, the UML class diagrams are used to represent the classes in an object oriented system. The schematic diagrams that we have used to describe classes in this text are also examples of UML class diagrams.

2.7 SUMMARY

In this unit we have learned about object oriented programming systems. We discussed about classes and objects and also how object oriented programming systems achieve abstraction and encapsulation using classes. We discussed inheritance which is based on the concept reusability. Inheritance allows us to reuse an existing class to describe a new class which is slightly different from the existing class. There are different types of inheritance which we will discuss in later blocks. We have also discussed polymorphism which is extensively used to implement inheritance. Using polymorphism we can have the same method behaving differently under different input conditions. All these concepts form the

basic features of any object oriented programming system. Different object oriented programming languages provide different ways of implementing these features. In the later blocks of the material we will discuss in detail the features available in C++ and also learn how to use them.

2.8 REVIEW QUESTIONS

1. Discuss the important features of object oriented programming systems.
2. How is abstraction different from encapsulation?
3. What do you understand by the term “Information Hiding”? Why is it important to hide certain information in a system?

UNIT-III ADVANCED PROGRAMMING CONCEPTS

Structure

3.1 INTRODUCTION

In this unit we will learn some of the advanced concepts related to programming. More specifically, we will learn about dynamism supported in C++ programming language. This will include dynamic typing, dynamic binding, late binding and dynamic loading. We will start with an introduction to these terms and then discuss about these features as they are available in C++. Then we will discuss about how object oriented programs are structured and also about reusability. Lastly, we will discuss how the object oriented projects are organized. This will be followed by a summary of the topics learned and review questions.

Objectives :

After learning this unit you should be able to understand:

1. The concept of dynamism in programming languages
2. How dynamism helps in designing better programs
3. How dynamism is implemented in C++
4. How to structure programs and how to organize object oriented projects
5. The concept of reusability and its importance

3.2 DYNAMISM

Dynamism in programming languages means reducing the amount of information that needs to be supplied during compilation and delaying much of the decisions such as type checking, binding of methods, loading, linking etc. until the program is run. Therefore, with dynamism such decisions are taken by the programming environment at the runtime instead by the compiler during the compilation of the code. Dynamism provides more and more flexibility to the users in terms of what can be done using the programming language. Many of the features of object oriented programming such as inheritance, reusability, polymorphism etc. are possible only because of dynamism. Dynamism can be of different types as also different programming languages support different types of dynamism. We will now discuss different types of dynamism in detail.

3.2.1 DYNAMIC TYPING

Type checking or simply typing in a programming language is the process through which the translator verifies whether all the constructs in the program such as the variables, constants, procedures etc. are written correctly or not. All the constructs of the programming languages need to follow the rules and constraints defined by the programming language and it is the job of the translator to verify if the constructs used in the code follow the language rules. This process of verification is known as type checking. Type checking can be of two types **static** and **dynamic**. In static typing the type checking is done by the translator before program execution and any type error is reported as compile-time error which prevents the execution of the program. In dynamic typing on the other hand, the type information is checked at runtime.

In strongly typed languages, all the type errors have to be caught before program execution and therefore, such languages must be statically typed. C compilers provide static type checking however; C language is not strongly typed as many of the type errors are removed automatically through conversion code with or without a warning message, thus not preventing the program from running. C++ provides stronger type checking than C though most of the type errors are reported as warning rather than compile-time error thus allowing the program to execute. Therefore, one must be very careful while ignoring the warnings that are generated by the compiler. Java is also a statically typed language.

Many scripting languages such as Perl, Python and PHP are dynamically typed. The biggest advantage of dynamically typed languages is that the variables need not be declared before they are used. For example the following code is totally valid in Python.

```
/* Python code */
x = 10;          // variable is directly used
```

However, dynamic typing has the disadvantage that any typing error in variable names will not cause the compiler to prompt an error message. The compiler will simply treat them as two different variables that may lead to serious consequences.

3.2.2 DYNAMIC BINDING

We have already studied polymorphism in previous units and discussed how the same function name can be used to define similar operations. For example, both the derived classes in Fig5 have their own implementation of the area method although both the methods have the same name.

Polymorphism can be of two types compile time polymorphism and run-time polymorphism. When the decisions as to which version of a function to invoke is taken at compile time then it is known as compile time polymorphism while if this decision is delayed until runtime then it is known as runtime polymorphism. Runtime polymorphism is possible due to late binding and dynamic binding.

Dynamic binding as the name implies involves dynamically deciding the correct function to invoke in response to a function call. In C++ dynamic binding is implemented through virtual functions. We will discuss virtual functions in detail when we will study inheritance. In most pure object-oriented languages such as Java, dynamic binding always applies as all the methods are implicitly virtual. Dynamic binding puts significant runtime overhead as the runtime environment must check which object is associated to which class in order to invoke the correct method. You should note however that it is possible to implement dynamic binding while retaining static typing.

3.2.3 LATE BINDING

There is very subtle difference between dynamic binding and Late Binding. Here also the decision as to which method to invoke in response to a function call is delayed until run time but there are restrictions on the type of the objects that can be used to invoke late binding. Let us discuss this in a bit detail.

C++ programming language implements late binding. Here, runtime polymorphism is achieved through virtual functions. To invoke late binding i.e. runtime polymorphism in C++, the object pointed to by a pointer can be the object of the base class or any of its derived class. Here, the compiler cannot decide the method belonging to which of the classes is to be invoked and this decision is delayed until the program is run. Thus, this is known as late binding.

Dynamic Binding is however a much broader term. The languages that allow dynamic binding do not put compile time restrictions on the objects. Therefore, the object may belong to literally any class and thus, the decision of invoking a particular function is totally dynamic. More

specifically, while static and late binding can ensure that the method that will be invoked actually exists; in dynamic binding there is no way the compiler can ensure that the method actually exists. Thus, unlike static and late binding, **dynamic binding may fail at runtime**.

3.2.4 DYNAMIC LOADING

Historically, the program code was written in a single file and the complete program was loaded into the memory before execution. With the advancement of technology or to be more specific, with the development of the concept of virtual memory, it is now possible to have only a part of the program in memory and the remaining part can be loaded dynamically into the memory on demand.

This concept of loading only that part of a program which is essential for program execution and loading other parts as and when they are needed is known as dynamic loading. The reason being that unlike static loading in which the whole program code needs to be in memory before program execution begins, the loading of the modules in dynamic loading systems takes place dynamically when they are required. Thus, dynamic loading implies that a module will not be loaded into the memory unless it is called. Many programming languages now provide the facility of dynamic loading.

The advantage of dynamic loading is that the program can be built in parts. Therefore, the software developers may build only a part of the software and deploy it to the users while new facilities may be added to the software later. Thus dynamic loading helps make the applications extensible. This however makes the implementation of modules hard.

With dynamic loading it is a necessary requirement to have clear cut interface between modules of the program so that the working of one module is not affected by that of the other. It is crucial for success of the concept of dynamic loading that a module can be developed or modified anytime without affecting the other modules. Object oriented programming comes handy in this case as the objects define the program modules with clear cut interfaces among them and can thus easily facilitate dynamic loading.

3.3 STRUCTURING PROGRAMS

The structure of object oriented programs is usually thought of in terms of a) class hierarchies and b) the interaction between objects.

The class hierarchies define the structure of the system. It describes how the classes are arranged into the system and how they relate to each other. Thus, as considered in the previous examples, the shape class describes the main module of the system while the classes triangle and rectangle are the subclasses. These three combined together describe

the system in terms of the objects present in the system and how the different objects stand in relation to each other in the system.

The objects in an object oriented system interact with each other through message passing. This interaction between objects describes how the system works. One object can send a message to another object requesting it to do some task; the receiver in turn does the required processing at its end and may send some message back to the sender or to some other object in the system. These relationships need not be static and may change dynamically as the program runs.

3.4 REUSABILITY

Reusability is one of the most important features of object oriented programming systems. Reusability in OOP implies that it is possible to use existing modules again in a related context either with or without modification. Object oriented languages provide facilities to write codes in such a way that it can be reused. The extent of reusability supported by a language may however vary.

Organizing system in terms of classes provides reusability as the existing classes can be extended with inheritance and new functionalities may be added to it as also new data members can be added. We now discuss the different ways in which classes can be modified to achieve reusability. These are listed below:

1) By increasing the number of data members or member functions present in a class

2) By restricting the number of data members or the member functions present in a class. This may be achieved by restricting the functions that operate on the data. The mechanism of restriction is however not a common programming practice and is therefore not found in many programming languages. It seems more natural to extend already restrict classes instead of restricting an existing class.

3) By redefining one or more of the operations defined in a class

4) By using the concept of polymorphism i.e. by making the same function act differently depending upon the type of object which is used to call the function.

Reusability of the code depends on a number of factors other than the facilities provided by the programming which include the reliability and efficiency of the code, the clarity of the documentation, the simplicity of the interface etc.

3.5 ORGANIZING OBJECT ORIENTED PROJECTS

Object oriented programming systems are generally used to design large scale systems. Designing large scale systems poses a lot of challenges which are not usually faced in small systems. Also large systems are usually designed by a group of people and not by a single person. As a result, it is necessary to organize the object oriented projects in a systematic manner otherwise the programmers may very easily lose track of the system. Following are the broad guidelines which should be followed while designing large object oriented systems:

1) Abstraction and Modularization : At abstract level it is easy to identify the main modules in a system, they can be understood easily by the people involved and also it would be easy to divide the work among different groups. The abstract modules can then be refined and further divisions may be created.

2) Simple Interface : The objects in an object oriented system interact through sending messages to each other using public interfaces of the class. It is therefore, important to keep the interface simple and clear so that new classes can be added as also new connections can be added to the system with least overhead.

3) Reusability : The strength of object oriented system lies in its reusability. The more you reuse the existing code the less effort you will need to put in designing the new system. It is therefore important to use the generic code though the mechanism of inheritance and also to reuse the already written and tested modules.

4) Dynamism : As already discussed dynamism allows the program to leave much of the decision regarding function call or association of objects to a type to runtime. This reduces the amount of information which needs to be provided during compilation and hence there is less effort required in coordinating different modules to work. Thus, the designer should focus on increasing the dynamism in the system.

3.6 SUMMARY

In this unit we studied about some of the advanced concepts related to object oriented programming systems. We discussed about the concept of dynamism. Dynamic typing allows the type checking of the program objects to be delayed until runtime. Similarly, dynamic loading allows the program to run with only a part of the program actually loaded into the memory. The modules are dynamically loaded as they are needed. Dynamic binding delays the decision as to which particular method is to be invoked in response to a function call until runtime. Late binding is also

a similar concept; it however puts some restrictions on the type of objects which can be used for invoking a method dynamically. We discussed about reusability and how it is achieved in object oriented programming followed by some guidelines for structuring programs and organizing object oriented projects.

3.7 REVIEW QUESTIONS

Q1. What do you mean by dynamism in object oriented systems? What are the different types of dynamism? Briefly explain.

Q2. Explain the difference between dynamic binding and late binding with example.

Q3. Suppose you want to automate the various activities that take place in a library. Identify the major objects that should be present in the system.

Bibliography :

1. The C++ Programming Language by Bjarne Stroustrup
2. Object Oriented Programming by Joyce Farrell
3. Programming Languages Principles and Practice by Kenneth C. Louden
4. Object Oriented Programming with C++ by E Balagurusamy

Uttar Pradesh Rajarshi Tandon Open University

DCECS-108/MCA-110/ PGDCA-110/BCA-109/ MCS-102

C++ & OBJECT ORIENTED PROGRAMMING

BLOCK

2

OVERVIEW OF C++, CLASSES AND OBJECTS

Course Design Committee

Prof. Ashutosh Gupta — **Chairman**
Director-In charge,
School of Computer and Information Science
UPRTOU, Prayagraj

Prof. U. S. Tiwari — **Member**
Dept. of Computer Science
IIIT Prayagraj

Prof. R. S. Yadav — **Member**
Department of Computer Science and Engineering
MNNIT-Allahabad, Prayagraj

Dr Marisha — **Member**
Assistant Professor (Computer Science)
School of Science, UPRTOU, Prayagraj

Mr. Manoj K. Balwant — **Member**
Assistant Professor (Computer Science)
School of Science, UPRTOU, Prayagraj

Course Preparation Committee

Dr Marisha — **Author (Block 1)**
Assistant Professor (Computer Science)
School of Science
UPRTOU, Prayagraj

Er. Pooja Yadav — **Author (Blocks 2, 3 and 4)**
Assistant Professor
Dept. of Computer Science and IT M.J.P.
Rohilkhand University Bareilly

Prof. Ashutosh Gupta — **Editor**
Director (In-Charge)
School of Computer and Information Science
UPRTOU, Prayagraj

Dr Marisha — **Coordinator**
Assistant Professor (Computer Science)
School of Science
UPRTOU, Prayagraj

ISBN : 978-93-83328-98-7

Printed and Published by Dr. Arun Kumar Gupta Registrar, Uttar Pradesh Rajarshi Tandon Open University, 2020.

DCECS-108/32

Printed By : Chandrakala Universal Pvt. 42/7 Jawahar Lal Neharu Road, Prayagraj.

BLOCK INTRODUCTION

In this section we discuss the overview of this block's content. This block consists of the following units:

Unit-4 : Overview of C++

In this unit we'll discuss about the basics of C++, specifically, identifier, keyword, data type variables and operators, with the help of examples explain how we can use them and their syntax and what the need of them is.

Unit-5 : Class and Objects

In this unit we'll discuss the differentiation between class and structure and each and everything about Class and Objects in C++, specifically, characteristics of OOPs and their purpose and also define use of friend function in different ways and static member with examples.

Unit-6 : Object initialization and clean-up

In this unit we'll discuss about initialization of object with the help of Constructor and its types and characteristics, and also explain the role of destructor and concept of nested class and examples.

UNIT-4 OVERVIEW OF C++

Structure

4.1 INTRODUCTION

In this unit, the focus is to explain the basics of C++. It gives the full explanation to building blocks of any C++ program. We will learn the data type in C++, variables and its naming convection. It will also include the type compatibility among data type for data storage and transfer among variables. It will also give the basic control structure of C++

program. At the end of this unit, information about structure and functions in C++ will also be given in brief.

4.2 OBJECTIVE

The objective of this unit is to explain the basic idea of C++ like variable, constants, data types, and also explain the working of function and how to define class and object with basic characteristics of OOPs.

4.3 TOKENS

Entire C++ program is made up of tokens. Token is the basic building block of any C++ program. We may also understand the token as most important part of any program. Token is the smallest meaningful unit in the program. Tokens are used for different purposes at different places.

For example, if we think of any house, it is made up of different elements like cement, iron, wood, bricks, doors, windows, electric elements, bath elements etc. But builder has to decide where to put which element. In the same way token is the basic element of any program. One program is made up of many tokens. Every token is used for any particular purpose. Some tokens have predefined purposes and some tokens are defined by programmer. As a programmer we have to decide, where to use which token. We may also define new tokens as per our need. Simply we may say that, anything we write down in program is token.

For example: if, for, while, switch, class, all variables name, all operators are tokens.

There are 5 types of tokens in the program-

1. Keyword
2. Identifier
3. Operators
4. Literals (Constants)
5. Punctuation marks

4.3.1 KEYWORDS

Keywords are the reserve words in compiler. Meaning of the keywords are fixed in the compiler and there meaning cannot be changed. There are fixed number of keywords in compiler. Keywords in C++ are also case sensitive.

There are 32 in C++ which were also in C language :

auto	const	double	float	int	short	struct	unsigned
break	continue	else	for	long	signed	switch	void
case	default	enum	goto	register	sizeof	typedef	volatile
char	do	extern	if	return	static	union	while

There are another 30 more reserved words in C++ those were not in C :

asm	dynamic_cast	namespace	reinterpret_cast	try
bool	explicit	new	static_cast	typeid
catch	false	operator	template	typename
class	friend	private	this	using
const_cast	inline	public	throw	virtual
delete	mutable	protected	true	wchar_t

There are 11 C++ reserved words, which are not essential in standard ASCII character set, but they are added for some of the C++ operators :

and	bitand	compl	not_eq	or_eq	xor_eq
and_eq	bitor	not	or	xor	

4.3.2 IDENTIFIERS

Identifiers are the words which don't have predefined meaning in compiler. Identifiers are the words which are used to identify something. Variables names are the best example of identifiers. There are two categories in identifiers: first library and second user defined.

In the first category, there are some words defined in the library i.e. getch, clrscr etc. These words are not given in compiler but they are given in header files. We need to first include the header files for such words. Purpose of such library identifiers are defined in library, programmer just need to call the identifier from library. As library identifiers are defined in library, there meaning will be same in all programs.

In the second category, programmer has to define the words i.e. variable names, array name, function name, object name, structure name etc. Programmer need not to include any header file in such case. One user defined identifier, used for one purpose, may be used for other purpose in different program.

Naming Conventions : There are some rules for defining the identifiers, those are called naming convection. Naming convention in C++ are as given:

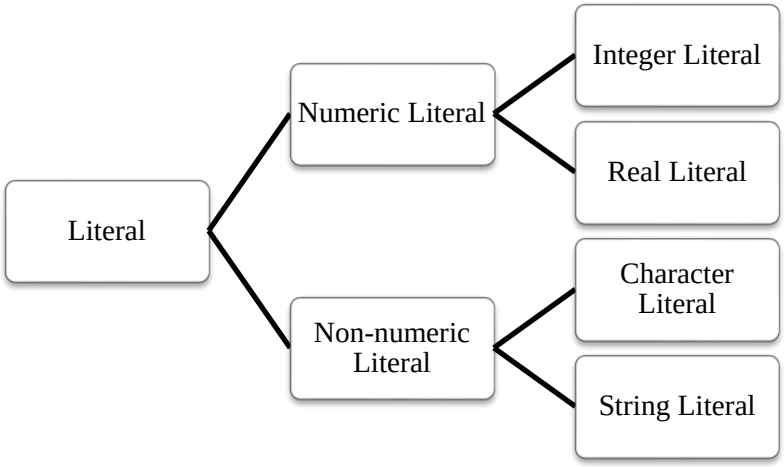
- May contain only: A to Z (Capital alphabets)
  Z to z (Lowercase alphabets)
  0 to 9 (Digits)
  _ (underscore)
- Must begin with alphabet or underscore.
- Identifiers are case sensitive.
- Identifiers must be unique within block. (No duplicity)

Identifiers should be in small letters because capital letters are used to identify constant values.

Valid identifiers	Invalid identifiers
age	123abc
name	abc 123
name1	student's
height1	Student name
address_1	int
weight	ur-name
rollno	roll number

4.3.3 LITERALS

Literals are the actual values, which are written in the programming code. The value of constant can't be changed.



Integer literals : Any whole number
e.g. 1, 2, 123, 0, -123 etc.

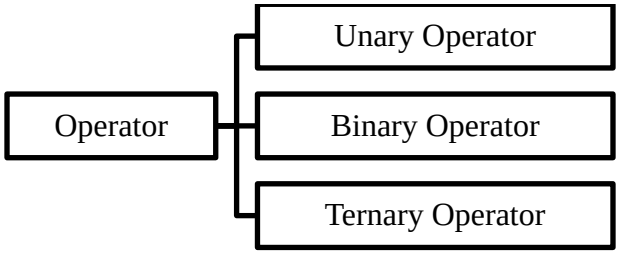
Real literal : Any real number (floating / double)
e.g. 1.23, 1.23f, 1.23F, 0.0, 1.0, -23.34, 3.14 etc.

Character literal : Any single letter within single quotes or escape sequences.
e.g. ‘a’, ‘x’, ‘1’, ‘ ’, ‘+’, ‘A’, etc.

String literal : Anything or nothing with double quotes
e.g. “ankur”, “C++”, “”, “ankur mittal”, “123”, “a”, “a-123” etc.

4.3.4 OPERATORS

Operators are symbols which take one or more operands or expressions and perform any arithmetic or logical computations. The values on which operator perform any action are known as operands. E.g. in a+b: + is the operator and a, b are the operands. There are 3 types of operators.



- Unary Operator : Unary operators are those operators which perform action on only one operand. There are some unary operators e.g. sizeof( ), negation operator, increment, decrement operator etc.

- Binary operator : Binary operators are those operators which perform action on exactly two operands. Mostly operators are the binary operators e.g. +, -, *, >, = etc.
- Ternary Operator : Ternary operators are those operators which perform action on three operands. There is only one ternary operator i.e. ternary operator (? :).

Operators are categorized in the following categories:

- Arithmetic Operators C++ provides five basic arithmetic operators. These are summarized in Table.

Operator	Name	Example
+	Addition	12 + 4.9 // gives 16.9
-	Subtraction	3.98 – 4 // gives -0.02
*	Multiplication	2 * 3.4 // gives 6.8
/	Division	9 / 2.0 // gives 4.5
%	Remainder	13 % 3 // gives 1

- Relational Operators

  C++ provides six relational operators for comparing numeric quantities. Relational operators evaluate to 1 (representing the true outcome) or 0 (representing the false outcome).

Operator	Name	Example
==	Equality	5 == 5 // gives 1
!=	Inequality	5 != 5 // gives 0
<	Less Than	5 < 5.5 // gives 1
<=	Less Than or Equal	5 <= 5 // gives 1
>	Greater Than	5 > 5.5 // gives 0
>=	Greater Than or Equal	6.3 >= 5 // gives 1

- Logical Operators

  C++ provides three logical operators for combining logical expression. These are summarized in Table. Like the relational operators, logical operators evaluate to 1 or 0.

Operator	Name	Example
!	Logical Negation	!(5 == 5) // gives 0
&&	Logical And	5 < 6 && 6 < 6 // gives 1
||	Logical Or	5 < 6 || 6 < 5 // gives 1

- Bitwise Operators

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

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

- Increment/Decrement Operators

  The auto increment (++) and auto decrement (--) operators provide a convenient way of, respectively, adding and subtracting 1 from a numeric variable. The examples assume:

  int k = 5;

Operator	Name	Example
++	Auto Increment (prefix)	++k + 10 // gives 16
++	Auto Increment (postfix)	k++ + 10 // gives 15
--	Auto Decrement (prefix)	--k + 10 // gives 14
--	Auto Decrement (postfix)	k-- + 10 // gives 15

- Assignment Operator

  The assignment operator is used for assigning a value at some memory location. Its left operand is an l-Value, and its right operand is an r-Value. And r-Value is assigned into l-Value.

  Assignment may also be used in shorthand form like :

Operator	Example	Equivalent To
=	n = 25	n = 25
+=	n += 25	n = n + 25
-=	n -= 25	n = n - 25
*=	n *= 25	n = n * 25
/=	n /= 25	n = n / 25
%=	n %= 25	n = n % 25
&=	n &= 0xF2F2	n = n & 0xF2F2
|=	n |= 0xF2F2	n = n | 0xF2F2
^=	n ^= 0xF2F2	n = n ^ 0xF2F2
<<=	n <<= 4	n = n << 4
>>=	n >>= 4	n = n >> 4

- Conditional Operator

  The conditional operator takes three operands. It has the general form :

  operand1 ? operand2 : operand3

  First operand1 is evaluated, which is treated as a logical condition. If the result is nonzero then operand2 is evaluated and its value is the final result. Otherwise, operand3 is evaluated and its value is the final result. For example:

```
m = 1; n = 2;
min = (m < n ? m : n);          // min receives 1
```

Note that of the second and the third operands of the conditional operator only one is evaluated. This may be significant when one or both contain side-effects (i.e., their evaluation causes a change to the value of a variable). For example, in

```
min = (m < n ? m++ : n++);
```

m is incremented because m++ is evaluated but n is not incremented because n++ is not evaluated.

Because a conditional operation is itself an expression, it may be used as an operand of another conditional operation, that is, conditional expressions may be nested. For example:

```
int m = 1, n = 2, p =3;
```

int min = (m < n ? (m < p ? m : p): (n < p ? n : p));

4.3.5 PUNCTUATION MARKS

A punctuator is a token that has syntactic and semantic meaning to the compiler, but the exact significance depends on the context. A punctuator can also be a token that is used in the syntax of the pre-processor. Some characters in C are used as punctuators, which have their own syntactic and semantic significance. Punctuators are not operators or identifiers. Here is the list of punctuators-

Punctuator	Use	Example
< >	Header file name	#include <limits.h>
[ ]	Array delimiter	char a[7];
{ }	Initializer list, function body, or compound statement delimiter	char x[4] = {'H', 'i', '!', '\0' };
( )	Function parameter list delimiter; also used in expression grouping	int f (x,y)
*	Pointer declaration	int *x;
,	Argument list separator	char x[4] = { 'H', 'i', '!', '\0'};
:	Statement label	labela: if (x :=,= 0) x += 1;
=	Declaration initializer	char x[4] = { "Hi!" };
;	Statement end	x += 1;
...	Variable-length argument list	int f (int y, ...)
#	Pre-processor directive	#include "limits.h"
' '	Character constant	char x = 'x';
" "	String literal or header name	char x[] = "Hi!";

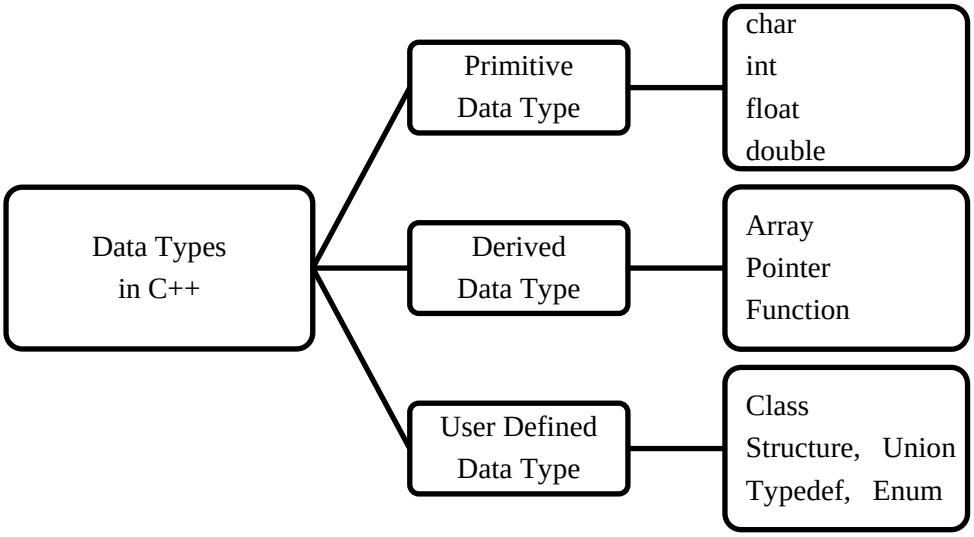
Check Your Progress :

In a AND operator what is necessary for execution?

4.4 DATA TYPES

Data type is the collection of values which can be assigned to the variable. Data type is mandatory to be declared before the use of any variable because it determines the type of operations those can be performed on the declared variables. Suppose arithmetic operation can be performed on the numeric values only and string operation can be performed on the char data type only. Every data type has its fixed memory consumption at compile time. The range of values is also there for the numeric values.

Data type can also be treated as the type of the variable. A data type in programming, a classification identifying one of various types of data, as floating-point, or integer, stating the possible values for that type, the operations that can be done on that type, and the way the values of that type are stored.



4.4.1 BASIC DATA TYPE (PRIMITIVE DATA TYPE)

Primitive data type is the data type which may be used in program directly. This data type does not need any header file or any other declaration before its use. They are directly available in compiler. They are also known as basic data type. There are 4 basic data types :

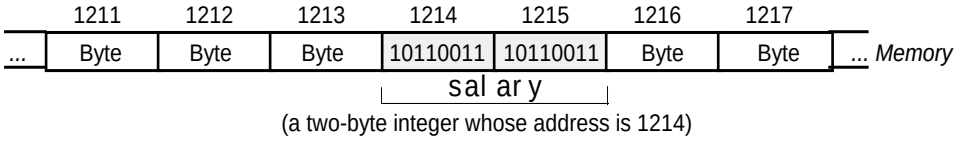
S. no.	Data type	Memory consumption	Utility	Range
1.	char	1 byte	For any single letter within single quote	-128 to 127
2.	int	2 bytes	Integer number (whole number) within range.	-32768 to +32767
3.	float	4 bytes	Real number with max 6 decimal places.	3.4 e-38 to 3.4 e+38
4.	double	8 bytes	Real number with max 14 decimal places.	1.7 e-308 to 1.7 e+308

For example, int variable definition

int salary;

This salary variable will consume 2 bytes in main memory like this-

Representation of an integer in memory.



(a two-byte integer whose address is 1214)

4.4.2 DERIVED DATA TYPES

Derived data types are those types which are derived from other types (either primitive or user defined data type). They are used to change the range of the data items. But they cannot be modified by the programmer.

S.no.	Data type	Memory consumption	Utility
1	Array	Number of items * memory consumed by 1 element	Where multiple items of homogenous types are needed to be stored. Supports static memory allocation.
2	Pointer	Number of items * memory consumed by 1 element	For Dynamic memory allocation
3	Function	Depends on return type	For modularity in program.

4.4.3 USER DEFINED TYPE

User defined data type is one type which is defined by the programmer as per the requirement of the user. It is the customized data type. In case of structure or union heterogeneous data types can be collected together to form a new data type.

Check Your Progress

What is size of void in C++?

4.5 TYPE COMPATIBILITY

Type compatibility is applied when we use two different types of data types in one operation. Type compatibility allows the use of two different types in one operation without any notification and it also allows to substitute one value for other without modification. It is very close to implicit type conversion. Type compatibility is applied in two forms: First at the assignment compatibility, second at the expression compatibility.

Example 5.1 Program to show the implementation of expression capability and assignment capability

```
#include<iostream.h>
#include<conio.h>
void main()
{
  clrscr();
  int a, b;
  a=10/3;   // expression capability
  b=10.5;  // assignment capability
  cout<<"a="<<a<<endl;
  cout<<"b="<<b<<endl;
  getch();
}
```

Output:

```
a=3
b=10
```

Value of a will be 3 because 10 and 3 are integers and their result will be integer in C++. It will be called expression compatibility.

Value of b will be 10 because b is also declared as integer; it cannot store the float value. It is called assignment compatibility.

Check your Progress

How would you declare a double called add and initialize it to 5? And which compatibility it represents?

4.6 REFERENCE

Reference variable in C++ is its new feature as compared to C language. In C language, we have to use pointer to variable for referencing. It is also possible in C++. Besides pointer to variable, C++ also supports reference variables. Reference variables are those variables which share the same memory location as assigned to them at the time of declaration.

If we change the value of reference variable, it will also update the value of that variable, which was assigned into it. And its opposite is also

true. If we change the value of original variable, it will also update the value of its reference variable.

Example 4.2 Program to show the implementation of reference variables

```
#include<iostream.h>
#include<conio.h>
void main()
{
  clrscr();
  int a,c;
  a=10;
  int &b=a;                    //reference variable to a
  c=a;
  cout<<"a="<<a<<endl;
  cout<<"b="<<b<<endl;
  cout<<"c="<<c<<endl;
  b=15;                        // update a and b
  c=20;                        // will not update a
  cout<<"a="<<a<<endl;
  cout<<"b="<<b<<endl;
  cout<<"c="<<c<<endl;
  a=19;                        // update a and b
  cout<<"b="<<b<<endl;
  getch();
}
```

Output :

```
a=10
b=10
c=10
a=15
b=15
c=20
b=19
```

Check Your Progress

Can we reinitialize reference variable?

4.7 VARIABLE

Whenever we create any program, we need to store some values. Values may be stored for input, processing or result. For all purposes, we need some memory in main memory. Programming languages provide variables to store values into them.

A variable is a named location in memory that is used to hold a value that can be modified by the program. All variables must be declared before they can be used. Variable can be declared as any valid identifier.

A variable is a symbolic name for a memory location in which data can be stored. This value may be recalled as and when necessary. Variables are used for holding data values so that they can be utilized in various computations in a program. All variables have two important attributes:

* A type which is established when the variable is defined (e.g., integer, float, character etc.). Once defined, the type of a C++ variable cannot be changed.
* A value which can be changed by assigning a new value to the variable. The kind of values a variable can assume depends on its type. For example, an integer variable can only take integer values (e.g., 2, 100, -12).

Variable Declaration : The general form of a declaration is

Datatype variable_list ;

Here, type must be a valid data type, and variable_list may consist of one or more identifier names separated by commas. In c language all the variables must be declared at the beginning of the function before any processing command.

Variable has the ability to change (vary) its value anywhere in the program. That's why it is known as variable. One variable can hold only one value at a time. On changing the variable value, the old will be replaced with the new value.

Variable Initialization : The assigning of a value to a variable for the first time is called initialization. It is important to ensure that a variable is initialized before it is used in any computation. It is possible to define a variable and initialize it at the same time. For example-

```
int a=10,b,c=5;

float x=10.25,y;
```

Types of Variables : There are four types of variables local, global variables, instance variables and class variables.

Local Variables : Local variables are declared in any specific block. Block may be any function, loop or any other block. These variables are also known as automatic variables. Scope or accessibility of such variables are within the block in which they are declared. Their lifetime or memory consumption will also be the same i.e. till the existence of that block. We cannot access local variables outside its block. There may be multiple variables with same name but in different blocks. All will consume memory separately till the existence of their block and may be accessed inside block only. A local variable is created upon entry into its block and destroyed upon exit.

Global Variables : Global variables are declared outside the main function. They are constructed at the start of the program execution and destroyed at the end. They may be accessed in the entire program. Values of global variable may be changed in any block. Once changed, they will be updated for all other locations as well. They will hold their value throughout the program's execution. If global variable and local variables are declared with same name, preference will be given to local variable to accessing block.

Example 4.3 Program to show the implementation of local and global variables

```
#include<iostream.h>
#include<conio.h>
int x;                                  //global variable
void disp1();
void disp2();
void main()
{
  clrscr();
  x=10;
  cout<<"x="<<x<<endl;     // print 10
  disp1();                 // print 10 and increase x by 1
  disp2();
  cout<<"x="<<x<<endl;     // print 11
  getch();
```

```
}
void disp1()                    // will access global variable
{
 cout<<"x="<<x<<endl;    // print value of global x
 x++;                                    // increase x by 1
}
void disp2()                    // will access local variable
{
 int x=5;                        // local variable
 cout<<"x"<<x<<endl;             // print 5
 x++;                                    // increase x by 1
}
```

Output:

```
x=10
x=10
x=5
x=11
```

Instance Variables : Instance variables are declared as member of the class. They will consume memory when object of that class is created. They are created on instance creation that's why they are known as instance variables. They cannot be accessed directly outside the class. They are like other local variables except the fact that they are created each time object of that class is created. Memory will be allocated separately for all instance variables for all objects of that class. They will be used when we will use class and objects.

Class Variables : Class variables are declared as member of the class but as static member. They will not consume memory separately for all objects of that class. They will consume memory only once irrespective of number of objects. They consume once for that class that's why they are known as class variables. They may be updated through one object and will be updated for all objects of that class as they consume only once for one class. They may be used where programmer needs common value for all objects. For example if we need to count the number of objects created for any class, we may declare one static integer data member and increase its value by one inside the constructor. They will also be used during class and objects like instance variables.

Check Your Progress

What happens if a local variable exists with the same name as the global variable you want to access?

4.8 CONSTANT

A **Constant** value is the one which does not modify throughout the execution of a program or we can say that **constant** has fixed value. **Constant** uses the secondary storage area. Constant is also called literals. Constants are of two types: Numeric and Non- Numeric Constants (Character Constant)

e.g. int x=5;

float t=3.2;

char =’p’;

4.9 TYPE CASTING

You can force an expression to be of a specific type by using a *cast*. The general form of a cast is

(type) expression

where *type* is a valid data type. For example, to cause the expression **x/2** to evaluate to type **float**, write

(float) x/2

Casts are technically operators. As an operator, a cast is unary and has the same precedence as any other unary operator. Casts can be very useful. Caste can be used for caste promotion or demotion both.

Type Conversion : Type conversion is the process in which one data type will be converted into another data type. When constants and variables of different types are mixed in an expression, they are all converted to the same type. The compiler converts all operands up to the type of the largest operand, which is called type promotion e.g.

Operand1 Type	Operand2 Type	Resultant Type
int	int	int
int	float	float
float	int	float
float	double	double
double	int	double
int	char	int

Check your Progress

What is the difference between type casting and type conversion?

4.10 OPERATOR PRECEDENCE

Operator Precedence determines which operator will be executed first in the expression or statement. Operator precedence is manipulated only if multiple operators in single expression are found. The order in which operators are evaluated in an expression is significant and is determined by precedence rules. These rules divide the C operators into a number of precedence levels. Operators in higher levels take precedence over operators in lower levels.

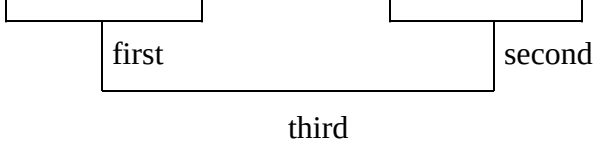
Operator Association : If multiple operators of equal precedence are found in the single expression, operator association determines in which sequence the operators will be executed. Association can be either left to right or right to left. Mostly operators are associated from left to right.

Precedence	Operator	Description	Associativity
1	::	Scope resolution	Left-to-right
2	++ --	Suffix/postfix increment and decrement	
	type() *type*{}	Function-style type cast	
	()	Function call	
	[]	Array subscripting	
	.	Element selection by reference	
	->	Element selection through pointer	
3	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	C-style type cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of	
	new, new[]	Dynamic memory allocation	
	delete, delete[]	Dynamic memory de-	

		allocation	
4	.* ->*	Pointer to member	Left-to-right
5	* / %	Multiplication, division, and remainder	
6	+ -	Addition and subtraction	
7	<< >>	Bitwise left shift and right shift	
8	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
9	== !=	For relational = and ≠ respectively	
10	&	Bitwise AND	
11	^	Bitwise XOR (exclusive or)	
12	|	Bitwise OR (inclusive or)	
13	&&	Logical AND	
14	||	Logical OR	
15	?:	Ternary conditional	Right-to-left
	=	Direct assignment (provided by default for C++ classes)	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= |=	Assignment by bitwise AND, XOR, and OR	
16	throw	Throw operator (for exceptions)	
17	,	Comma	Left-to-right

When two operators of the same priority are found in the expression, precedence is given to the extreme left operator.

Example: - x = 5 * 4 + 8 / 2;



Here, 5*4 is solved first. Through * and / have the same priorities. The operator * occurs before /.

Check Your Progress

Which operator has highest precedence in * / % ?

4.11 CONTROL STRUCTURES

A running program spends all of its time executing statements. The order in which statements are executed is called flow control (or control structures). This term reflect the fact that the currently executing statement has the control of the CPU, which when completed will be handed over (flow) to another statement. Flow control in a program is typically sequential, from one statement to the next, but may be diverted to other paths by branch statements. Flow control is an important consideration because it determines what is executed during a run and what is not, therefore affecting the overall outcome of the program.

There are 3 categories of control structures in C++

- Sequential Statement
- Selection Statement
- Iterative Statement

4.11.1 SEQUENTIAL STATEMENT

A statement, the smallest independent computational unit, specifies an action to be performed. In most cases, statements are executed in sequence. Sequential statements are those statements which are executed in linear order. These statement never change their flow of execution.

4.11.2 SELECTION STATEMENT

Selection statement applies the decision making and case control instruction in Programming. Decision and Case control instructions allow the computer to take a decision as to which instruction is to be executed next. Selection statement is used in structured programming to branch the execution of flow at run time. The branching will be done on the basis of any criteria given in the problem. This category involves if statement and switch statement. If statement is the branching statement which transfer the flow of control in the program.

Switch is the selective statement which select any case block on the basis of given value or variable. Switch can be used where programmer has multiple options but only one option can be selected by user. In selective statement specific block of code will be executed on the basis of criteria but only once. Suppose programmer needs to check

even/odd number, +ve/-ve number, Teenagers, leap year, conditional discount, Grade of student etc.; they need selective statement. C++ language must be able to perform different sets of actions depending on the circumstances. In sequence control structure, the various steps are executed sequentially, i.e. in the same order in which they appear in the program. In fact to execute the instructions sequentially, we don't have to do anything at all. By default the instructions in a program are executed sequentially. However, in serious programming situations, seldom do we want the instructions to be executed sequentially. Many a times, we want a set of instructions to be executed in one situation, and an entirely different set of instructions to be executed in another situation. This kind of situation is dealt using a decision control instruction. A decision control instruction can be implemented using:

- The **if** statement
- The **if-else** statement
- The **if-else ladder** statement
- The **nested if-else** statement
- Switch Statement

4.11.2.1 IF STATEMENT

It is sometimes desirable to make the execution of a statement dependent upon a condition being satisfied. The if statement provides a way of expressing this. C++ uses the keyword **if** to implement the decision control instruction. If the branching statement which directs the flow of execution at run time.

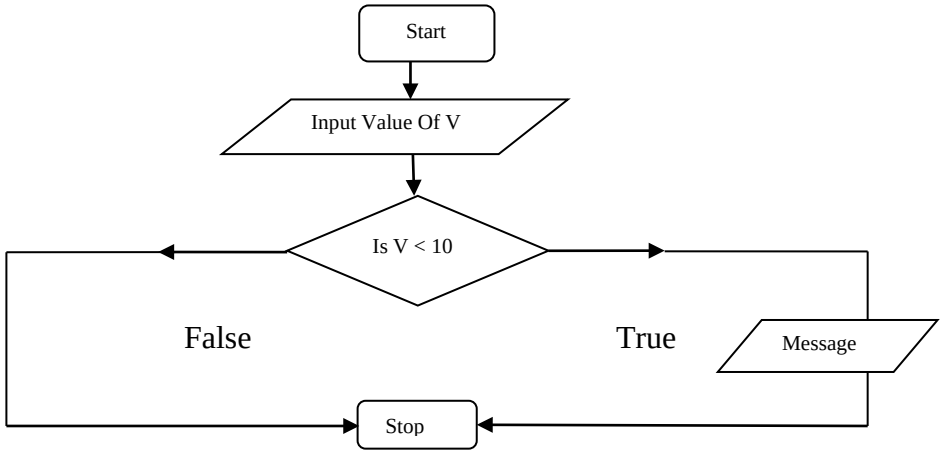
Syntax : The general syntax of if statement is as follows:

```
if (condition)
{
:
}
```

C++ uses the keyword "*if*" to execute a set of command lines or one command line when the logical condition is true. It has only one option. The set of command lines or command lines are executed only when the logical condition is true. The statement is executed only when the condition is true. In case condition is false the compiler skips the lines within the if block.

The keyword **if** tells the compiler that what follows is a decision control instruction. The condition following the keyword **if** is always enclosed within a pair of parentheses. If the condition, whatever it is, is true, then the statement is executed. If the condition is not true then the statement is not executed; instead the program skips past it. But how do

we express the condition itself? And how do we evaluate its truth or falsity? As a general rule, we express a condition using 'relational' operators. The relational operators allow us to compare two values to see whether they are equal to each other, unequal, or whether one is greater than the other.



Example 4.4 Program to convert any alphabet from lower case to upper case

```
#include<iostream.h>                    /* Header file inclusion  */
#include<conio.h>
void main()
{
char ch;                                /*  variable declaration;  */
clrscr( );
cout<<"Enter any letter ";
cin>>ch;                                /*  input statement  */
if(ch>='a' && ch<='z')                  /*  if statement to check for
      small letter*/
{
ch=ch-32;
}
cout<<"\n After conversion "<<ch;
getch( );
}
```

Output:

Enter any letter a

After conversion A

Observe that here the two statements to be executed on satisfaction of the condition have been enclosed within a pair of braces. If a pair of braces is not used then the compiler assumes that the programmer wants only the immediately next statement after the **if** to be executed on satisfaction of the condition. In other words we can say that the default scope of the **if** statement is the immediately next statement after it.

Check Your Progress

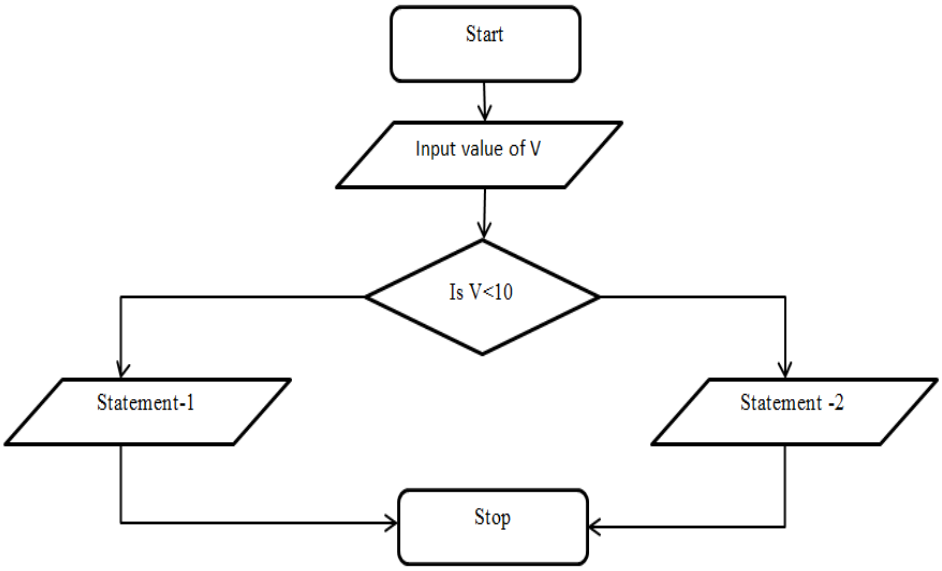
In an if statement when do you need the curly braces?

4.11.2.2 THE *IF-ELSE* STATEMENT

The **if** statement by itself will execute a single statement, or a group of statements, when the expression following **if** evaluates to true. It does nothing when the expression evaluates to false. Programmer can use the if-else statement to specify what to execute if the criteria is true and what to do when criteria is false. It means else block is optional.

Syntax : The general syntax of if-else is as follows:

```
if (condition)
 {
 :
 }
 else
 {
 :
 }
```



Example 4.5 Program of Compare two numbers .

```
#include<iostream.h>                    /* Header file inclusion  */
#include<conio.h>
void main()
{
int a,b;                               /*  variable declaration;  */
clrscr( );
cout<<"Enter two numbers ";
cin>>a>>b;                             /*  input statement  */
if(a>b)                                /*  if statement to compare
      two numbers  */
{
cout<<"\nGreater number is "<<a;
}
else
{
cout<< "\nGreater number is "<<b;
}
getch( );
}
```

Output:

Enter two numbers 23 4

Greater number is 23

Check Your Progress

The if.-else statement can be replaced by which operator?

4.11.2.3 NESTED *IF-ELSE*

If statement can be used to execute anything conditionally. So we can also use if statement within if statement. It is perfectly all right if we write an entire if-else construct within either the body of the if statement or the body of an else statement. This is called 'nesting' of ifs or nested if-

else. Nested if can be implemented to any level; but after second level it becomes very typical to debug and understand.

Syntax : The general syntax of nested if-else is as follows:

```
if
     (conditi
     on)
{
if( )
{  }
}
else
{
if( )
{  }
else
{  }
}
```

Example 4.6 Program to compare three numbers by using Nested *if-else*

```
#include<iostream.h>          /* Header file inclusion  */
#include<conio.h>
void main()
{
int a, b, c;                  /*  variable declaration;  */
clrscr( );
cout<<"Enter three numbers ";
cin>>a>>b>>c;                 /*  input statement  */
if(a>b)                       /*  if statement to compare two numbers
*/
{
if(a>c)                       /*    nested if statement to compare
remaining 2 numbers  */
    cout<<"\n The Greatest number is "<<a;
else
   cout<<"\n The Greatest number is "<<c;
}
```

```
else
{
if(b>c)                    /*  nested if statement to compare
remaining 2 numbers  */
    cout<<"\n The Greatest number is "<<b;
else
   cout<<"\n The Greatest number is "<<c;
}
getch( );
}
```

Output:

Enter three numbers 23 45 37

The Greatest number is 45

4.11.2.4 *IF-ELSE LADDER*

In this kind of statements numbers of logical conditions are checked for executing various statements. Here, if any logical condition is true the compiler executes the block followed by if condition otherwise it skip and executes else block.

If statement can be used to compare multiple conditions one after another and execute different block for different conditions. This can be done with if-else ladder. This category of if involves a range of conditions and their separate block regarding their execution.

Syntax : The general syntax of if-else ladder is as follows:

```
if
   (conditio
   n1)
{
:
}
else       if
   (conditio
   n2)
{
:
}
else       if
   (conditio
```

```
    n3)
{
:
}
else
{
:
}
```

Example 4.7 Program to compare four numbers by using if-else ladder

```
#include<iostream.h>                    /* Header file inclusion  */
#include<conio.h>
void main()
{
int a, b, c, d;                        /*  variable declaration;  */
clrscr( );
cout<<"Enter four numbers ";
cin>>a>>b>>c>>d;                       /*  input statement  */
if(a>=b && a>=c && a>=d)                /*  if statement to compare
1 number with 3  */
{
cout<<"\n The Greatest number is "<<a;
}
else if(b>=c && b>=d)                  /*  if statement to compare
1 number with 2  */
{
cout<<"\n The Greatest number is "<<b;
}
else if(c>=d)                          /*  if statement to compare
1 number with 2  */
{
cout<<"\n The Greatest number is "<<c;
 }
```

```
else
{
cout<<"\n The Greatest number is "<<d;
}
getch( );
}
```

Output:

Enter four numbers 67 78 98 34

The Greatest number is 98

Logical Operator

Logical operators are used to perform action on conditional operands. There are three types of logical operators:

1. *Logical AND operator* (&&) : operators are used to combine two conditions and return the 1 or 0. 1 represents true and 0 represents false. This is the binary operator. It returns true only if both the conditions are true otherwise it returns false.

Condition 1	Condition2	Result (condition1 && condition2)
True	True	True
True	False	False
False	True	False
False	False	False

2. *Logical OR operator* (||) : Logical or operator is also used to combine two conditions and return 1 or 0. This is also the binary operator. This operator returns true if any or both the conditions are true. If both the conditions are false only then it returns false.

Condition 1	Condition2	Result (condition1 || condition2)
True	True	True
True	False	True
False	True	True
False	False	False

3. *Logical NOT operator* (!) : Logical not operator is used to invert the condition result. This is the unary operator. If the condition is true it returns false and if the condition is false it return true.

Condition	Result (!condition)
True	False
False	True

4.11.2.5 *SWITCH STATEMENT*

The switch statement is a multi-way branch statement. If we have multiple choices and we want to choose any of the choice, we may use switch statement. But we may only compare any specific value with equality, we may not compare with any other relational operator like greater than or less than. Switch is the selective statement.

During making match against given value, if match is found in case statement, it execute the statement there onwards till break statement is found. If match is not found, it moves to next case value. At last if no case is matched, it executes default group. Default is optional in switch statement. If we want to compare one value against some values, we may use switch statement or we may use else-if ladder. But we cannot replace else-if ladder with switch statement as in switch, we cannot perform all comparison operations except equality with one value.

The Switch Case vs. Nested if

S.no.	**Switch**	**if**
1.	The switch () can only test for equality i.e. only constant values are applicable	The if can evaluate relational or logical expressions.
2.	No two case statements have identical constants in the same switch	Same conditions may be replaced for number of times.
3.	Character constants are automatically converted to integers.	Character constants are automatically converted to integers.
4.	In switch case statement nested	In nested if statement switch

	if can be used.	() case can be used.
5.	Switch statement's program can be created with if-else ladder statement.	All if programs cannot be developed with switch programs.

Syntax : The general syntax of switch is as follows :

```
switch(variable)
{
case value1:
:

        break;
case value2:
:
break;
case value3:
:
break;
default :
:
}
```

Example 4.8 Program of arithmetic operator and perform operation

```
#include<iostream.h>                    /* Header file inclusion  */
#include<conio.h>
void main()
{
int a, b, c;                           /*  variable declaration;  */
char op;
clrscr( );
cout<<"Enter two numbers ";
cin>>a>>b;                             /*  input statement  */
cout<<"Enter any arithmetic operator ";
cin>>op;                               /*  input statement  */
switch(op)                             /*  switch statement  */
{
```

```
  case '+':                                   /*  case value checking */
  c=a+b;
  cout<<"Result is "<<c;
  break;
  case '-':                                   /*  case value checking */
  c=a-b;
  cout<<"Result is "<<c;
  break;
  case '/':                                   /*  case value checking */
  c=a/b;
  cout<<"Result is "<<c;
  break;
  case '*':                                   /*  case value checking */
  c=a*b;
  cout<<"Result is "<<c;
  break;
  case '%':                                   /*  case value checking */
  c=a%b;
  cout<<"Result is "<<c;
  break;
  default:
  cout<<"Wrong operator";
  }
  getch( );
  }
```

Output:

```
Enter two numbers  34  23
Enter any arithmetic operator +
Result  is 57
```

In this example the value of op will be checked against all the case values. First of all + will be checked. If it if matched then its blocked will be executed otherwise next case value will be checked. The case matching will be checked till first match. If no case value is matched then default

block will be executed. We must use break statement before starting new case value otherwise all the statements after matching will be executed. Case value is ended with colon (:) not with semicolon.

Check Your Progress

In a switch statement what must come after every case?

4.11.3 ITERATIVE STATEMENT

Iterative is the non-sequential statement in programming. It is used to implement the reusability concept in code. It is also known looping statement. Looping execute the single or group of statement for multiple times. The number times is to be decided by programmer. It may be fixed number of times or variable number of times. In some cases it may also be infinite loop. But important thing is that single or group will be executed repeatedly. So we can also say that iterative statement involves the repetitive statement. If we write some statements inside the iterative block that will be executed multiple times continuously till any specific condition. Iterative statement defines a specific block which defines the number of repetition. For example display counting of numbers, table of any number, factorial etc. can be solved with iterative statement.

A loop is defined as a block of statements which are repeatedly executed for certain number of times. The condition may be predetermined or open ended.

Parts of the loop :

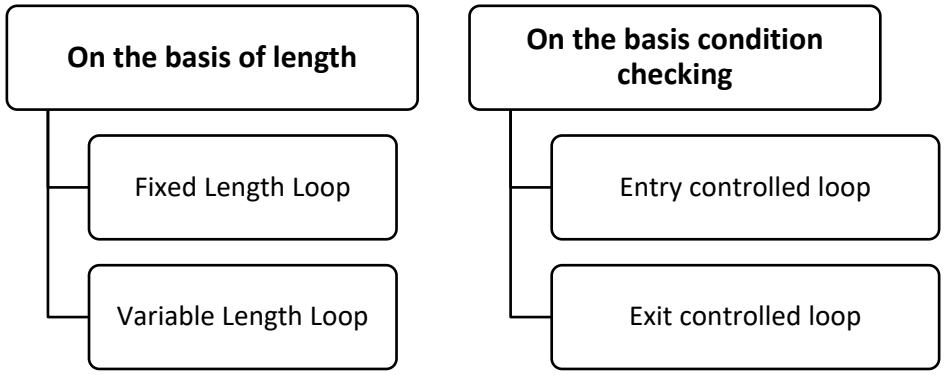
1. *Initialization***:** Initialization is the first step involved in loop. It specifies the starting value of the counter variable of loop. It is executed only once at the start of loop.

2. *Ending Condition:* Ending condition specifies the terminating condition for loop. Loop is executed till the condition is true. As soon as the condition gets false, loop stops. Condition is checked every time the loop is executed.

3. *Step value:* Step value defines the difference between two terms of the loop counter. If loop starts from smaller starting value to larger value, increment is applied. If loop starts from ending value to smaller value, decrement is applied. Increment or decrement is applied from the second time and then each time.

4. *Loop Body:* Loop body can be any single or a group of statements which executes till condition is true. It contains the actual statements which are required to be executed repeatedly.

Categories of looping



Fixed Length Loop :

It is also known as counter controlled loop. Fixed Length loop is used where number of iteration is known at the time of looping. For loop is the type of fixed loop. In this loop initial value of counter, ending condition and step value are known to programmer. For example :

Requirement	***Initial value***	***Ending condition***	***Step value***
Natural numbers	1	N	+1
Natural number in reverse order	N	1	-1
Even numbers	2	N	+2
Odd Numbers	1	N	+2

In the above table N is the integer variable, where value is input by user.

Variable length Loop :

Variable length loop is a kind of loop which is executed till specified condition is true. Sometimes programmer doesn't know the number of iterations then variable length loop is used. Its syntax does not include initialization and step value. It is variable length loop because with the help of loop we cannot specify how many times loop will execute. While and do-while is the type of variable length loop. For example to find the LCM, HCF, Binary search, sum of digits etc. we have to use variable length loop.

Fixed Length vs. Variable length Loop

S.no.	Fixed Length Loop	Variable length Loop
1.	In fixed length loop numbers of iterations are known.	In variable length loop numbers of iterations are not known.
2.	In this loop syntax contains initialization, condition and step value.	In this loop syntax contains only condition.
3.	For loop is fixed length loop.	While and do-while loop are the variable length loops.

Entry controlled Loop :

It is also known as top tested loop. Entry controlled loop is the loop where terminating condition is tested from the start of looping. For and while loops are the top tested loop.

Exit controlled Loop :

It is also known as Bottom tested loop. Exit controlled loop is the loop where terminating condition is tested at the bottom of looping. Do-while loop is the bottom tested loop. Exit controlled loop will execute one time even condition is false on first time. So we can say that when we need loop to be execute at least one time irrespective of the condition, we have to use do-while loop. If the condition gets false on the second time there is no difference between entry controlled and exit controlled loop.

Entry controlled Loop vs. Exit controlled Loop

S.no	Entry controlled Loop	Exit controlled Loop
1.	Condition is checked at the top of loop.	Condition is checked at the bottom of loop.
2.	It is used to execute till the condition is true.	It is used to execute till the condition is true but at least one time irrespective of the condition.
3.	If the condition gets false on first time then loop will not execute.	If the condition gets false first time, loop will execute one time.
4.	For loop and while are top length loop.	Do-while loop is the variable length loops.

Types of Loop :

The C++ language supports three types of loop control statements.

a. The *for* loop

b. The *while* loop

c. The *do-while* loop

4.11.3.1 THE *FOR* LOOP

For loop is the fixed length and entry controlled loop. Whenever programmer know the number of iterations for loop is used. The *for* loop allows to execute a set of instructions until a certain condition is satisfied.

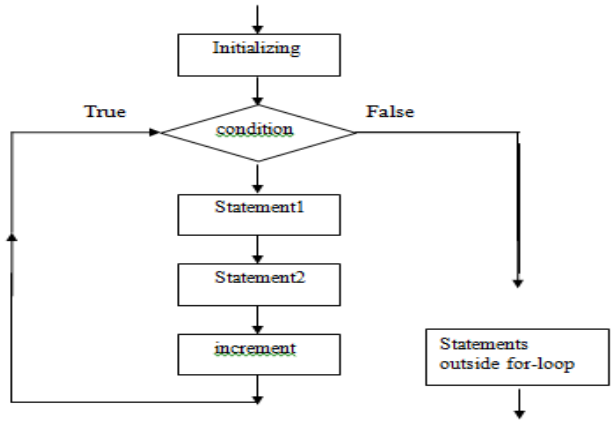
Syntax : The general syntax of for loop is as follows:

```
for (initialization ; test condition ; increment/decrement)
{
    Statement1;
    Statement2;
}
```

The for loop is the simplest and most commonly used loop. The loop consists of three expressions. The first expression is used to initialize the counter, the second is used to specify the terminating condition of loop. Loop will be executed till condition is true. Increment/decrement will specify the step value of counter. You must separate these three major sections by semicolons. The for loop is more powerful when compared to other loops. The number of iterations required to execute the body of the loop is compared by using the formula.

Number of iteration = (Final Value(FV) - Initial Value(IV) + Step Increment(SI)) / (Step Increment(SI))

Flowchart of *for*-loop



for Loop Variations

The previous discussion described the most common form of the for loop. However, several variations of the for are allowed that increase its power, flexibility, and applicability to certain programming situations. One of the most common variations uses the comma operator to allow two or more variables to control the loop.

The Infinite Loop

Although you can use any loop statement to create an infinite loop, **for** is traditionally used for this purpose. Since none of the three expressions that form the **for** loop are required, you can make an endless loop by leaving the conditional expression empty, as here:

```
for( ; ; ) cout<<"This loop will run forever.\n";
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but programmers more commonly use the **for(;;)** construct to signify an infinite loop. Actually, the **for(;;)** construct does not guarantee an infinite loop because a **break** statement, encountered anywhere inside the body of a loop, causes immediate termination. Program control then resumes at the code following the loop, as shown here:

```
ch = '\0';
for( ; ; )
{
        ch = getchar();                 /* get a character */
        if(ch == 'A')
```

```
            break;              /* exit the loop */
    }
    cout<<"you typed an A";
```

This loop will run until the user types an **A** at the keyboard.

for Loops with No Bodies

A statement may be empty. This means that the body of the **for** loop (or any other loop) may also be empty. You can use this fact to simplify the coding of certain algorithms and to create time delay loops

```
for( ; *str == ' '; str++) ;
```

As you can see, this loop has no body— and no need for one either.

Time delay loops

The following code shows how to create one by using **for**:

```
for(t=0; t < SOME_VALUE; t++) ;
```

The only purpose of this loop is to eat up time.

Syntax	Output	Remarks
for(a=0; a<=10; a++) printf("%d", a);	Display value from 1 to 10	'a' is increased from 0 to 10. Curly braces are not necessary. Default scope of for loop is one statement after loop.
for(a=10; a>=0; a--) cout<< a;	Display value from 10 to 0	'a' is decreased from 10 to 0.
for(; ;) cout<< a;	Infinite loop	No terminating statement
for(a=0; a<=20;) cout<< a;	Infinite loop	'a' is neither increased nor decreased.

Check Your Progress

The C code 'for (; ;) ' represents an infinite loop. It can be terminated by _____

Example 4.9 Program to Print natural numbers up to n

```
#include<iostream.h>                    /* Header file inclusion  */
#include<conio.h>
void main()
{
int i, n;                              /*  variable declaration;  */
clrscr( );
cout<<"Enter limit ";
cin>>n;                                /*  input statement  */
for(i=0; i<=n; i++)
{
    cout<<"\t"<<i;
}
getch( );
}
```

Output:

```
Enter limit 7
0   1   2   3   4   5  6    7
```

Check Your Progress

Can a for loop contain another for loop?

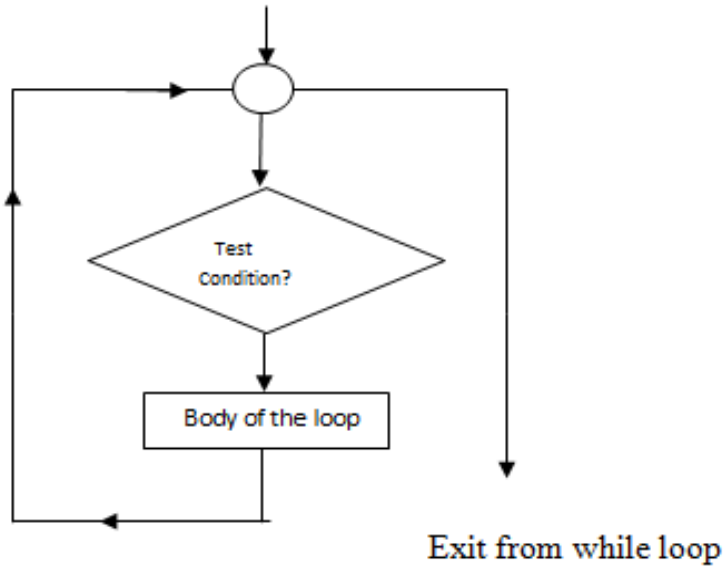
4.11.3.2 THE *WHILE* LOOP

While is the variable length and entry controlled loop. This loop contains a condition but not initialization and increment/decrement. The condition may be any expression, and true is any nonzero value. The loop iterates while the condition is true. When the condition becomes false, program control passes to the line of code immediately following the loop.

Syntax: The general syntax of while loop is as follows:

```
while (test condition)
{
      :
      :
}
```

The test condition may be any expression. The loop statements will be executed till the condition is true i.e. the test condition is evaluated and if the condition is true, then the body of the loop is executed. When the condition becomes false the execution will be out of the loop.



Example 4.10 Program to Print natural numbers up to n

```
#include<iostream.h>                    /* Header file inclusion  */
#include<conio.h>
void main()
{
int i, n;                               /*  variable declaration;  */
clrscr( );
cout<<"Enter limit ";
```

```
cin>>n;                                  /*  input statement  */
i=0;                                     /*  input statement  */
while(i<=n)
{
  cout<<"\t"<<i;
  i++;
}
getch( );
}
```

Output:

Enter limit 7

0 1 2 3 4 5 6 7

Check Your Progress

How many times ‘ its a while loop' should be printed?

```
int main()
{
int i = 1 ;
i = i - 1 ;
while(i)
{   cout<<"its a while loop";
   i++ ;
}
return 0;
}
```

4.11.3.3 THE *DO-WHILE* LOOP

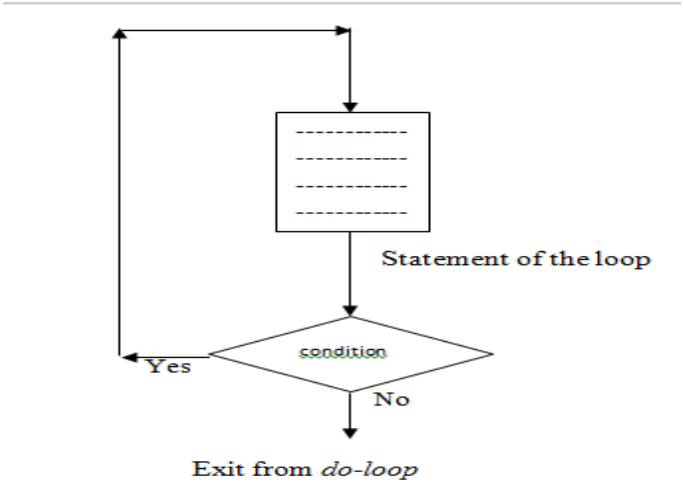
Do-while is the variable length and exit controlled loop.

Unlike for and while loops, which test the loop condition at the top of the loop, the do-while loop checks its condition at the bottom of the loop. This means that a do-while loop always executes at least once.

Syntax: The general syntax of do-while loop is as follows:

```
do
{
        :
        :
}while (test condition);
```

Do while loop is ended with terminator(;) after condition because it treats the block as a single statement starts from do.



Example 4.11 Program to Print natural numbers up to n

```
#include<iostream.h>                    /* Header file inclusion  */
#include<conio.h>
void main()
{
int i, n;                               /*  variable declaration;  */
clrscr( );
cout<<"Enter limit ";
cin>>n;                                 /*  input statement  */
i=0;                                    /*  input statement  */
do {
   cout<<"\t"<<i;
```

```
    i++;
  } while(i<=n)
  getch( );
  }
```

Output:

Enter limit 7

0 1 2 3 4 5 6 7

In this program if user input 10 in limit, loop will print natural numbers from 1 to 10. If user input 0 in range, even then it will print 1 because it is the exit controlled loop.

Perhaps the most common use of the **do-while** loop is in a menu selection function. When the user enters a valid response, it is returned as the value of the function. Invalid responses cause a return to next life after the condition. Here, the **do-while** loop is a good choice because you will always want a menu function to execute at least once. After the options have been displayed, the program will loop until a valid option is selected.

Check Your Progress

What loops will always execute at least once?

For Loop vs. While Loop

S.no.	For Loop	While Loop
1.	For loop is the fixed loop.	While loop is the variable loop.
2.	For loop syntax contains initialization, condition and increment/decrement.	While loop syntax contains only condition.
3.	Terminator(;) is used to separate 3 parts.	Only condition is used so separation is not required.
4.	Multiple initialization and increments can be used.	While loop syntax does not include initialization and increment.
5.	for keyword is used.	while keyword is used.
6.	Syntax-	Syntax-

	for (initialization ; condition ; increment)
	{
	:
	}
		while (test condition)
		{
		:
		}

While Loop vs. do-while Loop

S.no	While Loop	Do-while Loop
1.	While loop is the entry controlled loop.	Do-While loop is the exit controlled variable loop.
2.	While loop is executed till condition is true.	Do-while loop is executed till condition is true but at least one time.
3.	While loop is not ended with terminator (;).	While loop is ended with terminator (;).
4.	while keyword is used.	Do and while keywords are used.
5.	Generally used when number of iterations are not known and loop needs to be executed till condition is true.	Generally used in menu based programs where loop needs to be executed at least 1 time even condition is false very first time.

6.	Syntax-
	:
	while (test condition)
	{
	:
	}
		Syntax-
		:
		do
		{
		:
		} while (test condition);

The break statement is used to terminate a case in a switch construct. The break statement is also for termination of a loop, bypassing the normal loop conditional test.

When a break is encountered inside a loop, the loop is terminated and control passes to the statement following the loop body.

Example 4.12 Program to find the inverse of a number

```
#include<iostream.h>
#include<conio.h>
void main()
{
float n;
char reply;
clrscr();
do
{
cout<<"Enter a number:";
cin>>n;
if(n == 0)
break;
cout<<"Inverse of the number is "<<1/n<<endl;
cout<<"Do you want to input another number(y/n)?";
cin>>reply;
}
while(reply != 'n');
getch();
}
```

Output:

```
Enter a number : 2
Inverse of the number is 0.5
Do you want to input another number (y/n) ?y
Enter a number:67
Inverse of the number is 0.014925
Do you want to input another number(y/n)? n
```

The continue statement forces the next iteration of the loop to take place, skipping any code following the continue statement in the loop body. In the for loop, the continue statement causes the conditional test and then the re-initialization portion of the loop to be executed. In the while and do.

While loops, program control passes to the conditional test. The following program illustrates the usage of the continue statement:

Example 4.13 Program to finds the square of the numbers less than 100

```
#include<iostream.h>
#include<conio.h>
void main()
{
int n;
char reply = 'y';
clrscr();
do
{
cout<<"Enter a number:";
cin>>n;
if(n > 100)
{
cout<<"The number is greater than 100, enter another"<<endl;
continue;
}
cout<<"The square of the number is: "<<n*n<<endl;
cout<<"Do you want to enter another(y/n)? ";
cin>>reply;
}
while(reply != 'n');
getch();
}
```

Output:

```
Enter a number : 123
The number is greater than 100, enter another
Enter a number : 34
The square of the number is: 1156
Do you want to enter another (y/n) ?y
```

```
Enter a number :7
The square of the number is: 49
Do you want to enter another (y/n) ?n
```

Check Your Progress

Which keyword is used to come out of a loop only for that iteration?

4.12 STRUCTURE

Structure is the user defined data type. It is the collection of multiple members. Data type of all members may be similar or different. All the members declared inside the structure will consume memory when variable of that structure will be created. Size of the structure variable depends on the members declared in the structure.

Syntax : The general syntax of structure is as follows:

```
 struct structure_name{

member_1_declaration;

member_2_declaration;

member_3_declaration;

member_n_declaration;
      }variable_list;
```

Unlike array, data type of all elements in structure, are not mandatorily to be same but they will consume memory sequential order. For example if we declare the structure for storing the record of any student, that needs to store his name, roll number and age. It will be like this

Example :

```
 struct student
{
    int rollno;
```

```
    char sname[50];
    int age;
};
```

Variable for this structure variable will be declared like:

student s;

Memory representation of this variable be represented like:

			rollno	**sname**	**age**									
			2 bytes	50 bytes	2 bytes									

So each variable of this student structure will consume 54 bytes.

Members will be accessed via dot (.) operator through variable name. for example in the above case, we may access the members like:

s.rollno

s.sname

s.age

Example 4.14 Program to store and display the record of student with structure.

```
#include<iostream.h>
#include<conio.h>
struct student
{
  int rollno;
  char sname[50];
  int age;
};
void main()
{
```

```
    clrscr();
    student s;
    cout<<"Enter student name ";
    cin>>s.name;
    cout<<"Enter student roll number ";
    cin>>s.rollno;
    cout<<"Enter student age ";
    cin>>s.age;
    cout<<"\n\t\tStudent Record \n";
    cout<<"\n\t NAME : "<<s.sname;
    cout<<"\n\t ROLL NUMBER : "<<s.rollno;
    cout<<"\n\t AGE : "<<s.age;
    getch();
  }
```

Output:

```
Enter student name  Amit
Enter student roll number  101
Enter student age 17

                    Student Record


              NAME  :  Amit
              ROLL NUMBER : 101
              AGE :   17
```

Check Your Progress

What will happen when the structure is declared?

Unions

Like structure of C language, C++ also support unions. Union is also one of the user defined data type. It may contain multiple members of different data types. But union variable may store any one member's value at a time. If we try to store value in any other member then value of pervious member will be overwritten.

Syntax : The general syntax of Union is as follows:

```
union union_name{

  member_1_declaration;

  member_2_declaration;

  member_3_declaration;

  member_n_declaration;
      }variable_list;
```

Unlike array, data type of all elements in union, are not mandatorily to be same but they will consume in different way. Memory will depends on the member which consumes largest number of bytes. For example if we declare the union for storing the record of any student, that needs to store his name, roll number and age. It will be like this

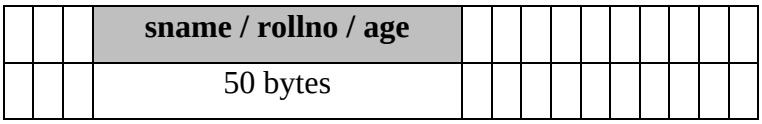
Example :

```
union student
{
    int rollno;
    char sname[50];
    int age;
};
```

Variable for this union variable will be declared like:

student s;

Memory representation of this variable be represented like:



So each variable of this student structure will consume 50 bytes because name consumes maximum number of bytes. If we store any member, it

will be stored in that location. If any new member is updated, it will be updated in the same location.

Members will be accessed via dot (.) operator through variable name. For example in the above case, we may access the members like:

s.rollno

s.sname

s.age

Example 4.15 Program to display the multiple student's records through unions.

```
#include<iostream.h>
#include<conio.h>
union student
{
  int rollno;
  char sname[50];
  int age;
};
void main()
{
 clrscr();
 student s1,s2,s3;
 cout<<"Enter name of 1st student ";
 cin>>s1.sname;
 cout<<"Enter roll number of 2nd student ";
 cin>>s2.rollno;
 cout<<"Enter age of 3rd student ";
 cin>>s3.age;
 cout<<"\n\t\tStudent's Record \n";
 cout<<"\n\t NAME : "<<s1.sname;
 cout<<"\n\t ROLL NUMBER : "<<s2.rollno;
 cout<<"\n\t AGE : "<<s3.age;
```

```
    cout<<"\n\nEnter roll number of first student ";
    cin>>s1.rollno;
    cout<<"\n\n\t After Input all values of first student";
    cout<<"\n\t NAME : "<<s1.sname;
    cout<<"\n\t ROLL NUMBER : "<<s1.rollno;
    cout<<"\n\t AGE : "<<s1.age;
    getch();
  }
```

Output:

```
Enter name of  1st student  Amit
Enter roll number of 2nd student 102
Enter age of 3rd student  17
                    Student's  Record
          NAME  :  Rahul
          ROLL NUMBER : 101
          AGE :   17

Enter roll number of first student 101

          After Input all values of first student
          NAME : e
          ROLL NUMBER: 101
          AGE: 101
```

Check Your Progress

What will be used when terminating a union?

4.13 FUNCTIONS

C++ supports sequential, selection and iteration in statements. For reusability in code, we use looping. But looping is used for reusability

only when code needs to be executed multiple times but in continuously. If we need to reuse the same code any different locations, looping cannot be used. We need some reusable modules which may be called at different locations. At such situation, we use functions.

"A function is defined as program segment that carries out some specific, well defined task."

A function is a group of statements for reusability purpose. There may be multiple functions in one program. Functions needs to be defined only once and they may be called any number of times at anywhere. There is always a specific purpose for function. One function is meant for one purpose. For different purposes, we need to create multiple functions. A function receives zero or more parameters, performs a specific task, and returns zero or one value. Every C++ program has at least one function, which is main. Sometimes function is also known as method or sub-routine or procedure or module etc.

Need of Functions

There are several reasons for which we need functions:

- Complex Problems
- Lengthy Projects
- Time Management
- Debugging Problems

Benefits of Functions :

- Simplicity : reduced complexity in program
- Reusability : Avoid code repetition
- Easy Debugging
- Time management : project completion in time.

Modularity is the solution to the above problems. Via functions, complex problems may be divided into small and simple modules. It also reduces the length of code through reusability. A lengthy and complex problem may be divided into team members for early solutions. Functions also makes the debugging easy because programmer knows the segments where error may arise.

Function Types :

- Library Functions: stored in compiler library
- User Defined Functions: Developed by programmer

Category of function :

- Parameterized with returning value

- Parameterized without returning value
- Non parameterized with returning value
- Non parameterized without returning value

Parts of FUNCTION :

- Function Prototype: Declaration
- Function Definition: Function body
- Function Calling: Revoking Statement

Type	Prototype	Definition	Calling
Library Functions	Pre-declared in header files	Predefined in library files	Call by programmer
User defined functions	Have to be declared by programmer	Have to be defined by programmer	Call by programmer

Function Prototype (Declaration) :

A function declaration tells the compiler about a function's name, return type, and parameters. It tells the compiler how to call the function.

ReturnType FunctionName (Parameterlist);

where	meaning	default value
ReturnType	Function type. (At most one type in counting)	int
FunctionName	Name of the function.	Any valid identifier is mandatory
Parameterlist	Values to be passed (Can be zero or multiple in counting)	void

Example :

```
union student
 {
```

```
    int rollno;
    char sname[50];
    int age;
};
```

Parameter names are not mandatory in function declaration, following is also valid:

```
void sum(int , int );
```

Function Definition :

A function definition provides the actual body of the function. The general form of a function definition in C++ programming language is as follows.

Syntax: The general syntax of Function is as follows :

```
return_type Function_Name ( parameter list )
{
  statement(s);
   …….
   …….
   return (value);  //if required
}
```

A function definition consists of a *function header* and a *function body*. Here are all the parts of a function:

- **Return Type**: A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void. If no return type is specified, it will be considered as int by compiler in C++.
- **Function Name:** This is the actual name of function. The function name and the parameter list together constitute the function signature. It must be any valid identifier.
- **Parameters:** It is also known as argument. Sometimes function needs some value for performing operations, in that case we may send some values into the function while calling. We may call the function multiple times with different values but number and type of arguments must be same. There are two types of parameters:

- **Formal parameters:** At the definition of function parameters are known as formal parameters. We need to specify the formal parameter names with data type.
- **Actual parameters:** At the calling of function, parameters are known as actual parameter. We need to specify the parameters without datatype. Actual parameters may be values, variables or any valid expression.

A parameter is like a placeholder. When a function is called, values are passed from actual parameters to formal parameters in same sequence. Parameters are optional; that is, a function may contain no parameters.

- **Function Body:** The function body is the actual definition of the function. It will be executed whenever function is called. Function may or may not any value after execution. But there may be at most one value while returning. We need to define the function body once irrespective of the number of function calling.

Example:

```
/* function definition to add two numbers */
int sum(int x, int y)
{
   int z;
   z=x+y;
   return z;
}
```

Calling a Function :

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

[variable=] FunctionName (Parameterlist);

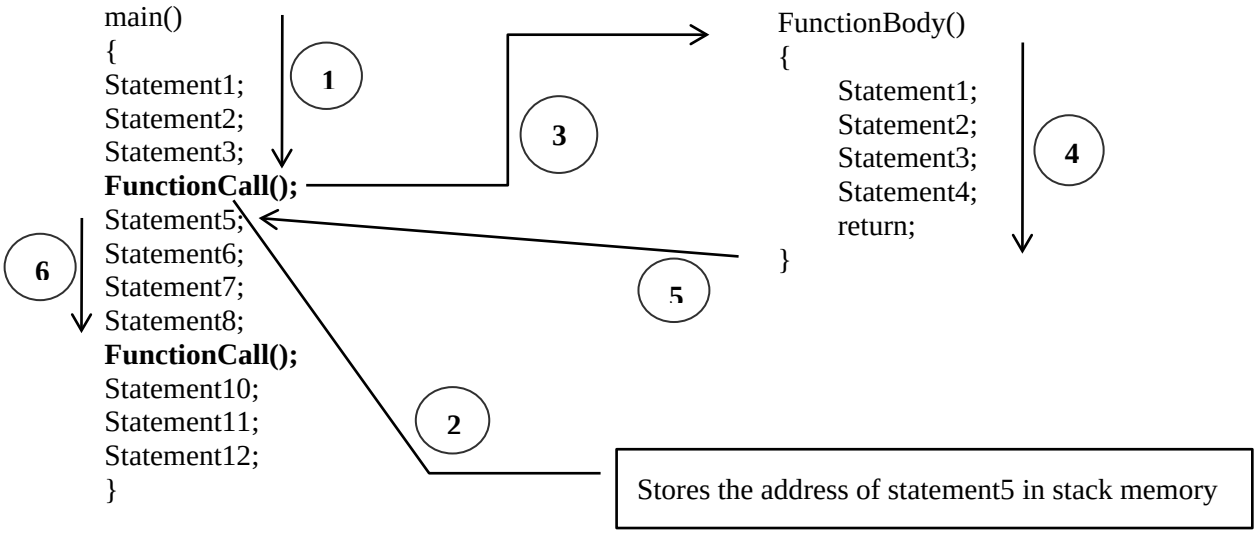
where

- *Variable* is used to store the result (if any). It is optional, depending the function.
- *FunctionName* is the name of the function as in the declaration.
- *Parameterlist* is the list of values to be passed. Here parameters are also actual parameters. Number and type of parameters depends on function prototype.

Example:

```
display( );          /* non parameterized function without returning value*/
sum(a, b);           /* parameterized function without returning value*/
f=factorial(5);      /* parameterized function with returning value*/
p=pi();              /* parameterized function with returning value*/
```

Execution of Function:



1. When we execute C++ program, normal execution is started.
2. When a program calls a function, first of all address of the next statement to function call is stored to stack memory. This address is stored so that we may return to that statement once function is executed.
3. Program control is transferred to the called function.
4. A called function performs defined task
5. When function execution is completed it returns program control back to the calling section from where it was called. That address will be received from stack memory where address was already stored before calling.
6. Once it is returned, normal course of action is performed again.

There may be multiple function call and they will be executed in the same way. Function may be called from main function or it may be called from any other function. Even in that case, execution will remain same and it will be returned to its calling section.

Example 4.16 Program to find the larger of two number with function

```
#include <iostream.h>
#include<conio.h>
 /* function declaration */
int max(int x, int y);
void main ()
{
   /* local variable definition */
   int a,b,large;
   clrscr();
   cout<<"Enter two numbers ";
   cin>>a>>b;
    /* calling a function to get max value */
   large = max(a,b);
   cout<<"\n Max value is :"<<large ;
   getch();
}
 /* function returning the max between two numbers */
int max(int x, int y)
{
   /* local variable declaration */
   int result;
    if (x > y)
      result = x;
   else
      result = y;
    return result;
}
```

Output:

```
Enter two numbers  23 45
Max value is 45
```

Function calling types :

Function may be called in two ways depending on the values/ address sent as parameter.

Call Type	Description
Call by value	When values of actual parameters are sent to formal parameters, it is called function call by value. In this case, changes in formal parameters is not reflected back into actual parameters.
Call by reference	When reference of actual parameters are sent to formal parameters rather than its value, it is called function call by reference. In this case, changes in formal parameters are reflected back into actual parameters. This is achieved via array, pointers and reference variables.

Suppose we need to swap the values of two variables, in that case, after swapping via function, we needs updated values back in calling section. So we need function call by reference in that case.

Example 4.17 Program to swap of two variable without calling be reference

```
#include<iostream.h>
#include<conio.h>
// function declaration
void swap(int x, int y);
void main ()
{
  // local variable declaration:
  int a, b;
  clrscr();
```

```
    cout<<"Enter two numbers ";
    cin>>a>>b;
    cout << "\n Before swap, value of a :" <<a;
    cout << "\n Before swap, value of b :" <<b ;
    // calling a function to swap the values.
    swap(a, b);
    cout << "\n After swap, value of a :" << a;
    cout << "\n After swap, value of b :" << b;
    getch();
  }

  void swap(int x, int y)
  {
    int temp;
    temp=x;
    x=y;
    y=temp;
    cout<<"\n During Swapping x="<<x;
    cout<<"\n During Swapping y="<<y;
  }
```

Output :

```
Enter two number 23 45
Before swap , value of a  : 23
Before swap , value of b  : 45
During swapping x= 45
During swapping y= 23
After swap, value of a : 23
After swap, value of b :45
```

In the above program code, swapping will be temporarily as it is not reflected back into calling section.

So we need to solve it via function call by reference.

Example 4.18 Program to Swap of two variable. (with calling by reference)

```
#include<iostream.h>
#include<conio.h>
// function declaration
void swap(int &x, int &y);
void main ()
{
  // local variable declaration:
  int a, b;
  clrscr();
  cout<<"Enter two numbers ";
  cin>>a>>b;
  cout << "\n Before swap, value of a :" <<a;
  cout << "\n Before swap, value of b :" <<b ;
  // calling a function to swap the values.
  swap(a, b);
  cout << "\n After swap, value of a :" << a;
  cout << "\n After swap, value of b :" << b;
  getch();
}

void swap(int &x, int &y)
{
 int temp;
 temp=x;
 x=y;
 y=temp;
 cout<<"\n During Swapping x="<<x;
 cout<<"\n during Swapping y="<<y;
```

```
}
```

Output:

```
Enter two number 23 45
Before swap , value of a  : 23
Before swap , value of b  : 45
During swapping x= 45
During swapping y= 23
After swap, value of a : 23
After swap, value of b :45
```

Above problem is solved via reference variables. This feature was not available in C language. In C language, we have to solve it with pointers. In C++, we may call the function by reference with pointers as well as with reference variables. If we pass the arrays as parameter, it will also be called by reference because array name is actually a pointer to first element of array.

Check Your Progress

Which is the default return value of functions in C++?

4.14 SUMMARY

C++ is one of the object oriented programming language. It is truly object oriented programming language as it supports all features of object orientation like: abstraction, encapsulation, data hiding, inheritance, polymorphism, message passing, class and objects.

C++ supports approximately all features of C language. It supports if statements with all its variants, all loop with their variants, array, pointer, structure, unions and functions etc.

Class is the collection of data members and member function. Class is a kind of template for customised user-defined data types. Object is known as the instance of class.

Functions are applicable in the same manner as in C language. Functions are the subprograms to solve any complicated and lengthy problems.

4.15 EXERCISE

Q1. Explain the features of OOPs in C++.

Q2. What is type casting? How it is achieved in C++. Explain with example.

Q3. What is function? Describe its types and how they are called.

Q4. What is control structure in C++? Describe their types with example.

Q5. Write a program in C++ to create a user defined function to calculate the factorial of any number.

Q6. Write a program in C++ to check whether any number is prime number or not.

Q7. Write a program in C++ to find the reverse of any number.

Q8. Write a program in C++ to create the structure to store two employee records and compare them on the basis of their salary. Display the record of employee who earns more salary. Employee records should store his name, address, employee ID and salary.

Q9. Write a program in C++ to display the Fibonacci series via function.

Q10. Write a program in C++ to swap the values of two variables without third variable.

UNIT-5 CLASS & OBJECTS

Structure

5.1 INTRODUCTION

Earlier programming was restricted to procedure oriented programming but it was sufficient to solve the all real world problems. Because many problems were dynamic in nature so we need a different approach. In this approach, our major requirement is to focus more on data as compared to procedure because procedure will be different in different scenarios. That's why object oriented approach was developed.

In this approach we use objects which may be any instance. For example any chair, table, student, school, hospital etc. may be an object. There may be multiple objects of one category. But each object of one category has some common fields for example if we have multiple students, all of them will have name, roll number, age etc. And their

values may be different. So an object can be considered a "thing" that can perform a set of activities. The set of activities that the object performs defines the object's behaviour.

Object-oriented programming attempts to provide a model for programming based on objects. Object-oriented programming integrates code and data using the concept of an "object". In object oriented data and functions are not freely moved as they were moving in procedure oriented programming. We need this kind of restriction so that we may run a complicated and lengthy problem. Besides movement, data and functions may also not be directly accessible anywhere. Addition to this, one concept may also be reused at other places to save the time.

Here are some features of object-oriented programming approach: *abstraction, encapsulation, data hiding, polymorphism* and *inheritance*.

Abstraction : Abstraction is the feature to hide the programming code to outside world and reveals its required information. Abstraction saves us form dealing with complicated and lengthy internal details. We just need to know the essential part for implementation.

For example if we want to implement the banking system, then customer only needs to know about deposit and withdrawal forms, rest internal details are of no use for him. Internal processing will be there but the customer only need to know the interfacing with forms. In the same way, in programming if we are implementing any problem, then we only need to know how to implement the pre solved things.

In simpler way, we can say abstraction hides the complicated details and reveals the essential information only.

Encapsulation : Encapsulation is the feature to wrap up the data members and methods into a single unit. Wrapping the members in a single unit restrict it to be accessed directly outside that unit. That unit is known as class. Encapsulation restricts the free flow of the data across the program. It also ensures that outside data to the class may also not be accessed inside the class. So we may also say that encapsulation creates a layer between inside class members and outside class members. Whenever we create the object of that class, that object may access only those members. It means, data of one object will be different from data of other object. One object cannot update the data of other object directly. If we need to update the data of one object, we must have one method for that in the class.

Data Hiding : Data hiding restricts the data member accessing via object / inheritance. It is used to implement the data security in class. There are three visibility modes for data accessing- public, private and protected. Public mode is used to access the members via object and inheritance. Private mode is used to restrict the member access outside the class. Protected mode is used to restrict the member accessing only via inheritance. So private visibility mode hides the data to be accessed via

object. Without object / inheritance any member cannot be accessed outside the class. Members may be accessed inside the class directly.

Inheritance : Inheritance is the ability to reuse the members of one class into other class without redefining them. The class which is to be accessed is called super class or base class. The class in which members will be accessed is known as sub class or derived class. In C++, we may inherit more than one class into any class.

Polymorphism : Polymorphism is the feature which enables any object to act differently in different scenarios. For example, there is a boy, at home- he is son, at school- he is student, at sports ground- he is sportsman but he is the same boy. He will act differently at different scenarios. Polymorphism is again very important feature of object oriented approach as we are more focused on real world problems and real world problems are dynamic in nature.

Some of the **advantages** of object-oriented programming include:

1. Improved software-development productivity.
2. Improved software maintainability.
3. Faster development.
4. Lower cost of development.
5. Higher-quality software.

Check Your Progress

Which concept allows you to reuse the written code?

5.2 OBJECTIVES

After studying this unit you should be able to identify and describe class and objects, Characteristics of OOP's, brief introduction of pointers and their use with class and function**.**

5.3 CLASS SPECIFICATION

The building block of C++ that leads to Object Oriented programming is a **Class**. It is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

For Example: Consider the Class of **Cars**. There may be many cars with different names and brand but all of them will share some common properties like all of them will have number of wheels, Speed

Limit, Mileage range etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

- A Class is a user defined data-type which has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.
- In the above example of class *Car*, the data member will be *speed limit*, *mileage* etc and member functions can be *apply brakes*, *increase speed* etc.

An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Defining Class and Objects

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

Syntax : The general syntax of Class is as follows:

```
class <CLASSNAME>
{
        <ACCESSSPECIFIER>:      // may be public/private/protected
                <Data Members List>;
                <Members Function List>;

        <ACCESSSPECIFIER>:      // may be public/private/protected
                <Data Members List>;
                <Members Function List>;

        <ACCESSSPECIFIER>:      // may be public/private/protected
                <Data Members List>;
                <Members Function List>;
```

```
        } ;
```

For Example : if we need to store the record of student with his name, roll number and average marks. We may define the class like this

```
class student

{

   private:

        int rollno;

        char sname[20];

        float avg;

   public:

        void input();

        void display();

};
```

Declaring Objects : When we define a class, we define only it's specification for the object. It does not consume any memory at runtime. So we cannot store any value in class. Whenever we need to store any record for that class, we need to define the object of that class. We may also call their methods, once object is defined.

Syntax :

```
ClassName ObjectName1, ObjectName2, ObjectName3;
```

For example:

student s1, s2, s3;

Accessing data members and member functions: The data members and member functions of class can be accessed using the dot('.') operator with the object. For example if the name of object is *s1* and we want to access the member function, we will use this syntax-

For example:

```
s1.sname        // not possible outside the class as sname is private
s1.rollno       // not possible outside the class as rollno is private
s1.avg          // not possible outside the class as avg is private
s1.input()      // possible as input() is public
s1.display()    // possible as display() is public
```

Check your Progress

The variables declared inside the class are known as data members and functions are known as …….

5.4 CLASS

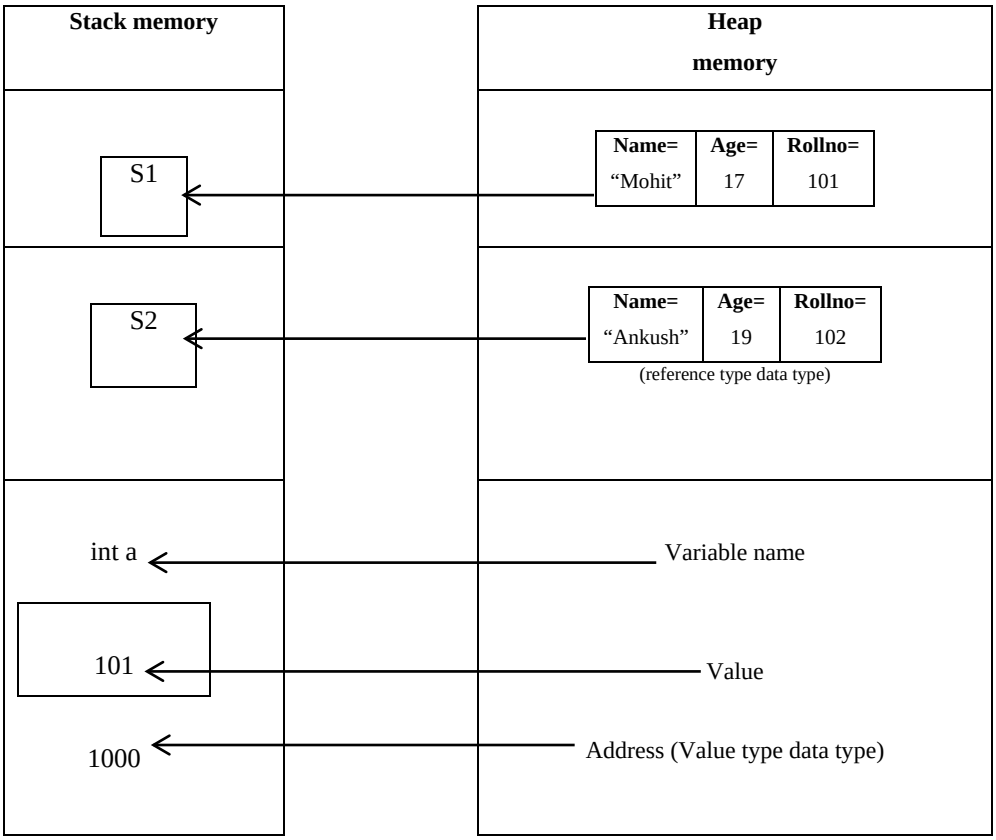
Class is the collection of data members and methods. Methods are also known as member functions. Data members are used to store the values and methods are used to be called. Data members are generally defined as private so that they may be hidden from outside the class and methods are generally made public so that object may access them.

- Class is also known as template data.
- Class is the user defined data type.
- Class is the logical concept of data because it does not consume memory at runtime.
- Class acts as a blue print for any problem as it has to be defined only once.
- Class may also be considered as the collection of similar type of objects.
- In Class members are encapsulated so members of class cannot be used directly outside the class.
- In Class members are encapsulated so outside data cannot be accessed inside the class directly.
- Class members may have different modifiers.
- Class members may have different visibility modes (access specifier).
- Classes may be nested or local.
- There may be multiple classes in one program.
- C++ program may also be developed without class definition like C language.

5.5 OBJECT

Object is declared to store the values and execute the methods defined inside the class. The may access the public members of their class. They cannot access private or protected members of the class. There may be multiple objects of one class depending on their need in the program. All objects will consume memory separately. Updating data members in object will not update values in other objects. We need to update each object separately as their allocated memory is different.

- An object is the instance of class.
- We can create multiple objects of one class.
- Objects consume memory at run time.



- Object consumes memory separately for each object.
- They are also known as physical concept of data.
- Objects can access public members of the class outside the class.
- Objects are called by reference in methods calling as they are created from class and class is the reference type data type.
- Reference of the object is stored in stack memory.

- Memory for values of object is reserved in heap memory when we allocate the memory with new keyword.
- Memory for object is automatically de-allocated by garbage collector when memory is of no-use in program.

Any object may call any public method with dot operator, which is declared in its class definition.

Check Your Progress

How many object can be created of a Class in C++?

5.6 ACCESSING DATA MEMBERS

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

Member Functions in Classes

There are 2 ways to define a member function:

- Inside class definition
- Outside class definition

To define a member function outside the class definition we have to use the scope resolution '**::**' operator along with class name and function name.

Example 5.1 Program to demonstrate method declaration inline and outside class

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
class student
{
   private:
    char sname[50];
```

```
      int rno;
      float avg;
     public:
       void input();
       void display()
       {
         cout<<"\n\t\t\tStudent record";
         cout<<"\n STUDENT NAME : "<<sname;
         cout<<"\n STUDENT ROLL NUMBER : "<<rno;
         cout<<"\n STUDENT AVERAGE MARKS : "<<avg;
       }
};
  void student::input()// outside definition
  {
     cout<<"\n ENTER STUDENT RECORD \n";
     cout<<" Enter Student name ";
     gets(sname);
     cout<<" Enter Roll Number ";
     cin>>rno;
     cout<<" Enter average marks ";
     cin>>avg;
  }
  void main()
  {
     student s1;
     clrscr();
     s1.input();
     s1.display();
     getch();
  }
```

Output:

ENTER STUDENT RECORD

Enter Student name Mohit

Enter Roll Number 101

Enter average marks 89.5

Student record

STUDENT NAME : Mohit

STUDENT ROLL NUMBER : 101

STUDENT AVERAGE MARKS : 89.5

Note that all the member functions defined inside the class definition are by default **inline**, but you can also make any non-class function inline by using keyword inline with them. Inline functions are actual functions, which are copied everywhere during compilation, like pre-processor macro, so the overhead of function calling is reduced.

5.7 SCOPE RESOLUTION OPERATOR

There are three uses of '**::** ' operators in C++

- Outside definition of methods of the class.
- Accessing global variable's value
- Accessing class members in some cases.

First use of **::** (scope resolution) operator in C++ is for outside method definition for classes. That is already discussed in the previous section.

Second use of **::** (scope resolution) operator is access global variables. There may be cases where we have global and local variables with same name. In that case, if we specify the variable name, it will access the local variable because local variable hides the global variable. We can use the scope operator as unary operator to access the global variable.

For example:

```
int count = 0;
int main()
{
  int count = 0;
```

```
::count = 1;  // set global count to 1
count = 2;    // set local count to 2
```

```
 return 0;
}
```

Count variable, which is declared outside the main function is global variable.

Count variable, which is declared inside the main function is local variable.

Local variables hides the global variables so the declaration of `count` declared inside the `main` function hides the integer named `count` declared outside the main.

The statement `::count = 1` accesses the variable named `count` declared in global namespace scope.

Third use of scope resolution operator is implemented in cases of class. It may be used to access the class members via class separated with it. It qualifies the class name with class member name. If a class member name is hidden, you can use it by qualifying it with its class name and the class scope operator. It can also access the static members of the class.

Example. 5.2 Program to demonstrate method declaration inline and outside class

```
#include<iostream.h>
#include<conio.h>
class math
{
   public:
   static int sum(int a, int b)
   {
     int c;
     c=a + b;
     return c;
   }
};
```

```
void main()
{
   int x, y, z;
   clrscr();
   cout<<"Enter two numbers ";
   cin>>x>>y;
   z=math::sum(x, y);
   cout<<"\n Sum="<<z;
   getch();
}
```

Output:

Enter two numbers 34 45

Sum =79

Check Your Progress

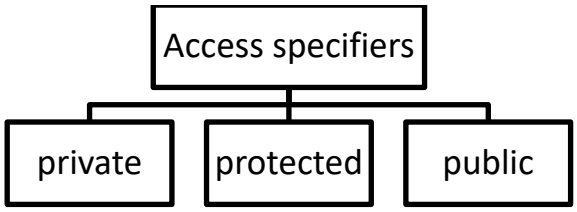
Where a class member function can be defined?

Static data member :

In C++, we can also define static data methods. But static members may be accessed via class names separated with **::** operator. Besides that static methods may access static members only. They cannot access non-static members.

5.8 DATA HIDING

Data hiding is also known as Information hiding. Data hiding is applicable in C++ as the feature of Object oriented programming concept. Data hiding restrict the access of data outside the class. Data which needs to be restricted in C++ classes are made private with private keyword at declaration time. All members including data members and methods which are made private cannot be accessed directly with object outside the class and they also cannot be inherited in sub class.



Private members : These can be accessed only from within the members of the same class.

Protected members : These can be accessed only from within other members of the same class and its derived classes.

Public members : These can be accessed from anywhere where the object is accessible.

By declaring the member variables and functions as a private in a class, the members are hidden from outside the class. Those private members and data cannot be accessed by the object directly.

Private and public members are already discussed the example-02 of this unit.

We have defined student name, roll number and average marks as private. It means it can be accessed inside the class directly but it may not be accessed outside the class directly or va object. If we try to do that, it will produce compile time error.

We have defined input() and display() as public. They may be accessed directly inside the class and outside the class via object name.

Check your Progress

Can we declare a member function private?

5.9 EMPTY CLASSES

C++ allows creating an Empty class. Empty class is that class which is defined but there is member defined inside the class. It's object will consume 1 byte in memory because it will take minimum memory storage. It is just like normal class except the fact that there is no data member. One byte is the minimum memory amount that could be occupied.

Example 5.3 Program to find the size of empty class.

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

```
class math
{
};
void main()
{
   math m;
   clrscr();
   cout<<"Size of math class "<<sizeof (m);
   getch();
}
```

Output:

Size of math class 1

5.10 POINTER WITHIN CLASS

Pointers are the very powerful features of C++. Pointers support dynamic memory management in C++. Dynamic memory management means, memory will be allocated and de-allocated at the runtime rather than compile time.

To manage the compile time and runtime memory, C++ used stack and heap memory. Stack memory is used for compile time memory management while heap memory is used for dynamic memory management.

Stack memory is generally low in size as compared to heap memory. It is automatically allocated and de-allocated at runtime. As programmer, we need not too do anything extra for that.

Heap memory is generally high in size as compared to stack memory. It has to be explicitly allocated and de-allocated by the C++ programmer. C++ provides two keywords-

- new : to allocate memory at runtime.
- delete : to de-allocate memory at runtime.

While allocating memory, we have to specify the size of memory in number of bytes but during de-allocation, we need not to specify its size. Memory will be allocate sequentially at runtime.

If we want to fix memory at compile time, we need not to do anything extra for memory management. For fixing memory at runtime, we have to manage the memory address for retrieving stored values.

Pointers are the variables, which store the address of any location.

Pointers may do the following-

- Use in function call by reference
- Use in referring / pointing any existing memory location
- Allocating and de-allocating new memory at runtime

In 32 bit C++, a pointer consumes 2 bytes as it needs to store only address. The address will indicate where the values are stored.

In pointer we use two major operators-

*	:	dereferencing operator To access the value stored at given address
&	:	to access the address of given variable/object

Pointer declaration-

Datatype *PointerName;

For example-

int *a;	pointer to integer
float *b;	pointer to float
char *c;	pointer to char
student *s;	pointer to student class (if student is a class)
employee *e;	pointer to employee structure (if employee is structure)
int *arr[10];	array of pointer to integer
student *s[10];	array of pointer to student class

Storing Address in pointers:

ptrName = &VarNAme;

for example-

```
int a,*ptr;
ptr=&a;
```

where

a is the variable of integer type

ptr is pointer to integer

Pointer to pointer-

Pointer to pointer may also be created easily. In that case we have to define double pointer.

Datatype **ptrName;

For example-

int a, *ptr1, **ptr2;

where

a is integer variable

*ptr1 is single pointer (pointer to integer)

**ptr2 is double pointer (pointer to pointer)

Example 5. 4 Program to show the referencing in pointers

```
#include<iostream.h>
#include<conio.h>
void main()
{
   clrscr();
   int a,*p1,**p2;
   a=10;   // initialized with value
   p1=&a;  // stores the address of variable a
   p2=&p1; // stores the address of p1
   cout<<"\n &a ="<<&a;
   cout<<"\n &p1="<<&p1;
   cout<<"\n &p2="<<&p2<<"\n";
   cout<<"\n a ="<<a;
   cout<<"\n p1="<<p1;
   cout<<"\n p2="<<p2<<"\n";
   cout<<"\n *a ="<<"Error";
   cout<<"\n *p1="<<*p1;
```

```
    cout<<"\n *p2="<<*p2<<"\n";
    cout<<"\n **a ="<<"Error";
    cout<<"\n **p1="<<"Error";
    cout<<"\n **p2="<<**p2<<"\n";
    getch();
  }
```

Output:

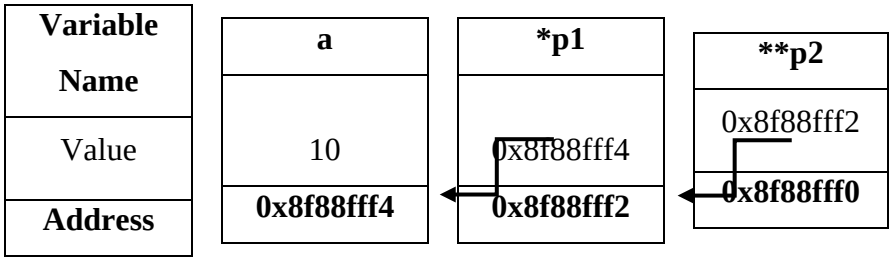
```
&a=0x8f88fff4
&p1=0x8f88fff2
&p2=0x8f88fff0

a=10
p1=0x8f88fff4
p2=0x8f88fff2

*a=Error
*p1=10
*p2=0x8f88fff4

**a=Error
**p1=Error
**p2=10
```

In the above example, we have write down the name, value and address of all variables-



Where

a is integer variable

p1 is pointer to a

p2 is pointer to p1 (double pointer to a)

* Denotes that value stored there is not actually a values, it is an address. It will jump to given address and return the value.

** denotes the task of * two times.

Check Your Progress

Which operator used for dereferencing or indirection ?

Pointer to Data Members of Class :

Pointers may be created for objects. In that case, they will refer to objects. Memory of objects may be used of existing objects or new memory may be allocated.

If we wish to access the existing memory of any object then, just assign the reference of existing object.

Syntax will be

className object, *ptrName;

ptrName=&object;

ptrName->Method();

For accessing methods or other members, we have to use -> operator instead of dot operator.

Example 5.5 Program to show the referencing of pointers with data members

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<string.h>
class student
{
```

```
  private:
    char sname[50];
    int rno;
    float avg;
  public:
    void input();
    void display();
};
void student::input()
{
   cout<<"\n ENTER STUDENT RECORD \n";
   cout<<" Enter Student name ";
   gets(sname);
   cout<<" Enter Roll Number ";
   cin>>rno;
   cout<<" Enter average marks ";
   cin>>avg;
}
void student::display()
{
    cout<<"\n\t\t\tStudent record";
    cout<<"\n STUDENT NAME : "<<sname;
    cout<<"\n STUDENT ROLL NUMBER : "<<rno;
    cout<<"\n STUDENT AVERAGE MARKS : "<<avg;
}

void main()
{
   clrscr();
   student s1,*ptr;
   ptr=&s1;
```

```
    s1.input();
    ptr->display();
    s1.display();
    getch();
}
```

Output:

```
ENTER STUDENT RECORD
Enter Student name Sanjay
Enter Roll Number 101
Enter average marks 56.5

                    Student  record
STUDENT NAME :  Sanjay
STUDENT ROLL NUMBER : 101
STUDENT AVERAGE MARKS : 56.5

                    Student  record
STUDENT NAME :  Sanjay
STUDENT ROLL NUMBER : 101
STUDENT AVERAGE MARKS : 56.5
```

In the above example, we have stored the reference of one object into pointer to class. Once reference is assigned, we may call the methods either with -> (arrow) operator through pointer or with . (dot) operator with object. Nut here no dynamic memory allocation is done. We are just referring existing memory.

Using Pointers with Objects (allocating memory):

If we wish to allocate new memory t pointer then we have to allocate it with new operator and delete operator will be used to de-allocate the memory.

Syntax will be

```
className object, *ptrName;
ptrName=new ClassName;
ptrName->Method();
delete ptrName;
```

For accessing methods or other members, we have to use -> operator instead of dot operator.

Example 5.6 Program to show the dynamic memory allocation to pointer to class.

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<string.h>
class student
{
  private:
    char sname[50];
    int rno;
    float avg;
  public:
    void input();
    void display();
};
void student::input()
{
   cout<<"\n ENTER STUDENT RECORD \n";
   cout<<" Enter Student name ";
   gets(sname);
   cout<<" Enter Roll Number ";
   cin>>rno;
   cout<<" Enter average marks ";
```

```
    cin>>avg;
  }
  void student::display()
  {
      cout<<"\n\t\t\tStudent record";
      cout<<"\n STUDENT NAME : "<<sname;
      cout<<"\n STUDENT ROLL NUMBER : "<<rno;
      cout<<"\n STUDENT AVERAGE MARKS : "<<avg;
  }

  void main()
  {
     clrscr();
     student *ptr;
     ptr=new student();
     ptr->input();
     ptr->display();
     delete ptr;
     getch();
  }
```

Output:

```
ENTER STUDENT RECORD
Enter Student name Rohan Garg
Enter Roll Number 105
Enter average marks78.5

                                Student  record
STUDENT NAME :  Rohan Garg
STUDENT ROLL NUMBER : 105
STUDENT AVERAGE MARKS : 78.5
```

In the above example, we have allocated memory of student class to pointer. It will allocate memory for object of student class and the reference of that object will be stored in pointer. So this time pointer is storing the new address. And that memory is allocated at runtime. Once the task is over, we should de-allocate the memory. Once reference is assigned, we may call the methods either with -> (arrow) operator through pointer.

Check Your Progress

What happens when delete is used for a NULL pointer?

5.11 PASSING OBJECTS AS ARGUMENTS

There may be situation when we need to send the objects in the functions for some purpose. In that case, like other variables, we may send the objects as parameter. We just have to specify the name of the class as data type in parameter list.

Example 5.7 Program to find and display the greater of two students on the basis of their average marks.

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
class student
{
   private:
    char sname[50];
    int rno;
    float avg;
   public:
     void input();
     float getavg()
     {
```

```
        return avg;
    }
    void display()
    {
      cout<<"\n\t\t\tStudent record";
      cout<<"\n STUDENT NAME : "<<sname;
      cout<<"\n STUDENT ROLL NUMBER : "<<rno;
      cout<<"\n STUDENT AVERAGE MARKS : "<<avg;
    }
};
void student::input()
{
   cout<<"\n ENTER STUDENT RECORD \n";
   cout<<" Enter Student name ";
   gets(sname);
   cout<<" Enter Roll Number ";
   cin>>rno;
   cout<<" Enter average marks ";
   cin>>avg;
}
void compare(student s1, student s2);
void main()
{
   student s1,s2;
   clrscr();
   s1.input();
   s2.input();
   compare(s1,s2);
   getch();
}
```

```
void compare(student s1,student s2)
{
 if(s1.getavg()>s2.getavg())
    s1.display();
 else
    s2.display();
}
```

Output:

ENTER STUDENT RECORD

Enter Student name Priya

Enter Roll Number 123

Enter average marks 87.5

ENTER STUDENT RECORD

Enter Student name Sanjay

Enter Roll Number 101

Enter average marks 56.5

Student record

STUDENT NAME : Sanjay

STUDENT ROLL NUMBER : 101

STUDENT AVERAGE MARKS : 56.5

In the above example, we have defined the compare function. This function takes two objects as parameter. In its definition, it compares the average marks of the students via getavg() method and display the record of student who scores higher marks.

Check Your Progress

Is it compulsory that If an object is passed to a function then it must be returned the function?

5.12 RETURN AN OBJECT FROM THE FUNCTION

Object is the instance of the class. Like we return any scalar value (int, char etc.) from method, we may also return an object from the function. We have to specify the name of the class as return type in the function definition. Return statement will be same as in other cases of returning values.

Example 5.8 Program to find the greater of two students on the basis of their average marks. And display the record after returning.

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
class student
{
   private:
    char sname[50];
    int rno;
    float avg;
   public:
     void input();
     float getavg()
     {
         return avg;
     }
     void display()
     {
        cout<<"\n\t\t\tStudent record";
        cout<<"\n STUDENT NAME : "<<sname;
        cout<<"\n STUDENT ROLL NUMBER : "<<rno;
        cout<<"\n STUDENT AVERAGE MARKS : "<<avg;
     }
```

```
};
void student::input()
{
   cout<<"\n ENTER STUDENT RECORD \n";
   cout<<" Enter Student name ";
   gets(sname);
   cout<<" Enter Roll Number ";
   cin>>rno;
   cout<<" Enter average marks ";
   cin>>avg;
}

student compare(student s1, student s2);

void main()
{
   student s1,s2,max;
   clrscr();
   s1.input();
   s2.input();
   max=compare(s1,s2);
   max.display();
   getch();
}
student compare(student s1,student s2)
{
  student max;
  if(s1.getavg()>s2.getavg())
     max=s1;
  else
     max=s2;
  return max;
}
```

Output:

ENTER STUDENT RECORD

Enter Student name Priya

Enter Roll Number 123

Enter average marks 37.5

ENTER STUDENT RECORD

Enter Student name Sanjay

Enter Roll Number 101

Enter average marks 56.5

Student record

STUDENT NAME : Sanjay

STUDENT ROLL NUMBER : 101

STUDENT AVERAGE MARKS : 56.5

In the above example, we have defined the compare function. This function takes two objects as parameter and return one object of the same class. In its definition, it compares the average marks of the students via getavg() method and return the record of student who scores higher marks. After returning the object in the main(), we display the record. This is done in such a manner because, at times, we need object to be processed after returning object.

Check Your Progress

Which error will be produced if a local object is returned by reference outside a function?

5.13 FRIEND FUNCTIONS

Data hiding is one of the most important features of OOPs. It restricts the accessing of private members of the class via object. In such a scenario, we may access the private data member by returning value

through public method. But we may access only one value via on method. For accessing multiple values, we have to define such multiple methods. Besides this, that method may be accessed via any object anywhere because that has to be public.

There may be situations, where we may need to access private data members to be accessed but only once for some particular purpose. In such scenarios, we may define friend functions. These functions may access private data members of the class as well.

Friend function becomes mandatory when we need private data members of more than two objects from different classes at same time. It helps us in writing complicated code as well.

Friend function is defined and declared like a normal function. One additional thing we have to do, that is we have to declare that function followed by friend keyword inside the class in which we have to define it as friend. One method may be friend in more than one class as well. After declaring any function friend inside the class, that function may access the private members of that class via object but only inside the definition of that friend function only.

One important point to remember is that friend function will be accessed like normal function. It cannot be access with class name or object name like methods.

Syntax : The general syntax of Friend Function is as follows:

```
class class_name

{

   ... .. ...

   friend return_type function_name(argument/s);

   ... .. ...

}
```

Example 5.9 Program to find the greater of two students on the basis of their average marks via friend function. And display the record after returning.

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

```
#include<conio.h>
#include<stdio.h>
class student
{
   private:
    char sname[50];
    int rno;
    float avg;
   public:
    friend student compare(student s1, student s2);   // friend function declaration
     void input();
     void display()
     {
        cout<<"\n\t\t\tStudent record";
        cout<<"\n STUDENT NAME : "<<sname;
        cout<<"\n STUDENT ROLL NUMBER : "<<rno;
        cout<<"\n STUDENT AVERAGE MARKS : "<<avg;
     }
};
void student::input()
{
   cout<<"\n ENTER STUDENT RECORD \n";
   cout<<" Enter Student name ";
   gets(sname);
   cout<<" Enter Roll Number ";
   cin>>rno;
   cout<<" Enter average marks ";
   cin>>avg;
}

student compare(student s1, student s2);   // normal function declaration
```

```
void main()
{
   student s1,s2,max;
   clrscr();
   s1.input();
   s2.input();
   max=compare(s1,s2);   // calling friend function like normal function
   max.display();
   getch();
}

student compare(student s1,student s2)
{
  student max;
  if(s1.avg>s2.avg)     // accessing private members due to friend function
     max=s1;
  else
     max=s2;
  return max;
}
```

Output:

```
ENTER STUDENT RECORD
Enter Student name Priya
Enter Roll Number 123
Enter average marks 37.5

ENTER STUDENT RECORD
Enter Student name Sanjay
Enter Roll Number 101
Enter average marks 56.5
```

```
                    Student  record
STUDENT NAME :  Sanjay
STUDENT ROLL NUMBER : 101
STUDENT AVERAGE MARKS : 56.5
```

In the above example, we have defined the compare function. This function takes two objects as parameter and return one object of the same class. In its definition, it compares the average marks of the students. But this function does not need any public method for accessing private value of the class as compare is declared as friend inside the student class.

Then compare function compare two private members via object and return the record of student who scores higher marks. After returning the object in the main(), we display the record.

Function as Friend function in two class :

One function may be declared as friend in two classes as well. For that we just have to declare that function in both classes.

Example: 5.10 Program to compare the names of student and teacher for equality and display the message according.

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<string.h>
class teacher;  // declaration of class as it has to be used before its definition in friend
class student
{
   private:
    char sname[50];
    int rno;
    float avg;
   public:
     friend void compare(student s, teacher t);  // normal function as friend
```

```
declaration
     void input();
};
void student::input()
{
   cout<<"\n ENTER STUDENT RECORD \n";
   cout<<" Enter Student name ";
   gets(sname);
   cout<<" Enter Roll Number ";
   cin>>rno;
   cout<<" Enter average marks ";
   cin>>avg;
}

class teacher
{
   private:
    char tname[50];
    int tid;
    float salary;
   public:
     friend void compare(student s, teacher t);  // normal function as friend
declaration
     void input();
};

void compare(student s, teacher t);   // normal function declaration

void teacher::input()
{
   cout<<"\n ENTER TEACHER RECORD \n";
   cout<<" Enter teacher name ";
   gets(tname);
```

```
    cout<<" Enter ID ";
    cin>>tid;
    cout<<" Enter Salary ";
    cin>>salary;
  }

  void main()
  {
    student s;
    teacher t;
    clrscr();
    s.input();
    t.input();
    compare(s,t);          // friend function calling
    getch();
  }

  void compare(student s,teacher t)  // friend function definition
  {
   if(strcmp(s.sname,t.tname)==0)
     cout<<"\n name of teacher and student is equal";
   else
     cout<<"\n name of teacher and student is different";
  }
```

Output:

```
ENTER STUDENT RECORD
Enter Student name Priya
Enter Roll Number 123
Enter average marks 37.5

ENTER TEACHER RECORD
```

```
Enter Student name  Amit Saxena
Enter ID 5967
Enter Salary 57000

name of teacher and student is different
```

In the above example, we have defined the compare function. This function takes two objects as parameter and compares the name of student with name of teacher. As we need to access the private member of both classes, we have introduced this function as friend inside both classes.

One point to remember during friend declaration in multiple classes is that, we have to declare the other classes before their definition as we have to use them in friend function declaration.

Check Your Progress

Which rule will not affect the friend function?

5.14 FRIEND CLASS

Like friend functions, we may use friend classes in C++. If we want to access the private members of one class to be accessed in any other class via object, we may declare that class as friend in the first class.

Example 5.11 Program to compare the names of student and teacher for equality and display the message accordingly with the help of friend class only.

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<string.h>
class teacher; // normal class declaration
class student
{
   friend teacher; // teacher class declaration as friend in student class
   private:
```

```
    char sname[50];
    int rno;
    float avg;
   public:
     void input();
};
void student::input()
{
   cout<<"\n ENTER STUDENT RECORD \n";
   cout<<" Enter Student name ";
   gets(sname);
   cout<<" Enter Roll Number ";
   cin>>rno;
   cout<<" Enter average marks ";
   cin>>avg;
}

class teacher
{
   private:
    char tname[50];
    int tid;
    float salary;
   public:
     void compare(student s);  // normal method with student as parameter
     void input();
};

void teacher::input()
{
   cout<<"\n ENTER TEACHER RECORD \n";
   cout<<" Enter teacher name ";
```

```
    gets(tname);
    cout<<" Enter ID ";
    cin>>tid;
    cout<<" Enter Salary ";
    cin>>salary;
}

void teacher::compare(student s)
{
  if(strcmp(s.sname,tname)==0)  //accessing private member of student due to friend class
     cout<<"\n name of teacher and student is equal";
  else
     cout<<"\n name of teacher and student is different";
}

void main()
{
    student s;
    teacher t;
    clrscr();
    s.input();
    t.input();
    t.compare(s);
    getch();
}
```

Output:

ENTER STUDENT RECORD

Enter Student name Priya

Enter Roll Number 123

```
Enter average marks 37.5


ENTER TEACHER RECORD
Enter Student name  Amit Saxena
Enter ID 5967
Enter Salary 57000


name of teacher and student is different
```

In the above example, we have declared teacher class as friend class inside the student class. After declaring this, teacher class may access the private members of the student class via student class object. We have to remember that teacher is the friend of student so teacher may access the private members of student but student is not the friend class inside the teacher so student class cannot access the private members of the teacher class.

Check Your Progress

What is the need of forward declaration of a class?

5.15 CONSTANT PARAMETERS AND MEMBER FUNCTIONS

Formal parameters may be modified inside the method definition. If method call is call by reference, it will be reflected back in formal parameter. If method call is call by value, it will not be reflected back in the actual parameters.

But there may be situation when, we want to restrict the changes in formal parameters. This can be done with the help of constant parameter. If we have defined any parameter as const, it cannot be changed in the method definition. If we try to change its value inside that method, compiler will produce the error.

Example 5.12 Program to store the student record and then update the average marks of the student.

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

```
#include<conio.h>
#include<stdio.h>
class student
{
   private:
    char sname[50];
    int rno;
    float avg;
   public:
     void input();
     void updateavg(const float a)
     {
         avg=avg+a;
     }
     void display()
     {
        cout<<"\n\t\t\tStudent record";
        cout<<"\n STUDENT NAME : "<<sname;
        cout<<"\n STUDENT ROLL NUMBER : "<<rno;
        cout<<"\n STUDENT AVERAGE MARKS : "<<avg;
     }
};
void student::input()
{
   cout<<"\n ENTER STUDENT RECORD \n";
   cout<<" Enter Student name ";
   gets(sname);
   cout<<" Enter Roll Number ";
   cin>>rno;
   cout<<" Enter average marks ";
   cin>>avg;
}
```

```
void main()
{
   student s;
   clrscr();
   s.input();
   s.display();
   float x;
   cout<<"\n\n Enter the marks to be increased ";
   cin>>x;
   s.updateavg(x);
   s.display();
   getch();
}
```

Output:

ENTER STUDENT RECORD

Enter Student name Sanjay

Enter Roll Number 101

Enter average marks 80.2

Student record

STUDENT NAME : Sanjay

STUDENT ROLL NUMBER : 101

STUDENT AVERAGE MARKS : 80.2

Enter the marks to be increased 6.0

Student record

STUDENT NAME : Sanjay

STUDENT ROLL NUMBER : 101

STUDENT AVERAGE MARKS : 86.2

In the above example, average marks are updated through updateavg() method. But we have defined parameter x as constant in its definition, so we cannot change the value of x inside that method else it will produce compile time error.

Check Your Progress

The constants are also called as_____

5.16 STRUCTURES AND CLASSES

C++ is the object oriented language and it is also known as extension of C language. So it supports both class and structure.

There is a difference class and structure. Members in the class are private by default while members in the structure are public by default. We may change the visibility of member in both class and structures. Unlike structures in C language, we may define methods inside the structures. But they will be public by default too.

5.17 STATIC MEMBERS

Earlier, we have defined variables as of 4 types in C++ language. We have already gone through local and global variables.

If we declare any data member inside the class, it will be known as instance variable because it will consume separately memory for each object. As memory is allocated differently, values may be different too.

There may be scenarios, where we may need the common values for all objects. In that case, we need class variables. Class variables are those variables which consumes memory only once for all object of that class and updating value by one object will affect all other objects of that class. Such class variable are declared with the help of static keyword.

If we specify static keyword before any data member, it will be treated as static data member. It will consume memory only once irrespective of number of objects of that class. Such data member may be accessed via :: (scope resolution operator) via class name rather than object name. If it is public, it may be accessed outside the class as well.

Example 5.13 Program to store two student's record with common school name.

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

```
#include<conio.h>
#include<stdio.h>
#include<string.h>
class student
{
  private:
    char sname[50];
    int rno;
    float avg;
  public:
    static char schoolname[20];
    void input();
    void display()
    {
        cout<<"\n\t\t\tStudent record";
        cout<<"\n STUDENT NAME : "<<sname;
        cout<<"\n STUDENT ROLL NUMBER : "<<rno;
        cout<<"\n STUDENT AVERAGE MARKS : "<<avg;
        cout<<"\n SCHOOL NAME : "<<schoolname;
    }
};
void student::input()
{
   cout<<"\n ENTER STUDENT RECORD \n";
   cout<<" Enter Student name ";
   gets(sname);
   cout<<" Enter Roll Number ";
   cin>>rno;
   cout<<" Enter average marks ";
   cin>>avg;
}
```

```
char student::schoolname[20];

void main()
{
   student s1,s2;
   strcpy(student::schoolname,"My School");
   clrscr();
   s1.input();
   s2.input();
   s1.display();
   s2.display();
   getch();
}
```

Output:

```
ENTER STUDENT RECORD
Enter Student name Sanjay
Enter Roll Number 101
Enter average marks 56.5

ENTER STUDENT RECORD
Enter Student name Amit
Enter Roll Number 105
Enter average marks 86.2

                    Student  record
STUDENT NAME :  Sanjay
STUDENT ROLL NUMBER : 101
STUDENT AVERAGE MARKS : 56.5
SCHOOL NAME :My School

                    Student  record
```

STUDENT NAME : Sanjay
STUDENT ROLL NUMBER : 105
STUDENT AVERAGE MARKS : 86.2
SCHOOL NAME :My School

Static & Non-Static data members in C++ Class		
Sno	**Static data members**	***Non-Static data members***
1.	Consumes memory in context of class.	Consumes memory in context of object.
2.	Consumes memory only once.	Consumes memory separately for each object.
3.	Memory is allocated at the time of class loading.	Memory is allocated at the time of object instantiation.
4.	Memory is de-allocated at the end of program.	Memory is de-allocated at the end of block.
5.	Values are common for all objects of that class.	Values may be different for all objects of that class.
6.	Also known as class variables.	Also known as instance variable.
7.	May access with class reference.	May access with object reference.

Static Function Members

Like static data members, we may define static methods. They may be accessed via class names. But we have to remember that static methods cannot access non-static members directly. They are generally used to access static data members or to perform some operation which are irrelevant to objects or non-static data members. Static members may be accessed with :: operator via class name.

Example of static method is already defined in the Example-02.

Static & Non-Static methods		
Sno	**Static methods**	***Non-Static Methods***
1.	Can access with class reference.	Can access with object reference.
2.	They cannot access non-static members.	They cannot access static members.

Check Your Progress

Which function can be called without using an object of a class in C++.

5.18 SUMMARY

In This chapter we discussed about the main characteristics of OOPs, how to create class and object, what is the use of pointer and how we can use it and also when and where we should use friend class also the importance of static members and the difference between static and non-static members.

5.19 EXERCISE

Q1. Explain the implementation of class and objects in C++ with example.

Q2. What are static data members and static methods? How they are different from non-static members. Differentiate with example.

Q3. What is pointers in C++? Explain in regard of classes with example.

Q4. What is friend function? How it is different from friend class?

Q5. Write a program in C++ to store two employee records. Compare them on the basis of their salary and display the record who earns higher salary.

Q6. Write a program in C++ to store 10 records of employees through objects. Find and display the record of any employee on the basis of employee ID. Record should contain employee ID, name, address and salary.

Q7. Write a program in C++ to store 10 records of employees through objects. Arrange them in ascending order on the basis of their roll number. Records should also store their name and average marks.

Q8. Write a program in C++ to store 10 records of employees through objects. Arrange them in ascending order on the basis of their roll number. Records should also store their name and average marks. Implement this via friend function.

Q9. Write a program in C++ to store 10 records of books. Find and display the record of book which is the costliest book.

Q10. Write a program in C++ to store two records of teachers. Swap them. Each record should contain teacher's ID, name, address and subject.

UNIT-6 OBJECT INITIALIZATION AND CLEAN-UP

Structure :

6.1 INTRODUCTION

In this unit, we will focus on constructor and destructor. Constructors are special class functions which performs initialization of every object. The Compiler calls the Constructor whenever an object is created. Constructors initialize values to object members after storage is allocated to the object. Whereas, Destructor on the other hand is used to destroy the class object.

6.2 OBJECTIVE

The main objective of this unit is to define the need of constructor and destructor. And also defined the type of the constructor and the way how they can be used.

6.3 CONSTRUCTOR

A constructor is a special type of member function that initialises an object automatically when it is created. Compiler identifies a given

member function is a constructor by its name and the return type. Constructor has the same name as that of the class and it does not have any return type. The constructor is generally public.

There are some characteristics of constructor:

- Its name is same as the class name.
- It never returns a value not even void.
- It is executed once for each object.
- It is called separately for each object.
- It may be parameterized.
- It may be overloaded.
- It may be static.

TYPE OF CONSTRUCTOR-

- Implicit Default constructor: An implicit default constructor, which is used to set the state of its objects with default values.
- Default constructor/ non-parameterized constructor: A default constructor is a constructor without any parameters.
- Parameterized constructor: Parameterized constructor is used to pass user defined values in object state.
- Dynamic constructor: Dynamic constructor is used to allocate the memory where size of memory will be finalized at runtime.
- Copy constructor: Copy constructor is used to create a clone of existing object with separate allocated memory.

Example 6.1 Program to create constructor in student class and use it

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<string.h>
class student
{
  private:
    char sname[50];
```

```
    int rno;
    float avg;
  public:
    student();  // non-parameterised constructor declaration

    void input();
    void display();
};

student::student()   // non-parameterised constructor definition
{
 cout<<"\n Non-parameterised constructor called";
 strcpy(sname,"UNKNOWN");
 rno=0;
 avg=0.0f;
}

void student::input()
{
   cout<<"\n ENTER STUDENT RECORD \n";
   cout<<" Enter Student name ";
   gets(sname);
   cout<<" Enter Roll Number ";
   cin>>rno;
   cout<<" Enter average marks ";
   cin>>avg;
}
void student::display()
{
    cout<<"\n\t\t\tStudent record";
    cout<<"\n STUDENT NAME : "<<sname;
```

```
    cout<<"\n STUDENT ROLL NUMBER : "<<rno;
    cout<<"\n STUDENT AVERAGE MARKS : "<<avg;
}

void main()
{
   clrscr();
   student s1,s2;
   s1.input();
   s1.display();  //displays values given as input
   s2.display();  //displays constructor values
   getch();
}
```

Output:

```
Non parameterised constructor called
Non Parameterised constructor called
ENTER STUDENT RECORD
Enter Student  name Rahul
Enter Roll Number 101
Enter average marks 57.5
                                Student record
STUDENT NAME  :  Rahul
STUDENT ROLL  NUMBER  :  101
STUDENT AVERAGE MARKS  :  57.5
                                Student record
STUDENT NAME  :  UNKNOWN
STUDENT ROLL  NUMBER  :  0
STUDENT AVERAGE MARKS  :  0
```

In the above example, constructor is called once for s1 object and once for s2 object. Values of s1 object will be updated via input method. But values

of s2 object are not updated here so it will display the same values by display method.

If we have not specified any constructor in the class, compiler will define its own constructor to allocate memory to object with garbage values.

How constructor works?

In the above example, student() is a public constructor. It is non-parameterised in nature. As soon as object is declared at runtime, constructor is called automatically be compiler. It will be called separately for each object. Its definition is like other methods but its purpose is to assign some values in some data members of the class before using them via object. If multiple objects are declared in C++, constructor is called in FIFO (First In First Out) order.

Example 6.2 Program to create parameterised constructor in student class and use it.

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<string.h>
class student
{
  private:
    char sname[50];
    int rno;
    float avg;
  public:
    student(char n[50], int r, float a);
    void input();
    void display();
};

student::student(char n[50], int r, float a)     // non-parameterised constructor
{
```

```
  cout<<"\n Parameterised constructor called";
  strcpy(sname,n);
  rno=r;
  avg=a;
}

void student::input()
{
   cout<<"\n ENTER STUDENT RECORD \n";
   cout<<" Enter Student name ";
   gets(sname);
   cout<<" Enter Roll Number ";
   cin>>rno;
   cout<<" Enter average marks ";
   cin>>avg;
}
void student::display()
{
    cout<<"\n\t\t\t Student record";
    cout<<"\n STUDENT NAME : "<<sname;
    cout<<"\n STUDENT ROLL NUMBER : "<<rno;
    cout<<"\n STUDENT AVERAGE MARKS : "<<avg;
}

void main()
{
   clrscr();
   student s("Rahul",101,57.50f);
   s.display();  //displays constructor values
   getch();
```

```
}
```

Output:

```
Parameterised constructor called
                    Student record
STUDENT NAME : Rahul
STUDENT ROLL NUMBER : 101
STUDENT AVERAGE MARKS : 57.5
```

In the above example, we have defined parameterised constructor. It will be called when we have passed parameters during declaration of that object.

Example 6.3 Program to create copy parameterised constructor in student class and use it

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<string.h>
class student
{
  private:
    char sname[50];
    int rno;
    float avg;
  public:
    student();      // non-parameterised constructor declaration
    student(student &s);   // copy constructor declaration
    void input();
    void display();
};
```

```
student::student()   // non-parameterised constructor
{
  cout<<"\n Non-parameterised constructor called";
  strcpy(sname,"UNKNOWN");
  rno=0;
  avg=0.0f;
}

student::student(student &s)  // copy-constructor definition
{
  cout<<"\n Non-parameterised constructor called";
  strcpy(sname,s.sname);
  rno=s.rno;
  avg=s.avg;
}

void student::input()
{
    cout<<"\n ENTER STUDENT RECORD \n";
    cout<<" Enter Student name ";
    gets(sname);
    cout<<" Enter Roll Number ";
    cin>>rno;
    cout<<" Enter average marks ";
    cin>>avg;
}
void student::display()
{
     cout<<"\n\t\t\tStudent record";
     cout<<"\n STUDENT NAME : "<<sname;
```

```
    cout<<"\n STUDENT ROLL NUMBER : "<<rno;
    cout<<"\n STUDENT AVERAGE MARKS : "<<avg;
}

void main()
{
   clrscr();
   student s1;
   s1.input();
   student s2(s1);      // copy-constructor calling
   s1.display();  //displays values given as input
   s2.display();  //displays copy constructor values
   getch();
}
```

Output:

```
Non parameterised constructor called
ENTER STUDENT RECORD
Enter Student  name Amit
Enter Roll Number 101
Enter average marks 78.5

Non Parameterised constructor called
                                Student record
STUDENT NAME  :  Amit
STUDENT ROLL  NUMBER  :  101
STUDENT AVERAGE MARKS  :  78.5
                                Student record
STUDENT NAME  :  Amit
STUDENT ROLL  NUMBER  :  101
STUDENT AVERAGE MARKS  :  78.5
```

In the above example, we have defined s1 object via son-parameterised constructor. Then we have cloned the s1 object into s2 object via copy constructor.

Keep in mind that parameter in copy-constructor is always call by reference as reference variable.

One more to be remember thing during copy-constructor is that while defining copy constructor, we also have to define non-parameterised / parameterised / both constructor. It is required because copy constructor will clone the object into new object so initially we need an object to be cloned.

Check Your Progress

What happens when a class with parameterized constructors and having no default constructor is used in a program and we create an object that needs a zero-argument constructor?

6.4 DESTRUCTOR

Destructor is the special method of the class which de-allocated the memory of the object. It will be automatically (implicitly) called. Its name is same as the class name with ~ symbol at the starting of destructor name. It cannot be parameterised. It cannot be overloaded. It is generally used when we have allocated memory to any of the data member as pointer.

If pointers are not used as data member for memory allocation, destructor is of very rare use.

Example 6.4 Program to create constructor & destructor in student class and use it.

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<string.h>
class student
{
  private:
```

```
    char sname[50];
    int rno;
    float avg;
  public:
    student();     // constructor declaration
    ~student();   // destructor declaration
    void input();
    void display();
};

student::student()   // non-parameterised constructor definition
{
 cout<<"\n Non-parameterised constructor called.\n";
 strcpy(sname,"UNKNOWN");
 rno=0;
 avg=0.0f;
}

student::~student()   // destructor definition
{
 cout<<"\n\n Destructor of student class is called";
}

void student::input()
{
   cout<<"\n ENTER STUDENT RECORD \n";
   cout<<" Enter Student name ";
   gets(sname);
   cout<<" Enter Roll Number ";
   cin>>rno;
   cout<<" Enter average marks ";
```

```
    cin>>avg;
  }
  void student::display()
  {
      cout<<"\n\t\t\tStudent record";
      cout<<"\n STUDENT NAME : "<<sname;
      cout<<"\n STUDENT ROLL NUMBER : "<<rno;
      cout<<"\n STUDENT AVERAGE MARKS : "<<avg;
  }

  void main()
  {
     clrscr();
     student s1;
     s1.display();
     getch();
  }
```

Output:

Non-parameterised constructor called

Student record

STUDENT NAME : UNKNOWN

STUDENT ROLL NUMBER : 0

STUDENT AVERAGE MARKS : 0

Destructor of the student class is called

Check Your Progress

When is the destructor of a global object called?

6.5 CONSTRUCTOR OVERLOADING

Constructor can be overloaded in a similar way as function overloading. Overloaded constructors have the same name (name of the class) but different number of arguments. Depending upon the number and type of arguments passed, specific constructor is called. Since, there are multiple constructors present, argument to the constructor should also be passed while creating an object.

Example 6.5 Program to implement constructor overloading in C++

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<string.h>
class student
{
  private:
    char sname[50];
    int rno;
    float avg;
  public:
    student();      // non-parameterised constructor declaration
    student(char n[50], int r, float a);     // paremeterised constructor declaration
    student(student &s);   // copy constructor declaration
    void input();
    void display();
};

student::student()   // non-parameterised constructor definiton
{
  cout<<"\n Non-parameterised constructor called";
```

```
  strcpy(sname,"UNKNOWN");
  rno=0;
  avg=0.0f;
}

student::student(char n[50], int r, float a)   // parameterised constructor definiton
{
  cout<<"\n Parameterised constructor called";
  strcpy(sname,n);
  rno=r;
  avg=a;
}

student::student(student &s)  // copy-constructor definition
{
  cout<<"\n Non-parameterised constructor called";
  strcpy(sname,s.sname);
  rno=s.rno;
  avg=s.avg;
}

void student::input()
{
    cout<<"\n ENTER STUDENT RECORD \n";
    cout<<" Enter Student name ";
    gets(sname);
    cout<<" Enter Roll Number ";
    cin>>rno;
    cout<<" Enter average marks ";
    cin>>avg;
```

```
}
void student::display()
{
    cout<<"\n\t\t\tStudent record";
    cout<<"\n STUDENT NAME : "<<sname;
    cout<<"\n STUDENT ROLL NUMBER : "<<rno;
    cout<<"\n STUDENT AVERAGE MARKS : "<<avg;
}

void main()
{
   clrscr();
   student s1;
   s1.input();   //non-parameterised constructor call
   student s2("Mudit", 151,59.5f);  //parameterised constructor call
   student s3(s1);            //copy constructor call
   s1.display();  //displays values given as input
   s2.display();  //displays parameterised-constructor values
   s3.display();  //displays copy constuctor calues
   getch();
}
```

Output:

```
Non parameterised constructor called
ENTER STUDENT RECORD
Enter Student  name Rahul
Enter Roll Number 101
Enter average marks 57.5

Parameterised constructor called
Non Parameterised constructor called
```

Student record

STUDENT NAME : Rahul

STUDENT ROLL NUMBER : 101

STUDENT AVERAGE MARKS : 57.5

Student record

STUDENT NAME : Mudit

STUDENT ROLL NUMBER : 151

STUDENT AVERAGE MARKS : 59.5

Student record

STUDENT NAME : Rahul

STUDENT ROLL NUMBER : 101

STUDENT AVERAGE MARKS : 57.5

In the above we have defined non-parameterised, parameterised and copy constructor. They will be invoked on the basis of their parameter list.

Check Your Progress

If the constructors are overloaded by using the default arguments, which problem may arise?

6.6 ORDER OF CONSTRUCTION AND DESTRUCTION

When a class object is created using constructors, the execution order of constructors is:

- First of all constructors of Virtual base classes are invoked. Sequence will be done in the order that they appear in the base list.
- Thereafter, constructors of non-virtual base classes are invoked as in the declaration order.
- Constructors of class members are invoked in the declaration order.

- The body of the constructor is executed.

Check Your Progress

Which destructor is called first?

6.7 CONSTRUCTOR WITH DEFAULT ARGUMENT

Default constructor is also known as non-parameterised constructor. If we have not defined any constructor, compiler will generate its own internal default constructor with garbage values.

We may also specify some parameters in parameterised constructor. If we wish to specify some default values in parameterised constructor, we may do it. It was also possible in the function as default parameters. Default parameters in constructor will be used only if, no values are specified at object declaration section.

Example 6.6 Program to implement constructor with default argument in C++

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<string.h>
class student
{
  private:
    char sname[50];
    int rno;
    float avg;
  public:
    student(char n[50], int r, float a=0.0f); // default parameter in third parameter
    void input();
```

```
    void display();
};

student::student(char n[50], int r, float a)   // parameterised constructor
{
  cout<<"\n Parameterised constructor called";
  strcpy(sname,n);
  rno=r;
  avg=a;
}

void student::input()
{
   cout<<"\n ENTER STUDENT RECORD \n";
   cout<<" Enter Student name ";
   gets(sname);
   cout<<" Enter Roll Number ";
   cin>>rno;
   cout<<" Enter average marks ";
   cin>>avg;
}
void student::display()
{
    cout<<"\n\t\t\tStudent record";
    cout<<"\n STUDENT NAME : "<<sname;
    cout<<"\n STUDENT ROLL NUMBER : "<<rno;
    cout<<"\n STUDENT AVERAGE MARKS : "<<avg;
}

void main()
```

```
{
   clrscr();
   student s1("Mudit", 151,59.5f);
   student s2("Anshul", 152);  // calling default parameter as third parameter
   s1.display();
   s2.display();
   getch();
}
```

Output:

```
Parameterised constructor called
Parameterised constructor called
                    Student record
STUDENT NAME  :  Mudit
STUDENT ROLL  NUMBER  :  151
STUDENT AVERAGE MARKS  :  59.5

                    Student record
STUDENT NAME  :  Anshul
STUDENT ROLL  NUMBER  :  152
STUDENT AVERAGE MARKS  :  0
```

In the above example, we have assigned 0.0f as default argument in parameterised constructor. In case of first object, average marks are passed during object creation so all 3 parameters will be sent to constructor definition.

In second object creation, we have passed only two parameters, so third value be considered as default argument in constructor.

Check Your Progress

In C++ ……. creates objects, even though it was not defined in the class.

6.8 NAMELESS OBJECTS

Nameless objects are also known as anonymous objects. These are the object which is declared without any name. Such objects are used to call constructor of that class for some processing. They may also be used as returning statement after updating object values.

Example 6.7 Program create anonymous object.

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<string.h>
class student
{
  private:
    char sname[50];
    int rno;
    float avg;
  public:
    student(char n[50], int r, float a);
    void display();
};

student::student(char n[50], int r, float a)   // parameterised constructor
{
 cout<<"\n Parameterised constructor called";
 strcpy(sname,n);
 rno=r;
 avg=a;
 display();
}
```

```
void student::display()
{
    cout<<"\n\t\t\tStudent record";
    cout<<"\n STUDENT NAME : "<<sname;
    cout<<"\n STUDENT ROLL NUMBER : "<<rno;
    cout<<"\n STUDENT AVERAGE MARKS : "<<avg;
}

void main()
{
   clrscr();
   student("Anshul", 152, 85.5f); //anonymous object
   getch();
}
```

Output:

Parameterised constructor called

Student record

STUDENT NAME : Anshul

STUDENT ROLL NUMBER : 152

STUDENT AVERAGE MARKS : 85.5

6.9 DYNAMIC INITIALIZATION THROUGH CONSTRUCTOR

Parameterised constructor may be literal (fixed) values at the time of object declaration. It may also be processed or user input values. In input or processed values case, such initialization will be known as dynamic initialization through constructor.

Example 6.8 Program create anonymous object

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

```
#include<conio.h>
#include<stdio.h>
#include<string.h>
class student
{
  private:
    char sname[50];
    int rno;
    float avg;
  public:
    student(char n[50], int r, float a);
    void display();
};
student::student(char n[50], int r, float a)   // parameterised constructor
{
 cout<<"\n Parameterised constructor called";
 strcpy(sname,n);
 rno=r;
 avg=a;
}

void student::display()
{
    cout<<"\n\t\t\tStudent record";
    cout<<"\n STUDENT NAME : "<<sname;
    cout<<"\n STUDENT ROLL NUMBER : "<<rno;
    cout<<"\n STUDENT AVERAGE MARKS : "<<avg;
}

void main()
```

```
{
    clrscr();
    char name[50];
    int rno;
    float avg;
    cout<<"\n ENTER STUDENT RECORD \n";
    cout<<" Enter Student name ";
    gets(name);
    cout<<" Enter Roll Number ";
    cin>>rno;
    cout<<" Enter average marks ";
    cin>>avg;
    student s(name,rno,avg);        // actual arguments for dynamic initialization
    s.display();
    getch();
}
```

Output:

```
ENTER STUDENT RECORD
Enter Student  name Mudit Kapoor
Enter Roll Number 101
Enter average marks 55.5

Parameterised constructor called
                                Student record
STUDENT NAME  :  Mudit Kapoor
STUDENT ROLL  NUMBER  :  101
STUDENT AVERAGE MARKS  :  55.5
```

In the above example, we have sent arguments which are not fixed at compile time. We may also do some calculations during assignment in constructor.

6.10 CONSTRUCTOR WITH DYNAMIC OPERATIONS

Dynamic constructors are those constructors which involves Dynamic initialization in their definition. For this we must have any pointer as data member in the class.

Suppose we want to store the marks in multiple subjects but we number of subjects are not fixed at compile time then we may use such dynamic constructor. These are most used in case of linked list, tree and graph related problems.

Example 6.9 Program to implement dynamic constructor

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<string.h>
class student
{
  private:
    char sname[50];
    int rno;
    int *marks;
    int subjects;
    float avg;
  public:
    student();          // Constructor definition
    ~student();       // Destructor definition
    void input();
    void calculate();
    void display();
};
```

```
student::student()   // Dynamic constructor definition
{
  cout<<"\n Dynamic constructor called";
  cout<<"\n\n Enter number of subjects ";
  cin>>subjects;
  marks=new int[subjects]; // runtime memory allocation
}

student::~student()      // destructor definition
{
  delete marks;            // runtime memory de-allocation
}

void student::input()
{
    cout<<"\n ENTER STUDENT NAME : ";
    gets(sname);
    cout<<"\n ENTER STUDENT ROLL NUMBER : ";
    cin>>rno;
    for(int i=0;i<subjects;i++)
    {
       cout<<"\t\tSubject "<<i+1<<" : ";
       cin>>*(marks+i);
    }
}

void student::calculate()
{
    int sum=0;
    for(int i=0;i<subjects;i++)
    {
```

```
        sum=sum+*(marks+i);
    }
    avg=(float)sum/subjects;
}

void student::display()
{
    cout<<"\n\t\t\tStudent record";
    cout<<"\n STUDENT NAME : "<<sname;
    cout<<"\n STUDENT ROLL NUMBER : "<<rno;
    cout<<"\n NUMBER OF SUBJECTS : "<<subjects;
    for(int i=0;i<subjects;i++)
    {
        cout<<"\n\t\tSubject "<<i+1<<" : "<<*(marks+i);
    }
    cout<<"\n STUDENT AVERAGE MARKS : "<<avg;
}
void main()
{
   clrscr();
   student s;
   s.input();
   s.calculate();
   s.display();
   getch();
}
```

Output:

Dynamic constructor called

Enter number of subject 4

```
ENTER STUDENT NAME :  Rohit Singh
ENTER STUDENT ROLL NUMBER  : 101
                    Subject 1 : 78
                    Subject 2 : 86
                    Subject 3 : 79
                    Subject 4 : 90

                        Student record
STUDENT NAME  :  Rohit Singh
STUDENT ROLL  NUMBER  :  101
NUMBER OF SUBJECT : 4
                    Subject 1 : 78
                    Subject 2 : 86
                    Subject 3 : 79
                    Subject 4 : 90
STUDENT AVERAGE MARKS  :  83.25
```

In the above example, we have allocated memory at runtime via constructor, so it is called dynamic constructor. One important thing is that, in such cases we should de-allocate memory through destructor.

Check Your Progress

Which type of construction of object s called in which allocation of memory to objects at the time of their construction?

6.11 CONSTANT OBJECT AND CONSTRUCTOR

Like constant variable, C++ also supports constant object. Once we have defined any object as constant, its values cannot be modified. Only constructor and destructor can modify the values of object, no other method can change the values of object. Constructors and destructors can never be declared as const. They are always allowed to modify an object even if the object is const.

If we have defined const object then they may call only const methods else the will give warning.

Example 6.10 Program to implement constant object

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<string.h>
class student
{
  private:
    char sname[50];
    int rno;
    float avg;
  public:
    student(char n[50], int r, float a);
    void display() const;       // declaring const method
};

student::student(char n[50], int r, float a)   // parameterised constructor
{
 cout<<"\n Parameterised constructor called";
 strcpy(sname,n);
 rno=r;
 avg=a;
}

void student::display() const // defining const method
{
    cout<<"\n\t\t\tStudent record";
    cout<<"\n STUDENT NAME : "<<sname;
    cout<<"\n STUDENT ROLL NUMBER : "<<rno;
```

```
    cout<<"\n STUDENT AVERAGE MARKS : "<<avg;
}

void main()
{
   clrscr();
   const student s("Nitin",101,60.5f);   // defining const object
   s.display();
   getch();
}
```

Output:

Parameterised constructor called

Student record

STUDENT NAME : Nitin

STUDENT ROLL NUMBER : 101

STUDENT AVERAGE MARKS : 60.5

In the above example, object is defined as const. so it cannot be updated, it can only be read or displayed.

Const Methods

First of all, important point is that const methods are used to be called by const object. Besides that const methods may be called by non-const methods. You can declare a method of a class to be const. This must be done both in the method's prototype and in its definition by coding the keyword const after the method's parameter list.

Check Your Progress

Whenever const objects try to invoke non-const member functions, the compiler_____

6.12 STATIC DATA MEMBERS WITH CONSTRUCTORS AND DESTRUCTORS

Data members and methods may be defined as static. In that case they will be called static data member and static method respectively. Constructor cannot be defined as static.

As we know static members may access only static members but non-static members may access static and non-static members both. Static data members are loaded into memory at the time of program loading and it is loaded only once.

Data members are declared as static to store common value for all objects. We can't declare static variables in a constructors.

Example 6.11 Program to count number of objects via static members

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<string.h>
class student
{
  private:
    char sname[50];
    int rno;
    float avg;
  public:
    static int count;              //static data member
    static void dispcount();     //static method
    student();
    ~student();
};

void student::dispcount()
```

```
{
  cout<<"\n Number of records till now : "<<count;  //display records count
}

student::student()
{
  strcpy(sname," ");
  rno=0;
  avg=0.0f;
  count++;  // increase in the counting of records
}

student::~student()
{
   count--;   // decrease in the count of records
}

int student::count=0;   //defining static data member

void main()
{
    clrscr();
    student s1;
    s1.dispcount();
    student s2[10];
    student::dispcount();
    student s3;
    s1.dispcount();
    getch();
}
```

Output :

Number of records till now : 1

Number of records till now : 11

Number of records till now : 12

In the above example, it is shown that static data members will store one value for all objects. And static methods may be called by object name and class name both.

Check Your Progress

What are the characteristics of static data member?

6.13 NESTED CLASSES

A nested class is a class which is declared in another enclosing class. A nested class is a member and as such has the same access rights as any other member. The members of an enclosing class have no special access to members of a nested class; the usual access rules shall be followed.

Example 6.12 Program to student record with his temporarily and permanent address

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<string.h>
class student
{
  private:
    char sname[50];
    int rno;
```

```
    float avg;
    class address   //nested class definition
    {
       int houseno;    //nested class data member
       char city[20];  //nested class data member
       public:
        void input()  //nested class method
        {
          cout<<"\n\t\t Enter house number ";
          cin>>houseno;
          cout<<"\t\t Enter city ";
          gets(city);
        }
        void display()  //nested class method
        {
          cout<<"\n\t\t House Number "<<houseno;
          cout<<"\n\t\t City "<<city;
        }
    };
    address temp;    //nested class object
    address permanent;   //nested class object
  public:
    void input();
    void display();
};

void student::input()
{
   cout<<"\n ENTER STUDENT RECORD \n";
   cout<<" Enter Student name ";
   gets(sname);
```

```
    cout<<" Enter Roll Number ";
    cin>>rno;
    cout<<" Enter average marks ";
    cin>>avg;
    cout<<" Enter temporarily address ";
    temp.input();                  //nested class's object method call
    cout<<" Enter permanent address ";
    permanent.input();             //nested class's object method call
  }

  void student::display()
  {
     cout<<"\n\t\t\tStudent record";
     cout<<"\n STUDENT NAME : "<<sname;
     cout<<"\n STUDENT ROLL NUMBER : "<<rno;
     cout<<"\n STUDENT AVERAGE MARKS : "<<avg;
     cout<<"\n TEMPORARILY ADDRESS : ";
     temp.display();                 //nested class's object method call
     cout<<"\n PERMANENT ADDRESS ";
     permanent.display();            //nested class's object method call
  }

  void main()
  {
          clrscr();
          student s1;
          s1.input();
          s1.display();
          getch();
}
```

Output:

```
ENTER STUDENT RECORD
Enter Student  name Amit
Enter Roll Number 101
Enter average marks 56.5
Enter temporarily address
                    Enter house number  56
                    Enter city Noida
Enter permanent address
                   Enter house number  445
                     Enter city Bareilly


                              Student record
STUDENT NAME  :  Amit
STUDENT ROLL  NUMBER  :  101
STUDENT AVERAGE MARKS  :  56.5
TEMPORARILY ADDRESS
                    HOUSE NUMBER  56
                    CITY Noida
PERMANENT ADDRESS
                   HOUSE NUMBER  445
                   CITY Bareilly
```

Nested Class Benefits

- Each individual class can be simple and straightforward.
- A class can focus on performing one specific task.
- The class is easier to write, debug, understand, and usable by other programmers.

Check your Progress

Can a class contain another class in it ?

6.14 SUMMARY

Constructor is a special method of the class which is used to initialize the data members of the class once the object is created. The most important thing with constructor is that it is executed implicitly for each object separately. There are several types of constructors: implicit, default, parameterised, copy and dynamic constructor. Like constructor allocates the memory, opposite to this, C++ provides destructor to destroy the objects. Dynamic constructors are used to allocate the memory in the constructor as well.

C++ allows the anonymous objects too. This may be handy in short notations for processing at times. C++ also supports static members to be managed in context of class while non-static members are managed in context of objects.

Sometimes problems are not too simple, they may be complicated. In that case we may need some objects as data members. One of the solution to that problem is nested class. With that, we may define class as member inside the other class. In that case, inside class is known as nested class. That nested class may be used only inside that parent class. Unlike containership, they cannot be used in outside the parent class.

6.15 EXERCISE

Q1. What is constructor? Describe their types with example.

Q2. What is destructor? Explain with example.

Q3. Explain the constructor overloading with example.

Q4. What is nested class? Describe with example.

Q5. Write a program in C++ to create the employee class with parameterised constructor. Employee class should contain Employee ID, address, department and salary.

Q6. Write a program in C++ to create the books class. It should contain ISBN, author, publication and price. Store 3 records of books by input. Load one more record with copy constructor.

DCECS-108/MCA-110/
PGDCA-110/BCA-109/
MCS-102

Uttar Pradesh Rajarshi Tandon Open University

C++ & OBJECT ORIENTED PROGRAMMING

BLOCK

3

OPERATOR OVERLOADING AND INHERITENCE

Course Design Committee

Prof. Ashutosh Gupta — **Chairman**
Director-In charge,
School of Computer and Information Science
UPRTOU, Prayagraj

Prof. U. S. Tiwari — **Member**
Dept. of Computer Science
IIIT Prayagraj

Prof. R. S. Yadav — **Member**
Department of Computer Science and Engineering
MNNIT-Allahabad, Prayagraj

Dr Marisha — **Member**
Assistant Professor (Computer Science)
School of Science, UPRTOU, Prayagraj

Mr. Manoj K. Balwant — **Member**
Assistant Professor (Computer Science)
School of Science, UPRTOU, Prayagraj

Course Preparation Committee

Dr Marisha — **Author (Block 1)**
Assistant Professor (Computer Science)
School of Science, UPRTOU,
Prayagraj

Er. Pooja Yadav — **Author (Blocks 2, 3 and 4)**
Assistant Professor
Dept. of Computer Science and IT
M.J.P. Rohilkhand University, Bareilly

Prof. Ashutosh Gupta — **Editor**
Director (In-Charge)
School of Computer and Information Science
UPRTOU, Prayagraj

Dr Marisha — **Coordinator**
Assistant Professor (Computer Science)
School of Science, UPRTOU, Prayagraj

ISBN : 978-93-83328-98-7

Printed and Published by Dr. Arun Kumar Gupta Registrar, Uttar Pradesh Rajarshi Tandon Open University, 2020.

DCECS-108/182

Printed By : Chandrakala Universal Pvt. 42/7 Jawahar Lal Neharu Road, Prayagraj.

BLOCK INTRODUCTION

In this section we discuss the overview of this block's content. This block consists of the following units:

Unit-7 : Operator Overloading

In this unit we'll discuss about the concept and working of polymorphism specially compile time polymorphism. Operator Overloading is one of important concept which makes user defined data types usable to perform variety of arithmetic, logical and much more operations.

Unit-8 : Inheritance

In this unit we'll discuss each and everything about inheritance in C++, specifically, what is inheritance and different ways to implement it types of inheritance, containership and constructor and destructor in inheritance with examples. Inheritance is one of the important concepts of C++. It permits user to create a new class (derived class) from an existing class (base class). The derived class inherits all the features from the base class and can have additional features of its own.

UNIT-7 OPERATOR OVERLOADING

7.0 INTRODUCTION

Operator overloading is one of the most important concept of object oriented programing. It is a type of polymorphism, in which an operator is overloaded to give user defined meaning to it. Operators are used to perform operations on built-indata type like c=a+b, but can't perform operations on user define data type's variable like obj3=obj1+obj2. This is the main reason of origin of operator overloading concept.

7.1 OBJECTIVE

The main objective of this unit is to define the main goals of operator overloading is to make their code cheaper to write and easier to understand. Operator overloading permits to redefine the working of operator for user-defined types (objects, structures) without changing the definition. It cannot be used for built-in types (int, float, char etc.).

7.2 DEFINITION

Operator overloading is a type of Compile time binding. It provide extra functionality to the existing operators so that they also operate on user defined data type's variable i.e. objects. Like '+' operator can be overloaded to perform addition on built-in data types, like Integer, float, String (concatenation) etc. But this concept that allows programmer to

redefine the importance of operator when they operate on class objects is known as operator overloading.

One of the difference between structure and class is:

```
typedef struct student {
int a,b;
}S1,S2,S3;
```

We can't do

```
S3=S1+S2;
```

but

```
class employee {
int a,b;
}e1,e2,e3;
```

We can do:

```
e3=e1+e2;
```

with the help of operator overloading concept.

Check Your Progress

Which type of polymorphism is Operator Overloading?

7.3 METHODS OF OPERATOR OVERLOADING

There are two methods to perform Operator Overloading one is by using member function and another is friend function.

There are 3 types of operators:

- Unary
- Binary
- Ternary

But we can't overload ternary operator and to overload unary operator by using member function no argument is passed but by using friend function

one argument is passed. And to overload binary operator by using member function one argument is passed and by using friend function two arguments are passed.

Method /Operator	Unary	Binary
Member Function (No. of argument =operand use -1)	0	1
Friend Function (No. of argument =operand)	1	2

7.4 OVERLOAD UNARY OPERATOR BY USING MEMBER FUNCTION

Syntax : The general syntax to overload unary operator by using member function is as follows:

```
return-type operator operator _to _be _overloaded( List of Arguments)
{
statement;
statement ;
return(expression); // it will not applicable if return type is void
}
```

operator is the keyword in C++, which indicate that **operator _to _be _overloaded** is an operator.

Example 7.1 To overload pre increment ++ operator

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

```
class sum
{
int a;
int b;
public:
void getdata()
{
 cout<< " enter the value of a and b:";
cin>>a>>b;
}
void display()
{
 cout<< "\na="<<a<<"\nb="<<b;
}
void operator++();
};

void sum::operator++()
{
a=++a;
b=++b;
}
void main( )
{
clrscr();
sum s;
s.getdata( );
s. operator ++();
s.display();
getch();
```

```
}
```

Output:

enter the value of a and b: 5 8

a=6

b=9

Check Your Progress

How to declare operator function?

7.5 OVERLOAD UNARY OPERATOR BY USING FRIEND FUNCTION

Syntax : The general syntax to overload unary operator by using member function is asfollows:

```
friend return-type  operator operator_to _be _overloaded ( List of Argument )
{
        statement;
        statement;
        return (expression); // it will not applicable if return type is void
}
```

friend and operator is the keyword in C++, which indicate that **operator _to _be _overloaded** is an operator which is overloaded by friend function.

Example 7.2 To overload pre increment ++ operator

```
#include<conio.h>
```

```
#include<iostream.h>
class sum
{
int a,b;
public:
void getdata()
{
   cout<< " enter the value of a and b:";
   cin>>a>>b;
}
void display( )
{
   cout<< "\na="<<a<<" \nb="<<b;
}
friend void operator++(sum &);  // ++ operator as friend function declaration
};
void operator++(sum &ob)  // ++ operator as friend function definition
{
ob.a=++ob.a;
ob.b=++ob.b;
}

void main( )
{
sum s;
clrscr();
s. getdata( );
operator ++(s);       //or  we can use   ++s;
s. display();
getch();
```

```
}
```

Output:

```
enter the value of a and b: 5  8
a=6
b=9
```

In this program we pass reference of object as an argument because friend function is not call with object so object is passed to access the member of class and if increment is perform in local area then it will never reflect in outside function but reference variable is the alias name so the reflection will be seen in outside function.

Check Your Progress

What is unary operator? How many arguments are passed in friend function to overload unary operator.

7.6 OVERLOAD BINARY OPERATOR BY USING MEMBER FUNCTION

Syntax : The general syntax to overload unary operator by using member function is as follows:

```
return-type  operator operator _to _be _overloaded ( List of Argument )
{
    statement;
    statement;
    return(expression); // it will not applicable if return type is void
}
```

operator is the keyword in C++, which indicate that **operator _to _be _overloaded** is an operator.

Example 7.3 To overload pre increment + operator

```
#include<conio.h>
#include<iostream.h>
class sum
{
int a;
int b;
public:
void getdata()
{
 cout<< " enter the value of a and b:";
cin>>a>>b;
}
void display()
{
 cout<< "\na="<<a<<" \nb="<<b;
}
void operator + (sum );
};

void sum::operator+(sum s1)
{
a=a+s1.a;
b=b+s1.b;
}
void main( )
{
clrscr();
sum ob1,ob2;
ob1.getdata( );
```

```
ob2.getdata();
ob1.operator +(ob2);
ob1. display();
getch();
}
```

Output:

enter the value of a and b: 4 5

enter the value of a and b: 7 8

a=11

b=13

Check Your Progress

In which type of operator overloading through a member function take one explicit argument?

7.7 OVERLOAD BINARY OPERATOR BY USING FRIEND FUNCTION

Syntax : The general syntax to overload unary operator by using member function is as follows :

```
return-type  operator operator _to _be _overloaded ( List of Argument )
{
    statement;
    statement;
    return(expression); // it will not applicable if return type is void
}
```


operator is the keyword in C++, which indicate that **operator _to _be _overloaded** is an operator.

In case of binary operator overloading by using member function it is compulsory to present object on R.H.S because i.e.

X+Y which is equivalent to **X . operator + (Y)** , where X and Y plays the role of object

So in this case 2+Y is not possible where Y is object and 2 is not a object so it can't be accessible operator function. This statement looks like **2.+operator (Y)** . This can't do .

By which we used friend function to overload binary operator in which we pass both operand as argument. Like: **operator + (X,Y)**

And **operator +(2,Y)**

Example 7.4 To overload pre increment + operator

```
#include<conio.h>
#include<iostream.h>
class sum
{
int a;
int b;
public:
void getdata()
{
 cout<< " enter the value of a and b:";
cin>>a>>b;
}
void display()
{
 cout<< "\na="<<a<<" \nb="<<b;
}
friend sum operator + (sum, sum );
};
```

```
sumoperator + (sum s1,sum s2)
{
sum s3;
s3.a= s1.a+s2.a;
s3.b=s1.b+s2.b;
return(s3);
}
void main( )
{
clrscr();
sum ob1,ob2,ob3;
ob1.getdata( );
ob2.getdata();
ob3=ob1+ob2;
ob3. display();
getch();
}
```

Output:

```
enter the value of a and b : 4   5
enter the value of a and b : 7   8
a=11
b =13
```

In this, ob3=ob1+ob2 can also be used as ob3=operator (ob1, ob2); both are equivalent.

Check Your Progress

In Which type of operator overloading through a member function, the left hand operand must be an object of the relevant class.

7.8 RULES FOR OVERLOADING OPERATOR

(i) Only built-in or existing operators can be overload, not built –in data type.

(ii) Syntax and Associativity can't be change

(iii) Operators must be overloaded explicitly.

(iv) Definition or meaning of any operand should not be change. If change then this code is confusing and, difficult to understand and debug.

(v) Two methods i.e. by using member function and by using friend function are used to overload any operator.

(vi) Some operators cannot be overload by using friend function and even some cannot be overload by any of method.

(vii) List of operators that can be overload:

(viii) Following is the list of operators, which cannot be overloaded:

Operators that can be overloaded							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	= =	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

(ix) Following is the list of operators, which cannot be overloaded :

Operators that cannot be overloaded				
.	.*	::	?:	sizeof

(x) There are some operators that can be overload by using member function but can't be overload by using friend function.

Assignment operator =
function call operator ()
subscriping operator []
class member access operator ->

7.9 TYPE CONVERSION AND ITS METHOD

The process of converting a value from one data type to another is called a **type conversion**. Type conversion is of two types: implicit conversion and explicit conversion. Implicit conversion is known as automatic type conversion where the compiler automatically transforms one fundamental data type into another.

Ex:- int x=5;

float z;

z=x;

But compiler can't convert class type in to built-in type and also can't convert in to another user defined data type. This can be done only by using casting operator.

This is known asexplicit type conversions, where the user uses a some extra code forconversion and explicit conversion is known as type casting.

There are three types situation where data type conversions are between incompatible types:-

- Basic to class type
- Class type to Basic type
- One Class type to another class type

Method :

One method is by using constructor and another is casting operator

```
Constructor_name( typename  variable)
{……
}
```

There are 3 condition of casting operator those should be satisfied-

1) It must be a class member.
2) It must not specify a return type.
3) It must not have any arguments.

Since, it is a member, it is invoked by the object and therefore, the values used for conversion inside the function belong to the object that invoked the function. As a result function does not need an argument.

Syntax :

```
operator typename( )
{
 statement
 ……
return( variable of type-name);
}
```

The above function converts a class type data to type-name.

7.9.1 BASIC TO CLASS TYPE CONVERSION

Example 7.5 Convert basic data type to class type data.

```
#include<conio.h>
#include<iostream.h>
class sum
{
int a;
intb;
public:
sum()
{
}
```

```
sum( int t)
{
a=t;
b=t+1;
}
void display()
{
cout<< "\na="<<a;
cout<<"\nb="<<b;
}
};
void main( )
{
sum s;
clrscr();
int num=5;
s=num;
s. display();
getch();
}
```

Output:

a=5

b=6

Check Your Progress

To perform the conversion from any other data type or class to a class type, a …….. should be used in the destination class.

7.9.2 CLASS TO BASIC TYPE CONVERSION

Example 7.6 Convert class type data to basic data type

```
#include<conio.h>
#include<iostream.h>
class sum
{
public:
int a;
int b;
sum(int x, int y)
{
        a=x;
        b=y;
}
operator int();
};
sum :: operator int()
{
return int(a+b);
}
void main()
{
clrscr();
int add;
sum s(10,20);
add=s;
cout<<"add ="<<add;
getch();
}
```

Output:

add=30

Check Your Progress

The conversion from a class to any other type or any other class is done by using …….. in the source class.

7.9.3 CLASS TO CLASS TYPE CONVERSION

Example 7.7 Convert one class type data to another class type data

```
#include<conio.h>
#include<iostream.h>
class sub
{
public:
int a1;
int b1;
void display()
{
 cout<<"\n a1 ="<<a1;
 cout<<"\n b1 ="<<b1;
}
};
class sum
{
public:
int a;
int b;
sum(int x,int y)
{
  a=x;
```

```
   b=y;
  }
  operator sub() //type conversion by using  casting operator
  {
      sub s;
      s.a1=a;
                    s.b1=b;
                    return s;
                }
          };

  void main()
  {
  clrscr();
  sum ob1(10,20);
  sub ob2;
  ob2=ob1;  //means ob2(ob1)
  ob2.display();
  getch();
  }
```

Output:

a1=10

b1=20

Method-II :

Example 7.8 Convert one class type data to another class type data

```
#include<conio.h>
#include<iostream.h>
class sum
{
```

```
public:
int a,b;
sum(int x,int y)
{
        a=x;
        b=y;
}
};

class sub
{
public:
int a1,b1;
sub()
{

}
sub(sum s) //type conversion by using  constructor
{
 a1=s.a;
b1=s.b;
}
void display()
{
        cout<<"a1 ="<<a1;
        cout<<"\nb1 ="<<b1;
}
};
void main()
{
clrscr();
sum ob1(10,20);
sub ob2;
```

```
  ob2=ob1;  //means ob2(ob1)
ob2.display();
getch();
}
```

Output:

a1=10

b1=20

Check Your Progress

How many parameters does a conversion operator may take?

7.10 SUMMARY

Operator overloading is one of the most helpful concept that creates new way of working for objects without changing its syntax, meaning, precedence and associativity with the help of Operator keyword in any of the method: by using member function and friend function.

In this three type of type conversion is performed: basic to class, class to basic and class to class with the help of constructor and type cast operator.

7.11 EXERCISE

1. What is operator overloading? Give some limitations of operator overloading.
2. Write a program to overload a '<' operator by using friend function.
3. What is operator overloading explain its properties? Write limitation of member function in binary Operator overloading.
4. List out the operators that cannot be overloaded.
5. What is the purpose of using operator function? Write its syntax.
6. What is the need of friend function, when member function is already use in operator overloading.

7. Class Distance consists of length in feet and inches. Class Distance contains

   i) One default constructor

   ii) One parameterized constructor

   iii) Function getdata() to take the value of feet and inches.

   iv) Function show() to display.

      a) Overload < operator to compare two distances.

      b) Overload += operator in the Distance class.

8. Write a program to-

   a) Overload ++ pre increment operator (++c1) and – post decrement operator (c2--) in class**.**

   b) Overload == operator to compare two value.

9. Add two complex number by overloading + operator

   a) Using Member function.

   b) Using Friend Function.

UNIT-8 INHERITANCE : EXTENDING CLASSES

8.1 INTRODUCTION

One of the most important concepts in object-oriented programming is inheritance. Inheritance allows us to derive a new class from old class which inherit the properties of old class.

8.2 OBJECTIVE

It provides the concept of reusability of code. And can easily debug error and duplicate codes. Because one class inherit all the features of existing class and add a new feature so this makes code cleaner, understandable and extendable.

8.3 DEFINITION

Inheritance is one of the key features of Object-Oriented Programming in C++. Inheritance is the property in which new classes can create from an existing class. A class can be derived from one/more class, which means it can inherit data and functions from base classes and can have additional features of its own.

New class is called derive class or child class or sub class and old class is called base class or parent class or super class.

Syntax : The general syntax of inheritance is as follows:

```
class Derive _class : Access Specifier Base_class
{……..
statements
…….
}
```

Where access-specifier is of public, protected, or private, and base-class is the name of an old defined class and Derived class is the name of new class. If the access-specifier is not used, then it is private by default.

8.4 ACCESS SPECIFIER

During creation of derived class from a base class, different access specifiers / visibility mode are used to inherit the data members of the base class. There are three access specifiers-

- Public
- Private
- Protected

The data which is mention in public visibility mode can be accessible by anywhere inside as well as outside the class but with the help of associated object. The data which is mention in private visibility mode can be used with in associated member function but can't be accessible outside the class, even with the help of associated object. The data which is mention in protected visibility mode can be accessible by inside as well as outside the class but only in just next derived class. But even not accessible with the help of associated object.

Check Your Progress

What is the difference between protected and private access specifiers in inheritance?

8.5 TYPES OF DERIVATION

Derive class is derived from base class and this process is called derivation. Derivation is of three types:

1. **Public Derivation :** When new class derives from old class in public mode. Then the public member of the old class will become public in the new class and protected members of the old class will become protected in new class and private members of the old class will become private in new class
2. **Protected Derivation** : When new class derives from old class in protected mode. Then the public member of the old class will become protected in the new class and protected members of the old class will become protected in new class and private members of the old class will become private in new class
3. **Private Derivation :** When new class derives from old class in private mode. Then every members of the old class will become Private in new class.

Base Class Access Specifier	**Types of Derivation**		
	Public	**Protected**	**Private**
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not Accessible (Hidden)	Not Accessible (Hidden)	Not Accessible (Hidden)

8.6 TYPES OF INHERITANCE

Inheritance is of five types :

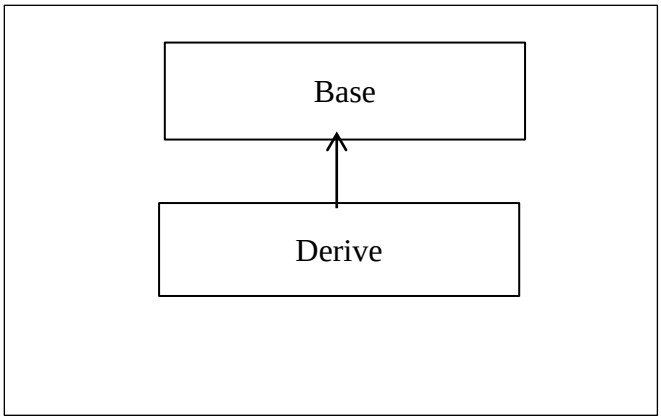
1) Single Inheritance
2) Multilevel Inheritance
3) Multiple Inheritance

4) Hierarchical Inheritance

5) Hybrid Inheritance

8.6.1 SINGLE INHERITANCE

When one derive class is inherit from one base class, then this type of inheritance is known as *Single Inheritance.*



Syntax : The general syntax of Single Inheritance is as follows:

```
class Base
{
….
…..
};
class Derive : access_specifier Base
{…
…
};
```

Example 8.1: Program for Single Inheritance
#include<iostream.h>

```
#include<conio.h>
class abc
{
int a;      //a is private so it cannot be inherited
public:
int b;
abc()
{
a=1;
b=2;
}  //constructor is used for initialization
void display_a()
{
cout<<"a="<<a;
}
int get_a()
{
return a;
} //a is accessible to its member function
};

class xyz : public abc
{
int x;
public:
        int y;
        xyz(){x=3;y=4;}
        void display()
        {
display_a();
/* here display_a() is used bcoz a is private member and it can only be
```

```
accessed through member function of class abc. */

cout<<"\nb="<<b;
cout<<"\nx="<<x;
cout<<"\ny="<<y;
        }
        void add()
        {
        int n;
        n=get_a()+b+x+y;
        cout<<"\naddition="<<n;
}
};

int main()
{
clrscr ();
abc ob1;
xyz ob2;
ob2.display();
ob2.add();
getch();
return 0;
}
```

Output:

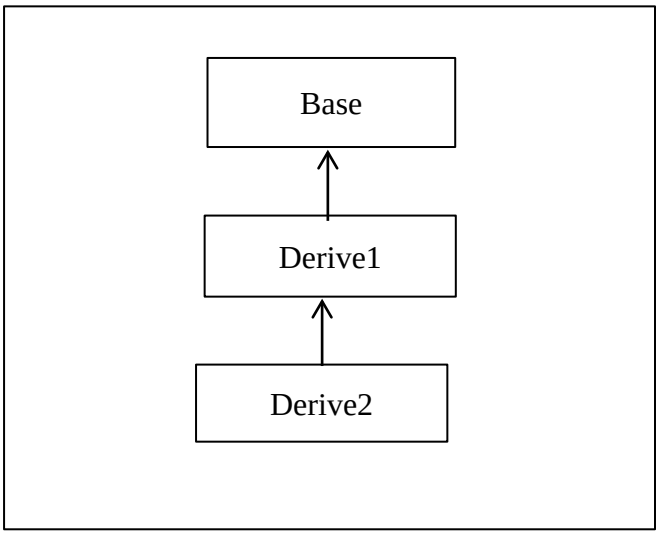
```
a=1
b=2
x=3
y=4
addition=10
```

Check Your Progress

The ……. inherits some or all of the properties of the …….. class.

8.6.2 MULTILEVEL INHERITANCE

When derived class inherits from the base class and another derived class inherits from this derived class, this form of inheritance is known as *Multilevel Inheritance*. Because more than one classes are derived at different level.



Syntax: The general syntax of Multilevel inheritanceis as follows:

```
class Base
{
….
…..
};
```

```
class Derive 1 : access specifier Base
{…
…
};

class Derive 2 : access specifier Derive 1
{…
…
};
```

Example 8.2 : Program for Multilevel Inheritance

```
#include<iostream.h>
#include<conio.h>
class abc
{
int a;      //a is private so it cannot be inherited
protected:
int b;
public:
        int get_a()
        {
        cout<<"enter the value of a=";
        cin>> a;
        return a;
        } //a is accessible to its member function
};

class xyz : public abc
```

```
{
int x,y;
public:
      int get_x()
      {
cout<<"enter the value of y=";
  cin>> y;
   b=2;
//protected data can be used either its original or its just next derive class
     x=y;
    return(x);
      }
};

class pqr: private xyz
{
int n;
public:
      void add()
      {
n=get_a()+get_x();
      }
      void display()
      {
cout<<"\naddition of two no. is="<<n;
      }
};

void main()
{
clrscr();
```

```
pqr ob;
ob.add();
ob.display();
getch();
}
```

Output:

enter the value of a=2

enter the value of y=3

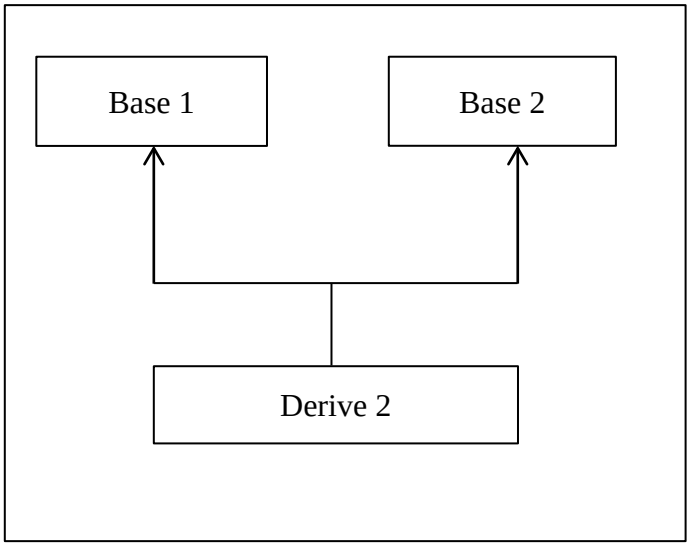
addition is =5

Check Your Progress

Which Class is having highest degree of abstraction in multilevel inheritance of 5 levels?

8.6.3 MULTIPLE INHERITANCE

When a derive class inherit from more than one base class than this type of inheritance is known as *Multiple Inheritance.*



Syntax : The general syntax of multiple inheritanceis as follows:

```
class Base 1
{
….
…..
};

class Base 2
{…
…
};

class Derive : access specifier Base 1 , access specifier Base 2
{
…
};
```

Example 8.3: Program for Multiple Inheritance

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

class abc
{
int a;      //a is private so it cannot be inherited
public:
        int get_a()
        {
        cout<<"enter the value of a=";
```

```
        cin>> a;
        return a;
        } //a is accessible to its member function
};

class xyz
{
int x;
public:
      int get_x()
      {
 cout<<"enter the value of x=";
 cin>> x;
  return(x);
      }
};

class pqr: protected abc, private xyz
{
int n;
public:
      void add()
      {
n=get_a()+get_x();
      }
      void display()
      {
  cout<<"\naddition of two no. is="<<n;
      }
};
```

```
void main()
{
clrscr();
pqr ob;
ob.add();
ob.display();
getch();
}
```

Output:

enter the value of a =2

enter the value of x=3

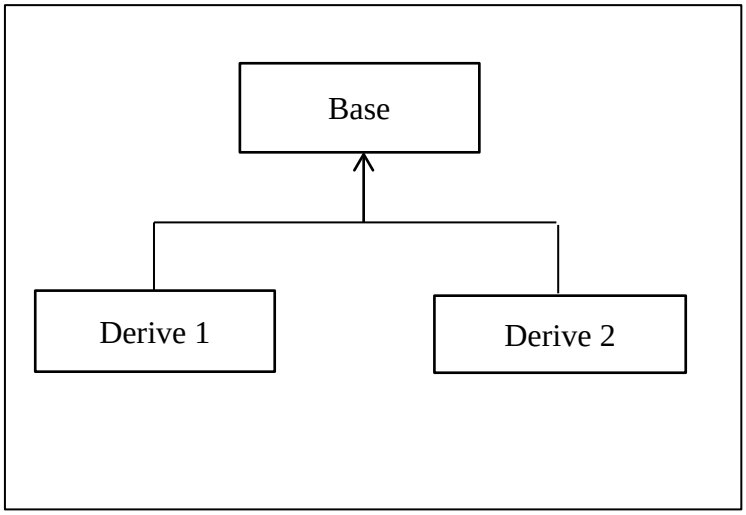
addition of two no. is =5

Check Your Progress

What is the syntax of multiple inheritance?

8.6.4 HIERARCHICAL INHERITANCE

When more than one derive class inherit from one base class. This form of inheritance is known as *Hierarchical Inheritance*.



Syntax : The general syntax of hierarchichal inheritanceis as follows:

```
class Base
{
….
…..
};

class Derive 1 : access specifier Base
{
…
…
};

class Derive 2 : access specifier Base
{
…
…
};
```

Example 8.4 : Program for Hierarchical Inheritance

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

class abc
{
int a;      //a is private so it cannot be inherited
public:
        int get_a()
```

```
        {
        cout<<"\nenter the value of a=";
        cin>> a;
        return a;
        } //a is accessible to its member function
};

class xyz : private abc
{
int x,m;
public:
    int get_x()
    {
cout<<"enter the value of x=";
cin>> x;
return(x);
    }
    void add()
    {
m=get_a()+get_x();
    }
    void show()
    {
cout<<"\naddition of two no. is="<<m;
    }
};
class pqr : protected abc
{
int n,p;
public:
    int get_p()
```

```
      {
cout<<"enter the value of p=";
cin>> p;
      return(p);
      }
      void sub()
      {
   n=get_a()-get_p();
      }
      void display()
      {
 cout<<"\nsubtraction of two no. is="<<n;
      }
};

void main()
{
clrscr();
xyz obx;
obx.add();
obx.show();
pqr obp;
obp.sub();
obp.display();
getch();
}
```

Output :

```
enter the value of a=2
enter the value of x=4
addition of two no. is =6
```

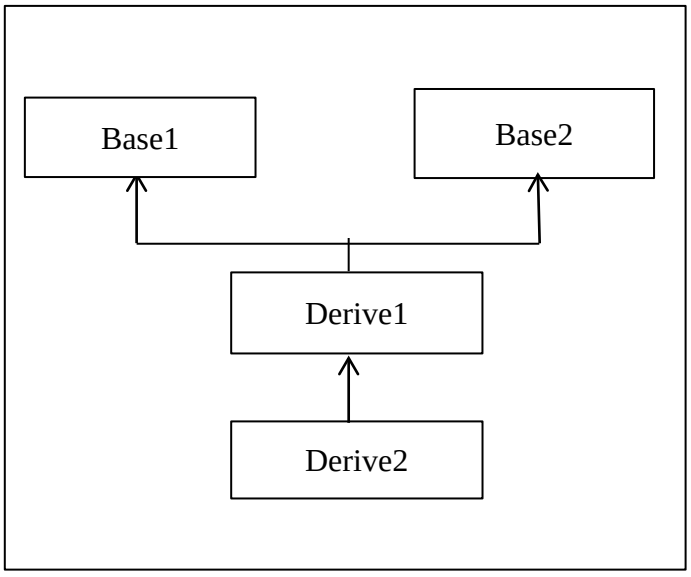
```
enter the value of a=9
enter the value of x=7
addition of two no. is =2
```

Check Your Progress

When a child class inherits traits from more than one parent class, this type of inheritance is called _____ inheritance.

8.6.5 HYBRID INHERITANCE

When derive class inherit from the base class with the combination of two or more different type of inheritance. This form of inheritance is known as *Hybrid Inheritance*.



Syntax : The general syntax of hybrid inheritanceis as follows :

```
class Base 1
{
    …
```

```
    …
};

class Base 2
{
    …
    …
};

class Derive 1 : access specifier Base1, access specifier Base2
{
    …
    …
};

class Derive 2 : access specifier Derive 1
{
    …
    …
};
```

Example 8.5 : Program for Hybrid Inheritance

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

class abc
{
int a;      //a is private so it cannot be inherited
```

```
public:
        int get_a()
        {
        cout<<"\nenter the value of a=";
        cin>> a;
        return a;
        } //a is accessible to its member function
};

class xyz
{
int x;
public:
      int get_x()
      {
   cout<<"\nenter the value of x=";
 cin>> x;
 return(x);
      }
};

class pqr: protected abc, private xyz
{
int n;
public:
      void add()
      {
  n=get_a()+get_x();
      }
      int display()
      {
```

```
  cout<<"\naddition of two no. is="<<n;
  return(n);
      }
};

class rst: private pqr
{
float avg;
public:
void show()
{
  add();
    avg=display()/2;
    cout<<"\naverage of two no. is="<<avg;
}
};

void main()
{
clrscr();
rst ob;
ob.show();
getch();
}
```

Output:

```
enter the value of a =3
enter the value of x=4
addition of two no. is =7
average of two no. is =3
```

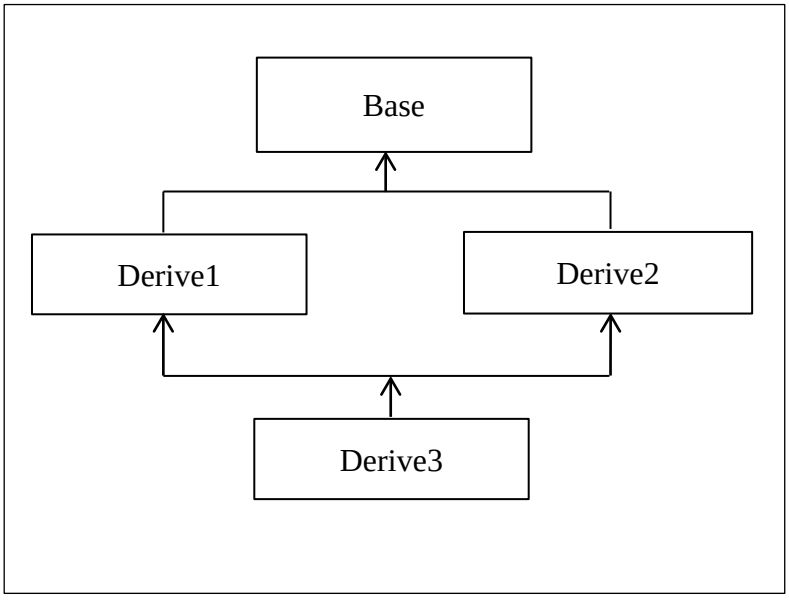
Check Your Progress

How many types of inheritance should be used for hybrid ?

8.6.5.1 VIRTUAL BASE CLASS

When derive class inherit from the base class in the implementation of combination of hierarchical inheritance and multiple inheritance. Then in this type of inheritance an ambiguity can arise due to multiple paths exist from the same base class to a derive class. This means that a child class could have duplicate sets or more than one copy of members inherited from a single base class. This problem is solved using virtual base class. In this case when derive class is derived from base class then we use virtual keyword either before the access specifier or after access specifier (because place of virtual keyword is optional) and that base class is known as virtual base class. When any class is made virtual, so that the duplication is discarded even number of paths exist to the child class.

Suppose you have two derived classes B and C that have a common base class A, and you also have another class D that inherits from B and C. You can declare the base class A as *virtual* to ensure that B and C share the same sub object of A.



Syntax : The general syntax of virtual base classis as follows:

```
class Base
{
….
…..
};

class Derive 1 :  Virtual access specifier Base
{
…
…
};

class Derive 2 : access specifier virtual  Base
{
…
…
};

class Derive 3: access specifier  Derive 1, access specifier Derive 1
{
….
…..
};
```

Example 8.6 : Program for Virtual Base Class

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

```
        class abc
        {
            int a;
            public:
            abc()
            {
            a=1;
            }
            get_a()
{
return a;
}
/*
 void display()
{
cout<<"\na="<<a;
}
*/
};

class xyz : virtual public abc
{
int x;
public:
xyz()
{
x=2;
}
get_x()
{
```

```
return x;
}
/*
 void display()
{
  cout<<"\nb1="<<b1;
}
*/
};

class pqr : virtual public abc
{
int p;
public:
pqr()
{
p=3;
}
void get_p()
{
return p;
}
/*
 void display()
{
cout<<"\nb2="<<b2;
}
*/
};
```

```
class rst : public xyz , public pqr
{
int r;
        public:
        void add()
        {
        r=get_a()+get_x()+get_p();
        }
        void display()
        {
//A::display();             // ambiguity resolution
//B1::display();
//B2::display();
cout<<"\naddition of three numbers are="<<r;
        }
};

int main()
{
clrscr();
rst ob;
ob.add();
ob.display();
getch();
return 0;
}
```

Output:

addition of three numbers are =6

Check Your Progress

In Multi-path inheritance by which concept we can remove duplicate set of records in child class?

8.7 ABSTRACT CLASS

An *abstract class* is a class that is designed to be specifically used as a base class. And no object is created for this base class and no non-static data members of an abstract class can be declared.

8.8 DATA MEMBER INITIALISATION

Initializer List is used to initialize data members of a class. The member initializer list is inserted after the constructor name. It begins with a colon (:), and then lists each variable to initialize along with the value for that variable separated by a comma.

Syntax : The general syntax of virtual base classis as follows:

```
Constructor(argument list ): initialisation section
{
argument section
}
```

Use of initialization list: Initialize the member of the class :

1. abc(int i , int j): a(i), b(j) { } // if i=1 & j=2 then a=1 & b=2
2. abc(int i , int j): b(i), a(j) { } // if i=1 & j=2 then b=1 & a=2
3. abc(int i , int j): a(i), b(a+j) {} // if i=1 & j=2 then a=1 & b=3
4. abc(int i , int j): b(i), a(b+j) {} //invalid
5. abc(int i , int j): a(i), {b=j;}

Example 8.7 : Program of Initialization list and constructor

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

```
#include<conio.h>


class abc
{
   int a;
   int b;
public:
abc(int i , int j ): a(i), b(j)
{}
/*
use of Initializer list is optional as the  constructor can also be written as:
abc(int i, int j)
{
  a = i;
  b = j;
}
*/
  void display()
{
cout<<"\na="<<a<<"\nb="<<b;
}
};


void main()
{
clrscr();
abc oba (10, 15);
oba.display();
getch();
}
```

Output:

a=10

b=15

8.9 CONSTRUCTOR AND DESTRUCTOR IN DERIVED CLASS

As earlier discuss that constructor plays a vital role in initializing an object. Constructors cannot inherit, while applying inheritance; we usually create objects of derived class. But, if a base class has a parameterised constructor with one or more parameter, then value of these parameters are passed with the help of initialiser list, which is associated in derived class constructor so it is compulsory to have a constructor in derived class and pass the arguments to the base class constructor. When both the base class and derived class have constructors, the base class constructor is executed first and then the constructor in the derived class is executed.

The syntax of the derived class constructor contains two parts separated by a colon (:). The first part defines the list of arguments that are passed to the derived class constructor and the second part defines the base class constructor call.

The order of exectution of constructor :

S.No	Inheritance	Syntax	Order
1.	Single	class Base {….}; class Derive: public Base{….};	first Base then Derive
2.	Multilevel	class Base {….}; class Derive: public Base{…}; class Derive1: public Derive{…};	first Base then Derive then Derive1
3.	Multiple	class Base1 {….}; class Base2{….}; class Derive: public	first Base2 then Base1 then Derive1

		Base2, public Base1{…};	• Depend on the order in which they appear in the declaration of the derived class derivation and at last derive class
4.	Hybrid	class Base1 {….}; class Base2{….}; class Derive: public Base2, virtual public Base1{…};	first Base1 then Base2 then Derive1 • First priority given to virtual base class then Depend on the order in which they appear in the declaration of the derived class derivation and at last derive class

In Inheritance each destructor is called in the *reverse* order of calling of constructor. i.e. when derive class is destroyed, the derive destructor is called first, then the destructor of base class is called.

Syntax : The general syntax of constructor of derived class is as follows:

```
class Base
{
….
 public:
 Base(..)
 {
  ….
 }
};
```

```
class Derive1 : access specifier Base
{
  …
  public:
  Derive1(….):Base (..)
  {
     …
   }
};

void main()
{
     Derive1 obd(…);
     ….
     …..
     ….
}
```

Example 8.8 : Program of Constructor and Destructor in a derive class

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

class abc
{
int a,b;
public:
abc(int a1,int b1)
{
a=a1;
```

```
b=b1;
}
void disp()
{
cout<<"\na="<<a<<"\nb="<<b;
}
};

class xyz :  public abc
{
int x,y;
public:
      xyz(int m,int n,int x1,int y1):abc(m,n)
      {
           x=x1;y=y1;
      }
     void display()
{
cout<<"\nx="<<x<<"\ny="<<y;
}
};

int main()
{
clrscr();
xyz ob(5,7,3,4);
ob.disp();
ob.display();
getch();
return 0;
}
```

Output:

a=5

b=7

x=3

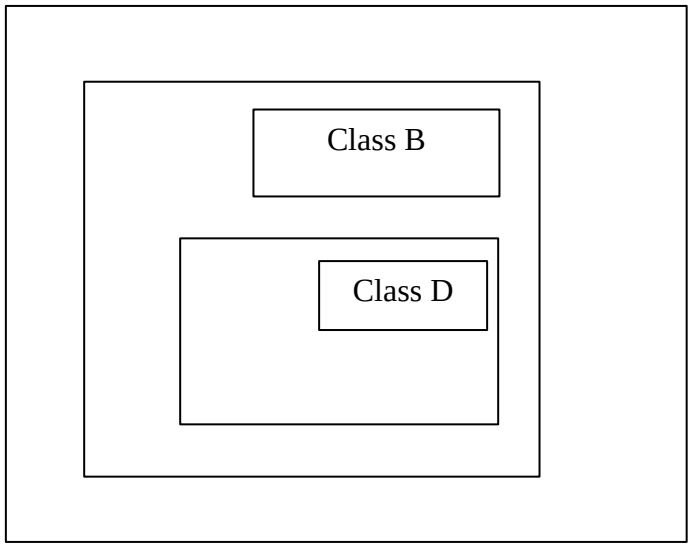
y=4

Check Your Progress

Can we pass parameters to base class constructor though derived class or derived class constructor?

8.10 CONTAINERSHIP

When a class contains an object of a different class as a member data. Ex: class A could contain an object of class B, class C and class C contain an object of class D as amember. Class A becomes the container, while class B becomes the contained class. Then all the public data member and member functions(only) defined in contained class can be executed in container class. Container ship is the substitute of inheritance. In OOPs inheritance represents an “is-a” relationship, but Containership represents a “has-a” relationship.



Example 8.9 : Program for Containership

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

class abc
{
int a;      //a is private so it cannot be inherited
public:
          int get_a()
          {
            cout<<"enter the value of a=";
            cin>> a;
            return a;
          } //a is accessible to its member function
};

class xyz
{
int x;
public:
      int get_x()
      {
        cout<<"enter the value of x=";
        cin>> x;
        return(x);
      }
};

class pqr
{
```

```
 int n;
 abc oba;
 xyz obx;
public:
      void add()
      {
        n=oba.get_a()+obx.get_x();
      }
      void display()
      {
        cout<<"\naddition of two no. is="<<n;
      }
};
void main()
{
clrscr();
pqr ob;
ob.add();
ob.display();
getch();
}
```

Output:

```
enter the value of a=3
enter the value of x =4
addition of two no. is =7
```

Check Your Progress

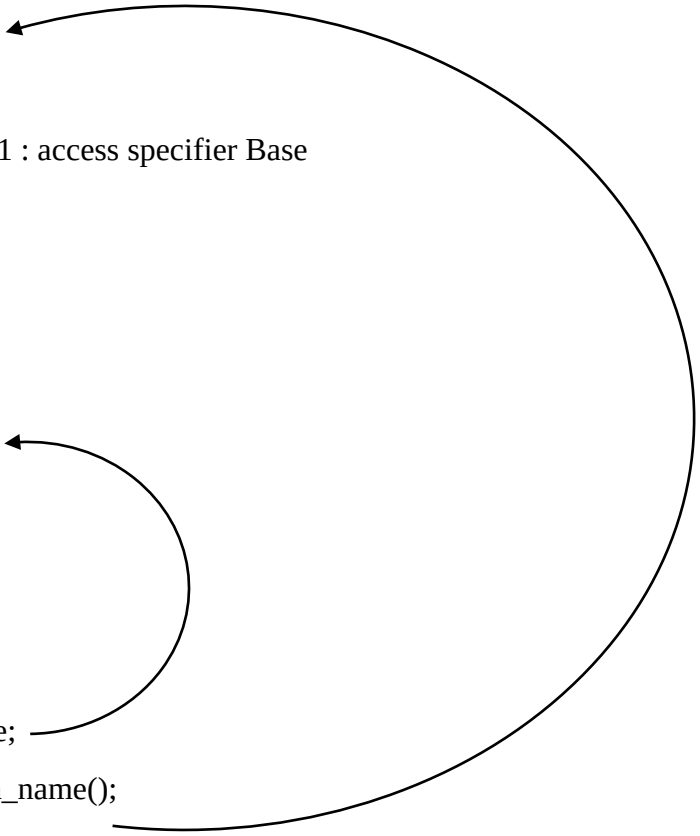
Can a class contain another class in it ?

8.11 AMBIGUITY IN INHERITANCE

When derive class create from the base class and both derive class as well as base class contains the same name member function. In this case, ambiguities can seem between different members with the same name, from different classes. because priority always goes to the object which is involve in calling the member function and in inheritance mostly we create derive class object that's why base class member function can't call because every time compiler found that member function within the object class and can't reach on base class section so the ambiguity arise . But this type of ambiguity is resolve by using scope resolution operator with class name.

Syntax : The general syntax of constructor of derived class is as follows :

```
class Base
{
….
public:
….
……
…….
fun_name();
};
class Derive1 : access specifier Base
{
…
public:
…
fun_name();
};
void main()
{
Derive1 ob;
ob.fun_name;
ob.Base::fun_name();
}
```



Example 8.10 : Program for Ambiguity resolution in inheritance

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

class abc
{
public:
void display()
{
 cout<< “\n Hello world”;
          }
};

class xyz : public abc
{
public:
void display()
{
    cout<<" \nyou are welcome";
 }
};

void main()
{
clrscr();
xyz ob;
ob.display();
ob.abc::display();
getch();
}
```

Output:

you are welcome

hello world

Check Your Progress

By which operator we can resolve ambiguity of members; which are available in both base and derive class.

8.12 DELEGATION

Delegation is simply passing a duty off to someone/something else.

- Delegation can be an alternative to inheritance.
- Delegation means that you use an object of another class as an instance variable, and forward messages to the instance.
- It is better than inheritance for many cases because it makes you to think about each message you forward, because the instance is of a known class, rather than a new class, and because it doesn't force you to accept all the methods of the super class: you can provide only the methods that really make sense.
- Delegation can be viewed as a relationship between objects where one object forwards certain method calls to another object, called its delegate.
- The primary advantage of delegation is run-time flexibility – the delegate can easily be changed at run-time. But unlike inheritance, delegation is not directly supported by most popular object-oriented languages, and it doesn't facilitate dynamic polymorphism.

Example 8.11 Pprogram to illustrate delegation

```
classRealPrinter
{
   // the "delegate"
   voidprint()
```

```
    {
        System.out.println("The Delegate");
    }
}

classPrinter
{
    // the "delegator"
    RealPrinter p = newRealPrinter();

    // create the delegate
    voidprint()
    {
        p.print(); // delegation
    }
}

publicclassTester
{
    // To the outside world it looks like Printer actually prints.
    publicstaticvoidmain(String[] args)
    {
        Printer printer = newPrinter();
        printer.print();
    }
}
```

Output:

When you delegate, you are simply calling up some class which knows what must be done. You do not really care how it does it, all you care about is that the class you are calling knows what needs doing

8.13 SUMMARY

The ability of a class to inherit the properties and characteristics from another class is called **Inheritance.** Inheritance is one of the most important feature of Object Oriented Programming. The class that inherits properties from another class is called Sub class or new or Derived Class. The class whose properties are inherited by sub class is called Super class or old class or Base Class. This unit defines about the inheritance, types of derivation types of Inheritance, how Constructor and Destructor invoke during inheritance.

8.14 EXERCISE

1. Explain different types of inheritance with block diagram and an example for each.

2. Explain the importance and role of Access specifier.

3. What is the difference between friend class and containership?

4. When we should use containership and inheritance?

5. Write a program of Constructor in a derived class.

6. What is virtual base class and abstract class? Explain with help of program.

7. How ambiguity that arises in multiple inheritance can be resolve?

8. Write a program of deriving a class ABC from an existing class XYZ by the 'protected derivation' and use data member and member function in public, protected and private visibility mode.

9. Write a program where derived class is a friend of base class.

10. Class **student** contains roll number, name and course as data member and Input_student and display_student as member function. A derived class **exam** is created from the class student with **publicly inherited**. The derived class contains mark1, mark2, mark3 as marks of three subjects and input_marks and display_result as member function. Create an array of object of the exam class and display the result of 5 students.

11. Base class 'count' contains a variable c. It contains a no argument constructor, one argument constructor, a method to return c and a operator overloading function for ++. Derived class 'counter' access the value of c from base class constructor through its constructor and a operator overloading function for.

Uttar Pradesh Rajarshi Tandon Open University

DCECS-108/MCA-110/ PGDCA-110/BCA-109/ MCS-102

C++ & OBJECT ORIENTED PROGRAMMING

BLOCK

4

POLYMORPHISM, FILE HANDLING AND OBJECT ORIENTED MODELLING

Course Design Committee

Prof. Ashutosh Gupta — **Chairman**
Director-In charge,
School of Computer and Information Science
UPRTOU, Prayagraj

Prof. U. S. Tiwari — **Member**
Dept. of Computer Science
IIIT Prayagraj

Prof. R. S. Yadav — **Member**
Department of Computer Science and Engineering
MNNIT-Allahabad, Prayagraj

Dr Marisha — **Member**
Assistant Professor (Computer Science)
School of Science, UPRTOU, Prayagraj

Mr. Manoj K. Balwant — **Member**
Assistant Professor (Computer Science)
School of Science, UPRTOU, Prayagraj

Course Preparation Committee

Dr Marisha — **Author (Block 1)**
Assistant Professor (Computer Science)
School of Science, UPRTOU,
Prayagraj

Er. Pooja Yadav — **Author (Blocks 2, 3 and 4)**
Assistant Professor
Dept. of Computer Science and IT
M.J.P. Rohilkhand University, Bareilly

Prof. Ashutosh Gupta — **Editor**
Director (In-Charge)
School of Computer and Information Science
UPRTOU, Prayagraj

Dr Marisha — **Coordinator**
Assistant Professor (Computer Science)
School of Science, UPRTOU, Prayagraj

ISBN : 978-93-83328-98-7

Printed and Published by Dr. Arun Kumar Gupta Registrar, Uttar Pradesh Rajarshi Tandon Open University, 2020.

DCECS-108/248

Printed By : Chandrakala Universal Pvt. 42/7 Jawahar Lal Neharu Road, Prayagraj.

BLOCK INTRODUCTION

In this section we discuss the overview of this block's content. This block consists of the following units:

Unit-9 : Pointers, Virtual Functions and Polymorphism

In this unit we'll discuss about pointers in C++, specifically, pointers to derived class, this pointer with the help of proper program. Polymorphism is one of the important concepts of C++. By using run time polymorphism binding of function can be performed at run time.

Unit-10 : Working With Files

In this unit we'll discuss each and everything about File Handling in C++, specifically, purpose of file handling and different ways to open and use files with examples. It also defines different file modes and file pointers.

Unit-11 : Object Oriented Modelling

In this unit we'll discuss about modelling of object oriented approach. We will discuss the graphical representation of object oriented system with various examples.

UNIT-9 POINTERS, VIRTUAL FUNCTIONS AND POLYMORPHISM

Structure:

9.0 INTRODUCTION

Polymorphism is an important feature of object oriented programming. It is of two types:

1) Compile-Time Polymorphism – This is also known as static or early binding. In early binding member function bind during compile time or before run time.

2) Runtime Polymorphism – This is also known as dynamic or late binding. In late binding member function bind during run time that means after compilation.

This is achieved by pointer, one another special pointer is also used in C++ that is this pointer.

9.1 OBJECTIVE

After studying this unit you should be able to understand the main goals of pointers, virtual function and run time polymorphism is to work in run time environment, avoid ambiguity and change the priority to the address in comparison of type of pointer.

9.2 POINTER TO OBJECT

As we make pointer of basic data types, we can also create a pointer of user define data type

like object. In the case of pointer to bject, pointers can access the members by using pointer to member operator '->'.

Syntax : The general syntax of pointer to object is as follows:

```
class classname{
statement;
                    statement ;

};
void main( )
{
classname ob,*ptr;
ptr=&ob;
ptr->datamember;
ptr->member function;
}
```

Example 9.1 Program for pointer to object

```
#include<iostream.h>
#include<conio.h>
class ABC
{
int a;
float b;
public:
        int c;
        void input(int x, int y)
        {
```

```
a=x; b=y;
}
        void display()
        {
cout<<"a="<<a;
cout<<"\nb="<<b;
cout<<"\nc="<<c;
}
        void add()
        {
cout<<"\nadd="<<a+b+c;
}
};
int main()
{
clrscr();
ABC ob;
ABC *ptr;     // pointer creation
ptr=&ob;      // Assign the address of object to the pointer of that class
ptr->c=3;
ptr->input(1,2);
ptr->display();
ptr->add();
getch();
return 0;
}
```

Output:

```
a=1
b=2
```

```
c=3
add=6
```

ptr->c =3

or

ob.c=3

both are equivalent .

When we use ptr then we use pointer to member operator. But we can use pointer with dot operator as

ob. c =5

or

(*ptr).c=5

Both are equivalent.

Check Your Progress

Can pointers to object access the private members of the class?

9.3 THIS POINTER

In C++ one keyword i.e. 'this' is used as a special pointer.Whenever any object is busy in calling then at that time location or reference of that particular object is pointed by a special pointer which is called **this** pointer or we can say in C++ every current objecthas access to its own address through an important pointer called **this** pointer.

Syntax : The general syntax of this pointer is as follows:

```
class classname{
                    statement;
                    statement ;
                      return type fun_name( ){
this->data member;
this-> memberfunction();
```

```
                                        }
                }
```

Because data member and member function are associated with the object. So Inside the class we can also use data member and member function by using this pointer. like

this->a=2

this->getdata()

but friend function is not access with the help of this pointer because friend function is not a member function.

Example 9.2 Program for this pointer

```
#include<iostream.h>
#include<conio.h>
class A
{
int a;
public:
        A(int x){a=x;}
        A(){}
        int display(){return a;}
        A greater (A &x)
        {
        if (x.a>a) return x;
        else return *this;
        }
};

int main()
{
clrscr();
```

```
A ob1(3),ob2(2),ob3;
ob3=ob1.greater(ob2);
cout<<"greater no. is="<<ob3.display();
getch();
return 0;
}
```

Output:

greater no. is =3

Check Your Progress

Explain this pointer with suitable program and also write down the output of cout<<*this and cout<<this

9.4 POINTERTO DERIVED CLASSES

In the case of Inheritance we can create the pointer of base class as well as derive class but base class pointer can access only those members of derived class which are inherited by base class i.e. its own member. And derived class pointer can also access its own as well as inherited data. But In C++ base class pointer or pointers to objects of base class are types compatible with pointer to object of derived class. And after that it can access the member of derived class.

((derive_class_name *) base_ptr)->fun( ) ;

This shows the type compatibility of base class pointer with pointer to object of derived class.

By this base class pointer can access the original members of derived class.

Syntax : The general syntax of Pointer to derived class is as follows:

```
class Baseclassname
{
```

```
datamember;
                public:
void fun()
                        {
 statement ;
  statement ;
                         }
};
class Deriveclassname
{
                public:
                        void memberfun()
                        {
  statement ;
  statement ;
                         }
};
void main( )
{
                 Deriveclassname ob1,ptr1;
                 ptr1=&ob1;
                 ptr1->fun();
                 ptr1->memberfun();
}
```

Example 9.3 Program for Pointer to derived classes

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

```
{
public:
        void display1(){cout<<"\nbase display()";}
        void show1(){cout<<"\nbase show()";}
};

class derived : public base
{
public :
        void display(){cout<<"\nderived display()";}
        void show(){cout<<"\nderived show()"; }
};

int main()
{
clrscr();
base *bptr, obb;
derived obd,*dptr;
//dptr=&obb;  // can't do
dptr=&obd;
dptr->display1();
dptr->show1();
dptr->display();
dptr->show();
bptr=&obd;
((derived *) bptr)->display( ); // type compatibility
((derived *) bptr)->show( );
//bptr->show( );// can't do ;because show is not a member of base
getch();
return 0;
```

```
}
```

Output:

base display()

base show()

derived display()

derived show()

derived display()

derived show()

Check Your Progress

What is the meaning of compatibility in case of pointer to base class ?

9.5 VIRTUAL FUNCTION

When base class and derived class both have the same name data member and member function then whenever base class pointer access that member function or data member then it access only its own data member or member function. similarly when derived class pointer access that common name data member or member function then at that time it access its own data member or function because in both time the priority given to type of pointer, not the address assign to that pointer.

A **virtual** function is a function ofbase class that is declared using the keyword **virtual**.

In C++ compile-time polymorphism is the concept of overloading, where signatures of all same function should be different in terms of no. of argument as well as type of argument. But runtime polymorphism is the concept of function **overriding** that means same function name and same signature.

Characteristics of Virtual Function:

- It must be a member of a class.
- It can't be static.
- It should be declare/define in base class and in public visibility mode.

- It must be accessed by a pointer of object
- It can't be friend of another class.
- Destructor can be virtual but Constructor can't be virtual.

Syntax : The general syntax of virtual function is as follows:

```
class Baseclassname
{
datamember;
        public:
                virtual void memberfun()
                {
statement ;
 statement ;
                }
};
class Deriveclassname
{
        public:
                void memberfun()
                {
     statement ;
     statement ;
                }
};

void main( )
{
Baseclassname ob,*ptr;
Deriveclassname ob1;
ptr=&ob;
ptr->memberfun();
```

```
ptr=&ob1;
ptr->memberfun();
}
```

Check Your Progress

What is the need of virtual function?

9.6 IMPLEMENTATION OF RUN TIME POLYMORPHISM

Run time polymorphism is achieved only by base class pointer and virtual function; which is define in base class and that function redefine in derive class. When we give the address of base or derive class object to base class pointer then just because of virtual function present in base class the preference goes to the address assign to the pointer instead of type of pointer.

```
base obb;
base *bptr;
derive obd;
//derive *dptr;
bptr=&obb;
bptr->fun();                    //  it invokes the fun( ) present in base class
```

If fun( ) is virtual function then

```
bptr = &obd;
bptr->fun( ); // it invokes the fun( ) present in derive class
```

just because when virtual function present in base class then priority shift to the assigned address to the pointer not to the type of pointer.

Example 9.4 Program for virtual function

```
#include<iostream.h>
#include<conio.h>
class base
{
public:
        virtual void display(){cout<<"\nbase display()";}
        void show(){cout<<"\nbase show()";}

};
class derived : public base
{
public :
        void display(){cout<<"\nderived display()";}
        void show(){cout<<"\nderived show()";}
};
int main()
{
clrscr();
base *bp;
base B;
bp=&B
bp->display();
bp->show();
derived D;
bp=&D;
bp->display();
bp->show();
B.display();
B.show();
```

```
getch();
return 0;
}
```

Output:

```
base display()
base show()
derived display()
derived show()
base display()
base show()
```

In the above example show( ) is not a virtual function so in both time when it is calling by base class pointer then both time show ( ) function of base class is calling; because pointer is of base class but In second case display( ) is a virtual function so in both time when it is calling by base class pointer then both time display ( ) function of that class is calling which address is pass to the base class pointer.

B.display( ) and B.show( ) represent compile time binding not run time binding. Because runtime polymorphism is achieved only by the pointer of base class not by the object and dot operator of the class. In both the case results are same but it is not the concept of run time polymorphism.

Check your Progress

Explain the problem with overriding functions.

9.7 PURE VIRTUAL FUNCTION

If virtual function in base class is defined null or has no definition and redefines in derived class, then this type of virtual function is called **Pure Virtual Function**.Because of Null definition pure virtual function is also called 'Do- nothing –function' and the class which contains pure virtual function is also called **Abstract Class.**

Abstract class have at least one pure function and that class used as a base class for inheritance. But C++does not allow programmer to create object for abstract class.

Syntax: The general syntax of Purevirtual function is as follows:

```
class Baseclassname
{
datamember;
                    public:
virtual void memberfun() =0;
};
class Deriveclassname
{
            public:
                        void memberfun()
                        {
 statement ;
  statement ;
                         }
};

void main( )
{
Baseclassname *ptr;
Deriveclassname ob1;
ptr=&ob1;
ptr->memberfun();
}
```


We can change the virtual function area ( ) in the base class to the following −

Example 9.5: Program for Pure Virtual function

```
#include<iostream.h>
#include<conio.h>
class base
{
public:
        virtual void display()=NULL;
        virtual void show()=0;
};
class derived : public base
{
public :
        void display()
        {
        cout<<"\nderived display()";
        }
        void show()
        {
        cout<<"\nderived show()";
        }
};

int main()
{
        clrscr();
        base *bp;
        derived D;
        bp=&D;
        bp->display();
        bp->show();
```

```
    getch();
    return 0;
}
```

Output:

derived display()

derived show()

Check Your Progress

Which is also called as abstract class?

9.8 SUMMARY

Runtime polymorphism is one of the important concept in which any function bind dynamically at run time by using pointer of object and this pointer help to represent the address of current object at particular moment when that object is busy to invoke any function or access data member .

9.9 EXERCISE

1) What is pure virtual function? What is the need of pure virtual function?

2) What is dynamic binding?

3) How can we achieve runtime polymorphism?

4) Can we access derive class data members and member function by using base class pointer? Explain in detail.

5) Can constructor and destructor bevirtual or not?

6) Which is the *pointer* which denotes the object calling the member function?

7) Write a program to access and print class member's variable 'x' and assign the value to 'x' in the function sum() and the value which is assign is the local variable's i.e. also 'x' which is pass in function sum()

8) Can we use This pointer Inside Static Member Function?

9) Can weuse This pointer in the Constructor? If yes then How? Write a program and explain it.

10) Write a program with Student as abstract class and create derive classes Engineering, Medicine and Science from base class Student. Create the objects of the derived classes and process them and access them using array of pointer of type base class Student.

11) Write a program using this pointer to find out the least number obtained among three subjects. Use ternary operator.

12) Class polygon contains data member width and height and public method set_value() to assign values to width and height. class Rectangle and Triangle are inherited from polygon class. Both the classes contain public method calculate_area() to calculate the area of Rectangle and Triangle. Use base class pointer to access the derived class object and show the area calculated.

13) Write a program to create a class shape with functions to find area of and display the name of the shape and other essential component of the class. Create derived classes circle, rectangle and trapezoid each having overridden functions area and display. Write a suitable program to illustrate virtual functions.

UNIT-10 WORKING WITH FILES

Structure

10.0 INTRODUCTION

Each and every collection of related data stored with a specific format on a secondary storage device, is called a file. Very large data is always stored in a file.

To communicate with files means to read or write from/on files, program or software is required.

In C++ there are two types of communication between I/O device, file and program.

(i) Communication between console and program.

(ii) Communication between program and files.

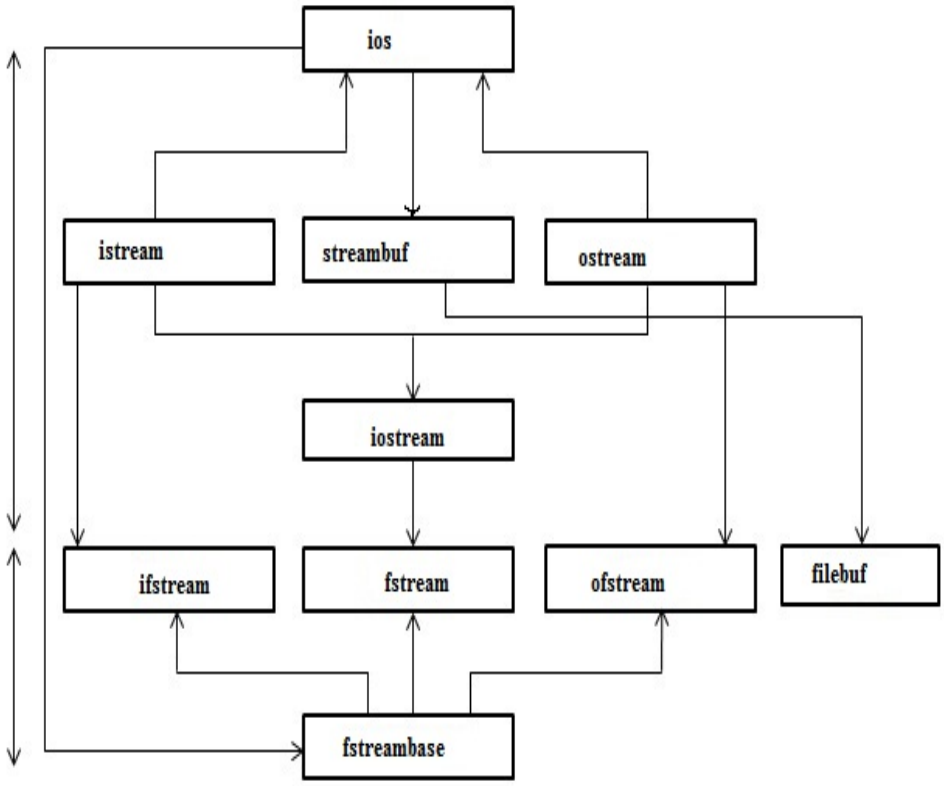
10.1 OBJECTIVE

After studying this unit you should be able to understand File Handling, why we used files, how to open and close the file and different operations/file modes and pointers that we can perform/used in files.

10.2 CLASSES FOR FILE STREAM OPERATIONS

The I/O system provides an interface/link to the programmer between program and file. This interface/link is known as stream. It refers

to a sequence of bytes Stream is establish/connected when open operation is performed and disconnected when close operation is performed. This stream is responsible for transmit the data between file and program. For input and output, there are two different streams called input stream and output stream. The source stream that supplies data to the program is called the input stream and the destination stream that receives output from the program is called the output stream.



filebuf-It contains open ( ) and close ( ) as its member function. It is derived from ios class and it is used to set the file buffer for read and write operation.

ifstream- It contains open ( ) and close ( ) get(),getline(),read(),seekg(), tellg( ) as its member function. And open ( ) is used to open a file in input mode. It is derived from istream and fstreambase class and it is used to perform read operation.

ofstream- It contains open ( ) and close ( ),put( ),write( ),seekp( ),tellp( ) as its member function. And open ( ) is used to open a file in output mode. It is derived from ostream and fstreambase class and it is used to perform write operation.

fstream-It inherit all the function from istream, ostream through iostream class . It contains open ( ) and close ( ) as its member function. And open(

) function by default works in input mode. It contains function for both read and write operation.

Check Your Progress

What is the role of fstream.h header file in a program?

10.3 OPENING AND CLOSING A FILE

Before using a file 4 points are required to know:

- Name of a file
- Purpose to open a file
- Opening method
- Used Data structure

So before start to do work on files it is necessary to open a file. and in C++ there are two methods to open a file:

- By using constructor
- By using open() function

10.3.1 BY USING CONSTRUCTOR

When file is open by using constructor then name of file is assign to the object at the creation time.

Syntax :

```
class_name object("file_name");
```

Example 10.1: Program of open a file by using constructor

```
class  file
{
   public:
   void fopen();
};
void file::fopen()
```

```
{
    char line1[100];
    ofstream fout("exp.cpp");
    fout<<"this is file handling program";
    fout.close();
    cout<<"\n In reading mode:";
    ifstream fin("exp.cpp");
    fin.getline(line1,5);
    cout<<line1;
    fin.close();
}
void main()
{
    clrscr();
    file obf;
    obf.fopen();
    getch();
}
```

Output:

In reading mode : this

10.3.2 BY USING OPEN () FUNCTION

When file is open by using open function then name of file is assign to the open function as an argument at the time of function call.

Syntax :

```
class_name object;
object.open(“file_name”);
```

Example 10.2 : Program of open a file by using open function

```
#include<conio.h>
#include<iostream.h>
#include<fstream.h>
class  file
{
        public:
        void fopenbyfun();
};
void file::fopenbyfun()
{
         char line1[100];
         ofstream fout;
         fout.open("abc1.cpp");
         fout<<"now we use open function for opening a new file";
         fout.close();
         ifstream fin;
         fin.open("abc1.cpp");
         fin.getline(line1,20);
         cout<<"\n\t"<<line1;
         fin.close();
 }
void main()
{
        clrscr();
        file obf;
        obf.fopenbyfun();
        getch();
}
```

Output:

now we use open fun

The difference between both methods is that if file is open by using constructor then every object is connected to only one file. But in open function any object is connected with different file at different time.

Check Your Progress

When we should use open function to open a file?

10.4 FILE POINTERS AND THEIR MANIPULATIONS

In C++ two pointers are used to indicate the position in the file at which the next input/output is to be present. In which one pointer is used to ask or tell the location of current position in the file that pointer is known as tellp/tellg. Another pointer is used to move the file pointer in a file for some input /output operation like insertion modification, deletion, searchingetc. and that pointer is known as seekp/seekg.

seekg( )/seekp( ) have two arguments; one is offset and second is base address and total no. of bytes movement is calculated as offset +base-address

base address is defined by position

(i) **ios::beg -** Start from beginning. In this case pointer moves only in forward direction. So offset is in the form of +x

eg: ob.seekg (x, ios::beg);

(ii) **ios::cur -** Start from current.In this case pointer moves in both forward and backward direction. So offset is in the form of **+x** for forward direction and **–x** for backward direction

eg: ob.seekg (-x, ios::cur);

(iii) ios::end - Start from end of the file. In this case pointer moves in backward

direction. So offset is in the form of **–x**for backward direction.

eg:ob.seekg (-x, ios::end);

i.e. to move to +**x**byte position from beginning, and current and to move to -**x**byte position from current and end.

seekg(): It moves the file pointer to the specified position during input operation.

Syntax: The general syntax of seekg() function is as follows :

```
ifstream ff;
ff.seekg(x,ios::beg);
```

seekp( ) : It moves the file pointer to the specified position during output operation.

Syntax : The general syntax of seekp() function is as follows:

```
ofstream ff1;
ff1.seekp(x,ios::beg);
```

tellg(): This function returns the file pointer's current position during input operation.

Syntax: The general syntax of tellg() function is as follows :

```
ifstream f;
int p=f.tellg();
```

tellp() : This function returns the file pointer's current position during output operation.

Syntax : The general syntax of tellp() function is as follows:

```
ofstream f1;
int pos=f1.tellp();
```

Check Your Progress

How to get the current position of the file pointer?

10.5 FILE MODES

File modes are used to define the purpose to open a file and it is the second argument first one is file name.Different type offile mode parametersand their meanings

ios::app- It is used to append to end of file

ios::ate-It is used to go to end of file for writing and move backward as well as forward direction.

ios::binary-It is used to file open in binary mode

ios::in- It is used to open file for reading only

ios::out- It is used to open file for writing only

ios::nocreate- open fails if the file does not exist

ios::noreplace- open fails if the file already exist

ios::trunc delete the contents of the file if it exist

All these flags can be combined using the bitwise operator OR (|).

Example 10.3 : Program for file modes and file pointers

```
class file
{
        public: void fmode();
};

void file:: fmode()
{
        char line[250];
        fstream fb("new.txt",ios::ate| ios::cur | ios::app);
        fb<<"Today is very hot day  ";
```

```
        cout<<"pos="<<fb.tellp();  //to check the pointer pos
        fb.seekg(0);     //for start point
        fb.getline(line,250);
        cout<<line;
        fb.close();
 }

void main()
{
        file obf;
        obf.fmode();
        getch();
}
```

Output:

pos=23 Today is very hot day

Check Your Progress

What does the nocreate and noreplace flag ensure when they are used for opening a file?

10.6 SEQUENTIAL INPUT AND OUTPUT OPERATIONS

There are get() ,getline() and read()functions for input operations and put() and write() are output operations

the function put() writes a single character to the associated stream. Similarly,

the function get() reads a single character form the associated stream.

While write() and read()

Both functions have two arguments. The first is the address of variable and the second is the size of that variable in bytes. The address of the variable must be type cast to type char * (i.e., a pointer to character type.)The data written to a file using write() can only be read accurately using read().

Syntax : get()

```
streamobject.get(ch);
```

Syntax : getline()

```
streamobject.getline(char_array, length);
```

Syntax : put()

```
streamobject.put(ch);
```

Syntax : read()

```
streamobject.read((char *)&variable, sizeof(variable));
```

Syntax : write ()

```
streamobject.write((char *)& variable, sizeof(variable));
```

Example 10.4: Program for input and output operations by using functions

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

```
class file
{
        public: void fkey();
};

void file::fkey()
{
      char line1[100],c,ch;
      ofstream fout("ep.txt");
      cout<<"enter the string";
      cin>>c;  //enter by using keyboard
      while(c!='\n')
      {
          fout.put(c);   //writing in file char by char
          cin.get(c);    //enter by using keyboard
      }
      fout.close();
      cout<<"\n In reading mode:\n";
      ifstream fin("ep.txt");
      while(fin.eof()==0)
      {
          fin.get(c);    //read the char
          cout<<c;       //print the char
      }
}
```

Output:

```
enter the string hello India
In reading mode:
hello India
```

Check Your Progress

What are put and get pointers?

Example 10.5 : Program for write()/read() function

```
#include<conio.h>
#include<iostream.h>
#include<fstream.h>
class binary
{
    private:
    int roll;
    char name[30];
    public:
    void input()
    {
      cout<<"\nenter your name:\t";
      cin>>name;
      cout<<"\nenter your rollno.:\t";
      cin>>roll;
     }
    void output()
    {
    cout<<"\nNAME:"<<name;
    cout<<"\nROLL  NO:"<<roll;
    }
};
```

```
void binary::create()
{
ofstream of;   binary b1;
  of.open("BIN1.txt",ios::out|ios::in);
  b1.input();
  of.write((char*)&b1,sizeof(b1));
  of.close();
  ifstream iff;
  iff.open("BIN1.txt",ios::in|ios::out);
  iff.read((char*)&b1,sizeof(b1));
  while(iff.eof()==0)
{
    iff.read((char*)&b1,sizeof(b1));
  }
b1.output();
  iff.close();
  }

  void main()
  {
      clrscr();
      binary b;
      b.create();
      getch();
  }
```

Output:

```
enter your name: priya
enter your rollno.:11
NAME: priya
```

ROLL: 11

Check Your Progress

Is it possible for the objects to read and write themselves?

10.7 ERROR HANDLING DURING FILE OPERATIONS

When user works on file then this is obvious to face lots of problem due to several reasons:

- To perform read operation and file doesn't exist.
- File extension is incorrect.
- Open a file with invalid file name
- Perform write operation and disk space if full
- Read /write operations does not appropriate with used stream and their objects.
- Perform Read operation while file is at end.

There are few functions that are used for handling the error.

eof() - This function is used for detection of end of the file.

It returns a non–zero value i.e. True value if end of file is encounter.

Otherwise zero i.e. False .

fail() - This function detect the failure of operation.

It returns a non–zero value i.e. True value if read or write operation is failed

due to some reasons.

Otherwise zero i.e. False .

bad() - This function detect the unrecoverable error.

It returns a non–zero value i.e. True value if any unrecoverable error is encounter.

Otherwise zero i.e. False .

good() - It indicate that everything is fine

It returns a non–zero value i.e. True value if no error / problem is encounter.

Otherwise zero i.e. False .

If bad() or fail() returns true then good() return false but if good() returns true then bad() or

fail () returns false.

Example 10.6: Program of error handling during file operation

```
class file
{
        public: void err_handling();
};
void file::err_handling()
 {
          clrscr();
           char str[50];
            ifstream iff,ff;
           iff.open("test1.txt");
           if(iff.fail()!=0)       //if i/p o/p operation is failed
           {
          cout<<"file does not exist";    getch();    delay(1000);
           }
           if(iff.bad()!=0)   //for unrecoverable error is occur
           {
          cout<<"\nread/write operation is not perform;due to some reason\n";
```

```
  getch();delay(100);
 }
 if(iff.good()!=0)           //if there is no error
 {
  cout<<"\nno error is occured so operation can be perform\n";
  getch();delay(100);
 }
 iff.close();
 ff.open("abc.cpp");
 while(ff.eof()==0)    //for till file is not ending
 {
 ff.getline(str,50);  //to read a line from file
  cout<<str<<"\n";
 }
 ff.close();
 }

 void main()
 {
 file obf;
 obf.err_handling();
 getch();
 }
```

Output:

file does not exist

read/write operation is not perform: due to some reason

this is file handling program

Check Your Progress

What is the meaning of 1 and 0 in fail( ) and good( )?

10.8 COMMAND LINE ARGUMENTS

In C/C++ main( ) is the essential function. In normal function we pass either the value or the address of the variable which hold particular type of value, but not in main. But command line argument provide this facility by which we can supply the argument to the main( ) function.

Syntax :

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

Where **argc** is represent number of argument in command line , which is called argument counter and **argv**is represent array of character type points to command line argument , which is called argument vector. These argument passed at the time of invoking the program by using the command prompt.

eg: C:\> filename argument1 argument2

C:\> test x y

Where argc is 3 → test x y

argv[0] →test

argv[1]→x

argv[2]→y

First index argv[0] always points the program file i.e. test is the filename in which this program is saved .

Example 10.7 : Program for commandline argument

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



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

```
{
        int i;
        clrscr();
        cout<<"argc="<<"\n"<<argc;
        for( i=0;i<argc;i++)
        {
        cout<<"argv["<<i<<"]="<<argv[i];
        cout<<"\n";
        }
        getch();
}
```

Output :

```
argc=6
argv[0]=C:\TURBOC3\BIN\COMMAND.EXE
argv[1]=p
argv[2]=q
argv[3]=r
argv[4]=s
argv[5]=t
```

File reading operation by using Command line argument.

Example 10.8 : Program to read the file

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


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

```
{
        int  x;
        char line1[50];
        clrscr();
        cout<<"\n total no. of arguments are:";
        cout<<"argc="<<argc<<endl;
        for(int i=1; i <argc; i++) // i=1, assuming files arguments are right after the executable
        {
          cout<<"\nargv["<<i<<"]="<<argv[i];
          cout<<"\n";
          char *fn = argv[i]; //filename
          fstream f;
          f.open(fn,ios::out);
          f<<"now we use open function for opening a new file";
          f.close();
          f.open(fn,ios::in);
          f.getline(line1,20);
          cout<<"\n"<<line1;
          f.close();
          getch();
        }
return 0;
    }
```

Output:

total no. of arguments are :argc=2

argv[1]=cute.txt

now we use open fun

Check your Progress

What is the first argument of command line?

10.9 SUMMARY

In real world applications lots of works perform in files. A file is used to read or write the data. For doing the same first of all, open the file by using either open function or constructor, and perform the working and after that close the file by using close function. The purpose of opening the file is defined by file modes. And during the working to locate different positions, file pointers are used and also used some functions for handling the error during file operations.

10.10 EXERCISE

1. What are command line arguments?

2. How many methods are used to open a file?

3. What is the difference to open a file by using constructor and open function?

4. What do you understand by File pointers?

5. What are file modes? Explain different type of file modes.

6. Write a C++ program to write number 1 to 100 in a data file NOTES.TXT

7. Write a C++ program, which initializes a string variable to the content " Surround yourself with only people who are going to lift you higher." and outputs the string to the disk file OUT.TXT.

8. Write a user-defined function in C++ to read the content from a text file “SHOW.TXT”, count and display the number of alphabets present in it.

9. Write a function to count the number of blank present in a text file named "SHOW.TXT".

10. Write a function to count number of words in a text file named "DEMO.TXT".

11. Write a function in C++ to count and display the number of lines not starting with alphabet 'A' present in a text file "DEMO.TXT".

UNIT-11 OBJECT ORIENTED MODELLING

11.1 INTRODUCTION

A model is an abstraction of something for the purpose of understanding it before building it. Because, real systems that we want to study are generally very complex. In order to understand the real system, we have to simplify the system. So a model is an abstraction that hides the non-essential characteristics of a system and highlights those characteristics, which are pertinent to understand it. Efraim Turban describes a model as a simplified representation of reality. A model provides a means for conceptualization and communication of ideas in a precise and unambiguous form. The characteristics of simplification and representation are difficult to achieve in the real world, since they frequently contradict each other. Thus modelling enables us to cope with the complexity of a system.

Most modelling techniques used for analysis and design involve graphic languages. These graphic languages are made up of sets of symbols. As you know one small line is worth thousand words. So, the symbols are used according to certain rules of methodology for communicating the complex relationships of information more clearly than descriptive text.

Modelling is used frequently, during many of the phases of the software life cycle such as analysis, design and implementation. Modelling like any other object-oriented development, is an iterative process. As the model progresses from analysis to implementation, more detail is added to it.

11.2 OBJECTIVE

After studying this unit you should be able to understand Object Oriented Modelling/UML, its requirements and real life applications. And

alsobrief introduction of designing methods and terminology used in UML.

11.3 NEED OF OBJECT ORIENTED MODELLING

A model provides the blueprints of a system. Models may encompass detailed plans. A good model includes those elements that have broad effect and omits those minor elements that are not relevant to the given level of abstraction. Every system may be described from different aspects using different models, and each model is therefore a semantically closed abstraction of the system. A model may be structural, emphasizing the organization of the system, or it may be behavioural, emphasizing the dynamics of the system.

Through modelling, we achieve four aims.

1. Models help us to visualize a system as it is or as we want it to be.
2. Models permit us to specify the structure or behaviour of a system.
3. Models give us a template that guides us in constructing a system.
4. Models document the decisions we have made.

Modelling is not just for big systems. Even the software equivalent of a dog house can benefit from some modelling. However, it's definitely true that the larger and more complex the system, the more important modelling becomes, for one very simple reason: We build models of complex systems because we cannot comprehend such a system in its entirety.

There are limits to the human ability to understand complexity. Through modelling, we narrow the problem we are studying by focusing on only one aspect at a time. Furthermore, through modelling, we amplify the human intellect. A model properly chosen can enable the modeller to work at higher levels of abstraction.

Civil engineers build many kinds of models. Most commonly, there are structural models that help people visualize and specify parts of systems and the way those parts relate to one another. Depending on the most important business or engineering concerns, engineers might also build dynamic models- for instance, to help them to study the behavior of a structure in the presence of an earthquake. Each kind of model is organized differently, and each has its own focus.

In software, there are several ways to approach a model. The two most common ways are from an algorithmic perspective and from an object-oriented perspective. The traditional view of software development takes an algorithmic perspective. In this approach, the main building block of all software is the procedure or function. This view leads developers to focus on issues of control and the decomposition of larger algorithms into smaller ones. There's nothing inherently evil about such a point of view

except that it tends to yield brittle systems. As requirements change (and they will) and the system grows (and it will), systems built with an algorithmic focus turn out to be very hard to maintain. The contemporary view of software development takes an object-oriented perspective. In this approach, the main building block of all software systems is the object or class. Simply put, an object is a thing, generally drawn from the vocabulary of the problem space or the solution space; a class is a description of a set of common objects. Every object has identity (you can name it or otherwise distinguish it from other objects), state (there's generally some data associated with it), and behaviour (you can do things to the object, and it can do things to other objects, as well).

For example, consider a simple three-tier architecture for a billing system, involving a user interface, middleware, and a database. In the user interface, you will find concrete objects, such as buttons, menus, and dialog boxes. In the database, you will find concrete objects, such as tables representing entities from the problem domain, including customers, products, and orders.

In the middle layer, you will find objects such as transactions and business rules, as well as higher-level views of problem entities, such as customers, products, and orders. The object-oriented approach to software development is decidedly a part of the mainstream simply because it has proven to be of value in building systems in all sorts of problem domains and encompassing all degrees of size and complexity. Furthermore, most contemporary languages, operating systems, and tools are object-oriented in some fashion, giving greater cause to view the world in terms of objects. Object-oriented development provides the conceptual foundation for assembling systems out of components using technology such as Java Beans or COM+.

11.4 PRINCIPLES OF MODELLING

There four basic principles of modeling :

1. ***The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped.***

   In other words, choose your models well. The right models will brilliantly illuminate the most wicked development problems, offering insight that you simply could not gain otherwise; the wrong models will mislead you, causing you to focus on irrelevant issues. In software, the models you choose can greatly affect your world view. If you build a system through the eyes of a database developer, you will likely focus on entity-relationship models that push behaviour into triggers and stored procedures. If you build a system through the eyes of a structured analyst, you will likely end up with models that are algorithmic-centric, with data flowing from process to process. If you build a system through the eyes of an object-oriented developer, you'll end up with a system whose

architecture is centred around a sea of classes and the patterns of interaction that direct how those classes work together. Any of these approaches might be right for a given application and development culture, although experience suggests that the object-oriented view is superior in crafting resilient architectures, even for systems that might have a large database or computational element. That fact notwithstanding, the point is that each world view leads to a different kind of system, with different costs and benefits.

2. Every model may be expressed at different levels of precision.

If you are building a high rise, sometimes you need a 30,000-foot view- for instance, to help your investors visualize its look and feel. Other times, you need to get down to the level of the studs- for instance, when there's a tricky pipe run or an unusual structural element. The same is true with software models. Sometimes, a quick and simple executable model of the user interface is exactly what you need; at other times, you have to get down and dirty with the bits, such as when you are specifying cross-system interfaces or wrestling with networking bottlenecks. In any case, the best kinds of models are those that let you choose your degree of detail, depending on who is doing the viewing and why they need to view it. An analyst or an end user will want to focus on issues of what; a developer will want to focus on issues of how. Both of these stakeholders will want to visualize a system at different levels of detail at different times.

3. The best models are connected to reality.

A physical model of a building that doesn't respond in the same way as do real materials has only limited value; a mathematical model of an aircraft that assumes only ideal conditions and perfect manufacturing can mask some potentially fatal characteristics of the real aircraft. It's best to have models that have a clear connection to reality, and where that connection is weak, to know exactly how those models are divorced from the real world. All models simplify reality; the trick is to be sure that your simplifications don't mask any important details.

4. No single model is sufficient. Every nontrivial system is best approached through a small set of nearly independent models.

If you are constructing a building, there is no single set of blueprints that reveal all its details. At the very least, you'll need floor plans, elevations, electrical plans, heating plans, and plumbing plans. The operative phrase here is "nearly independent." In this context, it means having models that can be built and studied separately but that are still interrelated. As in the case of a building, you can study electrical plans in isolation, but you can also see their mapping to the floor plan and perhaps even their interaction with the routing of pipes in the plumbing plan.

Check Your Progress

What Is Modelling? What Are The Advantages Of Creating A Model?

11.5 SIMULATION OF REAL LIFE PROBLEMS USING OOPS : EXAMPLE

The Unified Modelling Language (UML) is a standard language for writing software blueprints. The UML may be used to visualize, specify, construct, and document the artifacts of a software intensive system. The UML is appropriate for modelling systems ranging from enterprise information systems to distributed Web-based applications and even to hard real time embedded systems. It is a very expressive language, addressing all the views needed to develop and then deploy such systems. Even though it is expressive, the UML is not difficult to understand and to use. Learning to apply the UML effectively starts with forming a conceptual model of the language, which requires learning three major elements: the UML's basic building blocks, the rules that dictate how these building blocks may be put together, and some common mechanisms that apply throughout the language.

The UML is only a language and so is just one part of a software development method. The UML is process independent, although optimally it should be used in a process that is use case driven, architecture-centric, iterative, and incremental. The UML is a language for

- Visualizing
- Specifying
- Constructing
- Documenting

The artifacts of a software-intensive system.

The UML Is a Language

A language provides a vocabulary and the rules for combining words in that vocabulary for the purpose of communication. A modelling language is a language whose vocabulary and rules focus on the conceptual and physical representation of a system. A modelling language such as the UML is thus a standard language for software blueprints.

The UML Is a Language for Visualizing

For many programmers, the distance between thinking of an implementation and then pounding it out in code is close to zero. You think it, you code it. In fact, some things are best cast directly in code. Text is a wonderfully minimal and direct way to write expressions and

algorithms. In such cases, the programmer is still doing some modelling, albeit entirely mentally. He or she may even sketch out a few ideas on a white board or on a napkin.

The UML Is a Language for Specifying

In this context, specifying means building models that are precise, unambiguous, and complete. In particular, the UML addresses the specification of all the important analysis, design, and implementation decisions that must be made in developing and deploying a software-intensive system.

The UML Is a Language for Constructing

The UML is not a visual programming language, but its models can be directly connected to a variety of programming languages. This means that it is possible to map from a model in the UML to a programming language such as Java, C++, or Visual Basic, or even to tables in a relational database or the persistent store of an object-oriented database. Things that are best expressed graphically are done so graphically in the UML, whereas things that are best expressed textually are done so in the programming language.

The UML Is a Language for Documenting

A healthy software organization produces all sorts of artifacts in addition to raw executable code. These artefacts include (but are not limited to)

- Requirements
- Architecture
- Design
- Source code
- Project plans
- Tests
- Prototypes
- Releases

Depending on the development culture, some of these artifacts are treated more or less formally than others. Such artifacts are not only the deliverables of a project, they are also critical in controlling, measuring, and communicating about a system during its development and after its deployment. The UML addresses the documentation of a system's architecture and all of its details. The UML also provides a language for expressing requirements and for tests. Finally, the UML provides a language for modelling the activities of project planning and release management.

Check your Progress

What should be mentioned as attributes for conceptual modelling ?

11.5.1 BUILDING BLOCKS OF THE UML

The vocabulary of the UML encompasses three kinds of building blocks:

1. Things
2. Relationships
3. Diagrams

Things are the abstractions that are first-class citizens in a model; relationships tie these things together; diagrams group interesting collections of things.

Things in the UML

There are four kinds of things in the UML. These things are the basic object-oriented building blocks of the UML. You use them to write well-formed models.

1. ***Structural Things :*** Structural things are the nouns of UML models. These are the mostly static parts of a model, representing elements that are either conceptual or physical. In all, there are seven kinds of structural things: classes, interfaces, collaborations, use cases, active classes, components, and nodes.
2. ***Behavioural Things :*** Behavioural things are the dynamic parts of UML models. These are the verbs of a model, representing behaviour over time and space. In all, there are two primary kinds of behavioural things: interaction and state machine.
3. ***Grouping Things :*** Grouping things are the organizational parts of UML models. These are the boxes into which a model can be decomposed. In all, there is one primary kind of grouping thing, namely, packages.
4. ***An notational Things :*** An notationalthings are the explanatory parts of UML models. These are the comments you may apply to describe, illuminate, and remark about any element in a model. There is one primary kind of an notational thing, called a note. A note is simply a symbol for rendering constraints and comments attached to an element or a collection of elements.

Check your Progress

Which things are dynamic parts of UML models?

11.5.2 RELATIONSHIPS IN THE UML

There are four kinds of relationships in the UML :

1. Dependency
2. Association
3. Generalization
4. Realization

These relationships are the basic relational building blocks of the UML. You use them to write well-formed models.

11.5.3 DIAGRAMS IN THE UML

A diagram is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships). You draw diagrams to visualize a system from different perspectives, so a diagram is a projection into a system. For all but the most trivial systems, a diagram represents an elided view of the elements that make up a system. The same element may appear in all diagrams, only a few diagrams (the most common case), or in no diagrams at all (a very rare case). In theory, a diagram may contain any combination of things and relationships. In practice, however, a small number of common combinations arise, which are consistent with the five most useful views that comprise the architecture of a software-intensive system.

The UML includes nine such diagrams:

1. ***Class diagram :*** A class diagram shows a set of classes, interfaces, and collaborations and their relationships. These diagrams are the most common diagram found in modelling object-oriented systems. Class diagrams address the static design view of a system. Class diagrams that include active classes address the static process view of a system.
2. ***Object diagram :*** An object diagram shows a set of objects and their relationships. Object diagrams represent static snapshots of instances of the things found in class diagrams. These diagrams address the static design view or static process view of a system as do class diagrams, but from the perspective of real or prototypical cases.

3. ***Use case diagram :*** A use case diagram shows a set of use cases and actors (a special kind of class) and their relationships. Use case diagrams address the static use case view of a system. These diagrams are especially important in organizing and modelling the behaviours of a system.

4. ***Sequence diagram :*** Both sequence diagrams and collaboration diagrams are kinds of interaction diagrams. An shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them. Interaction diagrams address the dynamic view of a system. A sequence diagram is an interaction diagram that emphasizes the time-ordering of messages.

5. ***Collaboration diagram :*** Collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages. Sequence diagrams and collaboration diagrams are isomorphic, meaning that you can take one and transform it into the other.

6. ***State-chart diagram :*** A state-chart diagram shows a state machine, consisting of states, transitions, events, and activities. State-chart diagrams address the dynamic view of a system. They are especially important in modelling the behaviour of an interface, class, or collaboration and emphasize the event-ordered behaviour of an object, which is especially useful in modelling reactive systems.

7. ***Activity diagram :*** An activity diagram is a special kind of a state chart diagram that shows the flow from activity to activity within a system. Activity diagrams address the dynamic view of a system. They are especially important in modelling the function of a system and emphasize the flow of control among objects.

8. ***Component diagram :*** A component diagram shows the organizations and dependencies among a set of components. Component diagrams address the static implementation view of a system. They are related to class diagrams in that a component typically maps to one or more classes, interfaces, or collaborations.

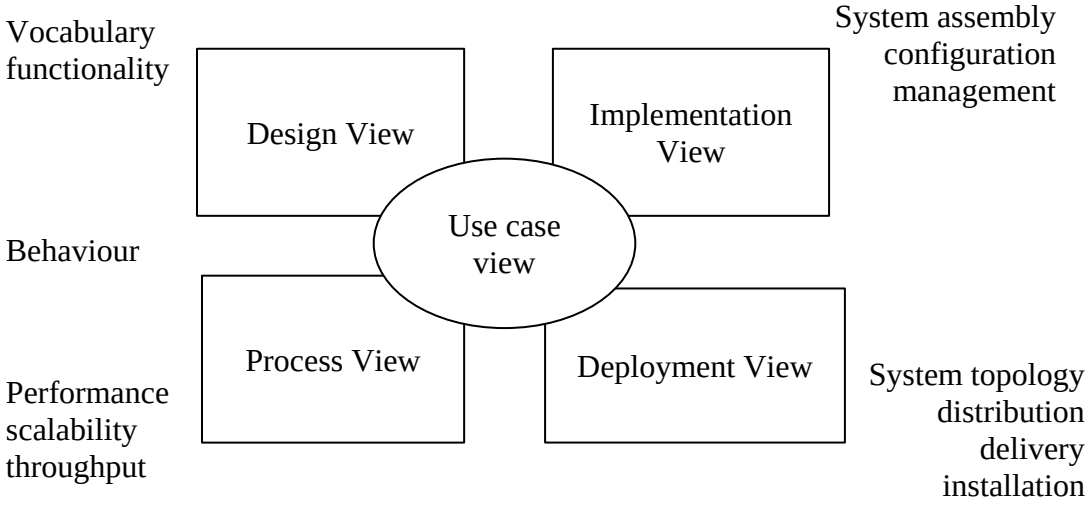
9. ***Deployment diagram :*** A deployment diagram shows the configuration of run-time processing nodes and the components that live on them. Deployment diagrams address the static deployment view of architecture. They are related to component diagrams in that a node typically encloses one or more components.

Check Your Progress

Which are the rules to be considered to form Collaboration diagrams?

11.5.4 ARCHITECTURE OF UML

Visualizing, specifying, constructing, and documenting a software-intensive system demands that the system be viewed from a number of perspectives. Different stakeholders- end users, analysts, developers, system integrators, testers, technical writers, and project managers- each bring different agendas to a project, and each looks at that system in different ways at different times over the project's life. A system's architecture is perhaps the most important artifact that can be used to manage these different viewpoints and so control the iterative and incremental development of a system throughout its life cycle.



Modelling a System's Architecture

Architecture is the set of significant decisions about

- The organization of a software system
- The selection of the structural elements and their interfaces by which the system is composed
- Their behavior, as specified in the collaborations among those elements
- The composition of these structural and behavioral elements into progressively larger subsystems

The architectural style that guides this organization: the static and dynamic elements and their interfaces, their collaborations, and their composition Software architecture is not only concerned with structure and behavior, but also with usage, functionality, performance, resilience, reuse, comprehensibility, economic and technology constraints and trade-offs, and aesthetic concerns. The architecture of a software-intensive system can best be described by five interlocking views. Each view is a

projection into the organization and structure of the system, focused on a particular aspect of that system.

The *use case view* of a system encompasses the use cases that describe the behaviour of the system as seen by its end users, analysts, and testers. This view doesn't really specify the organization of a software system. Rather, it exists to specify the forces that shape the system's architecture. With the UML, the static aspects of this view are captured in use case diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

The *design view* of a system encompasses the classes, interfaces, and collaborations that form the vocabulary of the problem and its solution. This view primarily supports the functional requirements of the system, meaning the services that the system should provide to its end users. With the UML, the static aspects of this view are captured in class diagrams and object diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams. The *process view* of a system encompasses the threads and processes that form the system's concurrency and synchronization mechanisms. This view primarily addresses the performance, scalability, and throughput of the system. With the UML, the static and dynamic aspects of this view are captured in the same kinds of diagrams as for the design view, but with a focus on the active classes that represent these threads and processes.

The *implementation view* of a system encompasses the components and files that are used to assemble and release the physical system. This view primarily addresses the configuration management of the system's releases, made up of somewhat independent components and files that can be assembled in various ways to produce a running system. With the UML, the static aspects of this view are captured in component diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

The *deployment view* of a system encompasses the nodes that form the system's hardware topology on which the system executes. This view primarily addresses the distribution, delivery, and installation of the parts that make up the physical system. With the UML, the static aspects of this view are captured in deployment diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

Each of these five views can stand alone so that different stakeholders can focus on the issues of the system's architecture that most concern them. These five views also interact with one another- nodes in the deployment view hold components in the implementation view that, in turn, represent the physical realization of classes, interfaces, collaborations, and active classes from the design and process views. The UML permits you to express every one of these five views and their interactions.

11.6 REPRESENTATION OF PROBLEMS USING OBJECT AND CLASS DIAGRAMS AT DESIGN LEVEL

A class describes a collection of similar objects. It is a template where certain basic characteristics of a set of objects are defined. A class defines the basic attributes and the operations of the objects of that type. Defining a class does not define any object, but it only creates a template. For objects to be actually created, instances of the class are to be created as per the requirement of the case.

Classes are built on the basis of abstraction, where a set of similar objects is observed and their common characteristics are listed. Of all these, the characteristics of concern to the system under observation are taken and the class definition is made. The attributes of no concern to the system are left out. This is known as abstraction. So, the abstraction is the process of hiding superfluous details and highlighting pertinent details in respect to the system under development.

ClassName
Attrubute-name1 : data-type1 = default-value1 Attrubute-name2 : data-type2 = default-value2
Operation-name1 (arguments2) : result-type1 Operation-name2 (arguments2) : result-type2

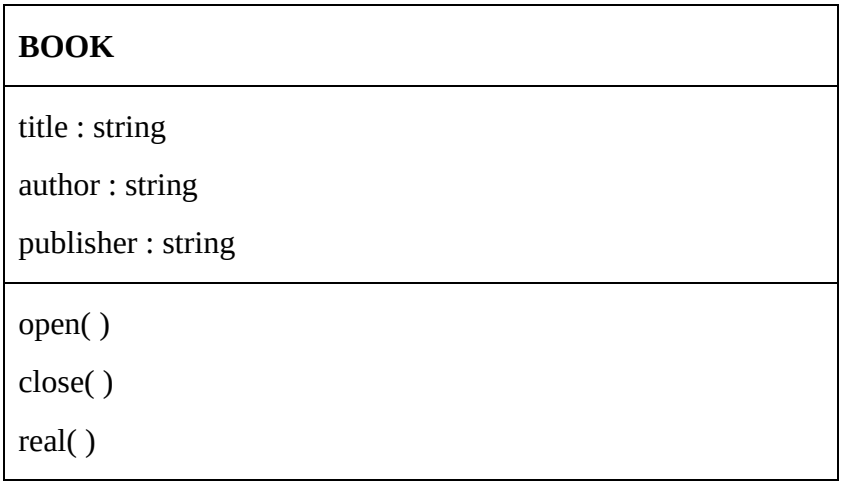
It should be noted that the abstraction of an object varies according to its application. For instance, while defining a pen class for a stationery shop, the attributes of concern might be the pen colour, ink colour, pen type etc., whereas a pen class for a manufacturing firm would be containing the other dimensions of the pen like its diameter, its shape and size etc. Each application-domain concept from the real world that is important to the application should be modelled as an object class. Classes are arranged into hierarchies sharing common structure and behaviour and are associated with other classes. This gives rise to the concept of inheritance.

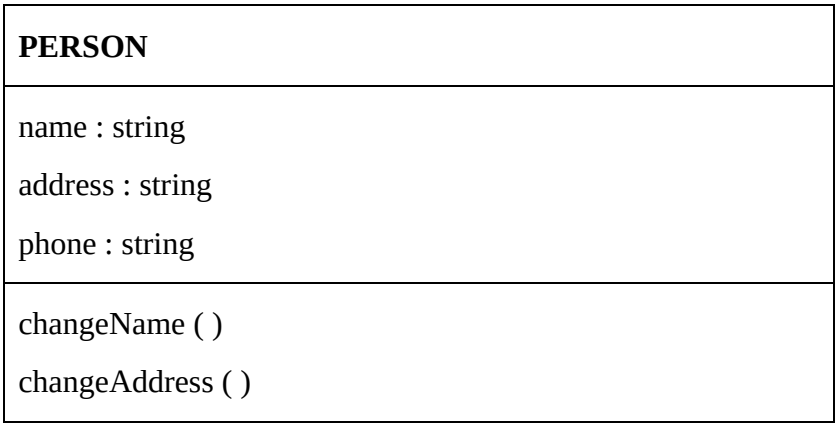
In OMT, classes are represented by a rectangular box which may be divided into three parts as shown in Figure. The top part contains the name of the class written in bold, middle part contains a list of attributes and bottom part contains a list of operations by the optional details such as type and default value. An object model should generally distinguish independent base attributes from dependent derived attributes. A **derived attribute** is that which is derived from other attributes. For example, age

is a derived attribute, as it can be derived from date-of-birth and current-date attributes. An operation is a function or transformation that may be applied to or by objects in a class. Operations are listed in the third part of the class box. Operations may or may not be shown; it depends on the level of detail desired. Each operation may be followed by optional details such as argument list and result type. The name and type of each argument may be given. An empty argument list in parentheses shows explicitly that there are no arguments. All objects in a class share the same operations. Each operation has a target object as an implicit argument.

An operation may have arguments in addition to its target object, which parameterize the operation. The behaviour of the operation depends on the class of its target. An operation may be polymorphic in nature. A polymorphic operation means that the same operation takes on different forms in different/same classes. Overloading of operators, overloading of functions and overriding of functions provided by object-oriented programming languages are all examples of polymorphic operations. A method is the implementation of an operation for a class. The method depends only on the class of the target object.

Figure shows the class Book. Attributes of Book are title, author, and publisher along with their data types. Operations on Book are open(), close(), read().





changePhone ( )

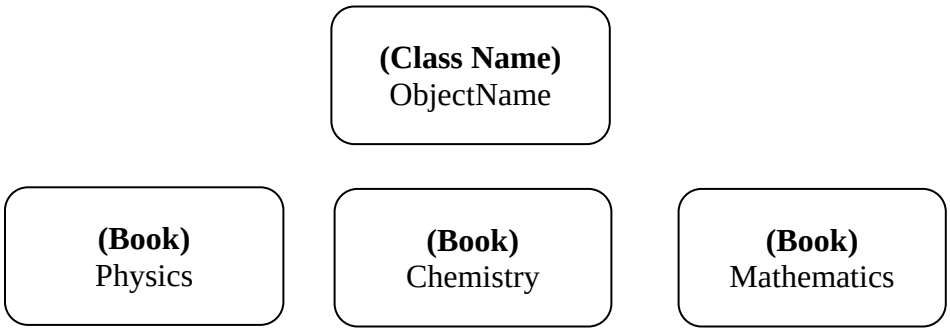
Now, let us formally define an object. An object is a concept, abstraction, or thing with crisp boundaries and meaning for the problem at hand.

An object has the following four main characteristics :

- Unique identification
- Set of attributes
- Set of states
- Set of operations (behaviour)

Unique identification, we mean every object has a unique name by which it is identified in the system. Set of attributes, we mean every object has a set of properties in which we are interested in. Set of states we mean values of attributes of an object constitute the state of the object. Every object will have a number of states but at a given time it can be in one of those states. Set of operations we mean externally visible actions an object can perform. When an operation is performed, the state of the object may change.

In other words, an object is an instance of an object class. Figure (rounded box) represents an object instance in OMT. Object instance is a particular object from an object class. The box may/may not be divided in particular regions. Object instances can be used in instance diagrams, which are useful for documenting test cases and discussing examples.



Attributes

An *attribute* is a data value held by objects in a class. Each attribute has a value for each object instance. This value should be a pure data value, not an object. Attributes are listed in the second part of the class box. Each attribute name may be followed by the optional details. An object modelshould generally distinguish independent *base attributes* from dependent *derived attributes*

Operations

An *operation* is a function or transformation that may be applied to or by objects in a class. All objects in a class share the same operations. Each operation has a target object as an implicit argument. An operation may have arguments in addition to its target object, which parameterize the operation. The behaviour of the operation depends on the class of its target.

A *polymorphic* operation means that the same operation takes on different forms in different classes.

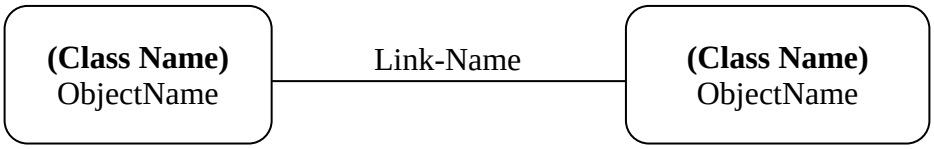
A *method* is the implementation of an operation for a class. The method depends only on the class of the target object. Operations are listed in the lower third of the class box. Each operation name may be followed by optional details, such as argument list and result type. The name and type of each argument may be given. An empty argument list in parentheses shows explicitly that there are no arguments.

Check Your Progress

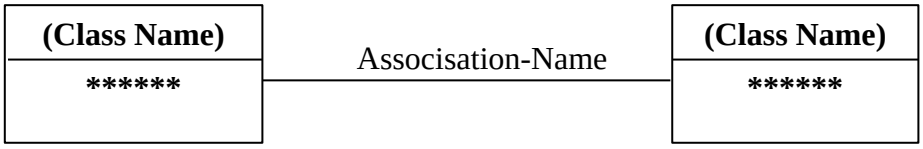
Which are part of class operation specification format?

Links and Associations

A link is a physical or conceptual connection between object instances. In OMT, link is represented by a line labelled with its name as shown in Figure



An association describes a group of links with common structure and common semantics between two or more classes. Association is represented by a line labelled with the association name in italics as shown in Figure

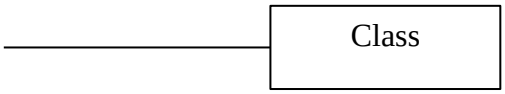
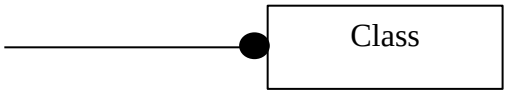
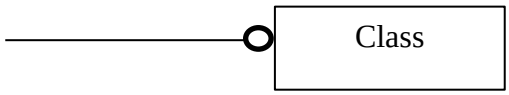
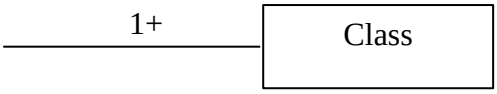
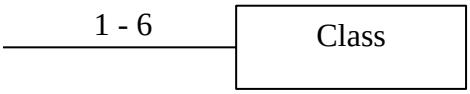
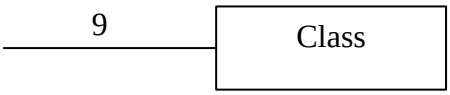
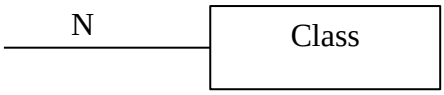


Association names are optional. If the association is given a name, it should be written above the line. Association names are italic. In case of a binary association, the name reads in a particular direction (i.e. left to

right), but the binary association can be traversed in either direction. For example, a pilot flies an airplane or an airplane is flown by a pilot. All the links in an association connect objects from the same classes. Associations are bidirectional in nature.

Multiplicity :

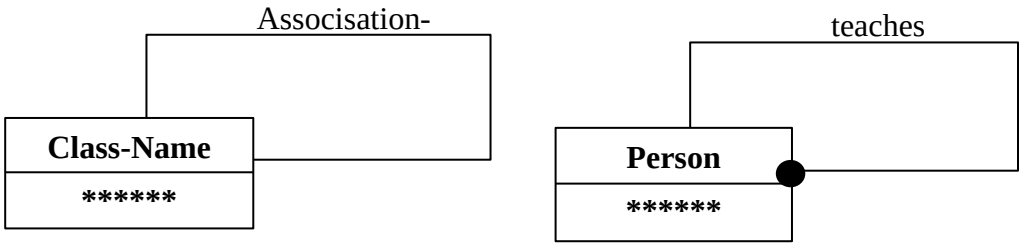
It specifies how many instances of one class may relate to a single instance of an associated class. Multiplicity constrains the number of related objects. There are special line terminators to indicate certain common multiplicity values. A solid ball is the symbol for "many", meaning zero, one or more. A hollow ball indicates "optional", meaning zero or one. The multiplicity is indicated with special symbols at the ends of association lines. In the most general case, multiplicity can be specified with a number or set of intervals. If no multiplicity symbol is specified that means a one-to-one association.

	Exactly one
	Many (zero or more) unlimited
	Optional (zero or one)
	One or more
	Specific range
	Exact Number
	Unlimited Number (Zero or more)
NOTATIONS FOR MULTIPILICITY	

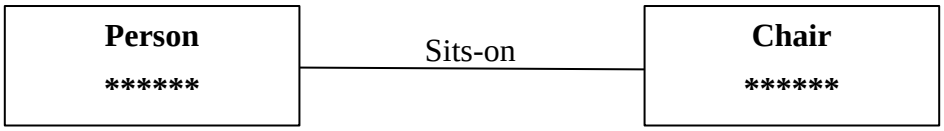
The rules of multiplicity are summarized below :

- Line without any ball indicates one-to-one association.
- Hollow ball indicates zero or one.
- Solid ball indicates zero, one or more.
- Numbers written on solid ball such as 1,2,6 indicates 1 or 2 or 6.
- Numbers written on solid ball such as 1+ indicates 1 or more, 2+ indicates 2 or more etc.

An association can be unary, binary, ternary or n-ary. Unary association is between the same class as shown in Figure

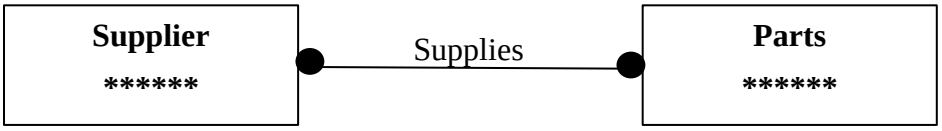


A binary association is an association between two classes as shown in Figure

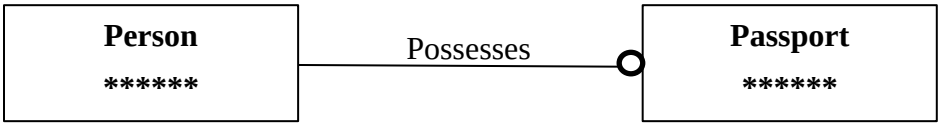


Example of a binary association is “Person sits on a Chair”. One person can sit at one chair. So multiplicity of this association is one-to-one as shown. Example of a binary association is “Supplier supplies Parts”. One supplier can supply many parts or one part can be supplied by

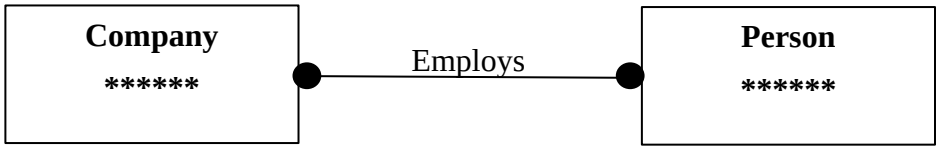
many suppliers. So multiplicity of this association is many-to-many as shown in Figure



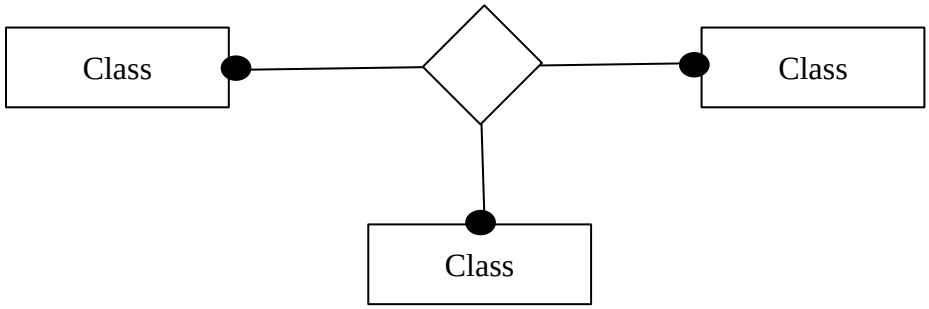
Another example of binary association is “Person possesses a Passport”. Either a person can have one passport or no passport but one passport can be with one person. So multiplicity of this association is one-to-optional as shown in Figure



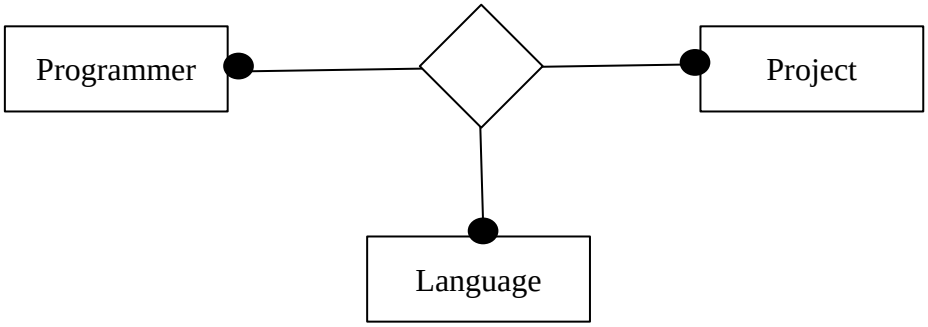
Another example of binary association is “Company employs Person”. One company can employ zero, one or more persons but one person can be employed in one company only (assume). So multiplicity of this association is one-to-many as shown in Figure



Ternary association is an association among three classes. On the same line, n-ary association is an association among n classes. The OMT symbol for ternary and n-ary associations is a diamond with lines connecting to related classes as shown in Figure. A name for the association is optional and is written next to the diamond. An n-ary associations cannot be subdivided into binary associations without losing information.



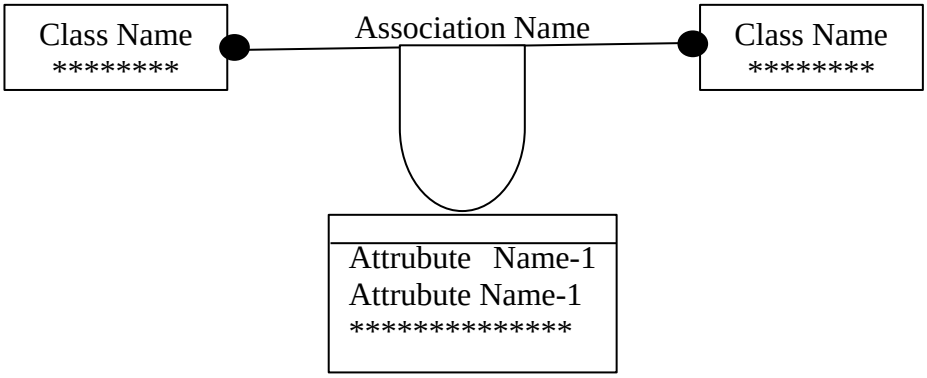
ow let us consider the example of a ternary association. Programmers develop Projects in (programming) Languages. One programmer can be engaged in zero; one or more projects and can know zero, one or languages. Similarly, one project can be developed by one or more programmers and in one or more languages. So this association along with its multiplicity is shown in Figure. Other examples of ternary and higher order associations are “Teacher teaches Students in a Classroom”, “Doctor diagnoses Patient in Room at a given Schedule” etc.



Association :

Association is a relationship between classes. There can be some attributes, which cannot be associated with either of the two classes related by an association. Such attributes are called as link attributes. A binary association can have two roles, which may be written at the ends of the association.

Other relationships between classes are aggregation and inheritance. Aggregation specifies one object may be composed of other objects. It is a part-whole relationship. Inheritance is a way to form new classes using classes that have already been defined. Inheritance is intended to help reuse existing code with little or no modification.



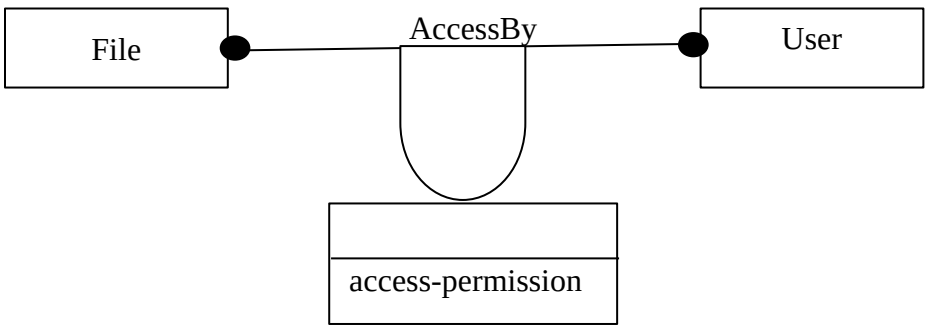
Check your Progress

What is multiplicity for an association?

Presentation of Contents

Link attributes

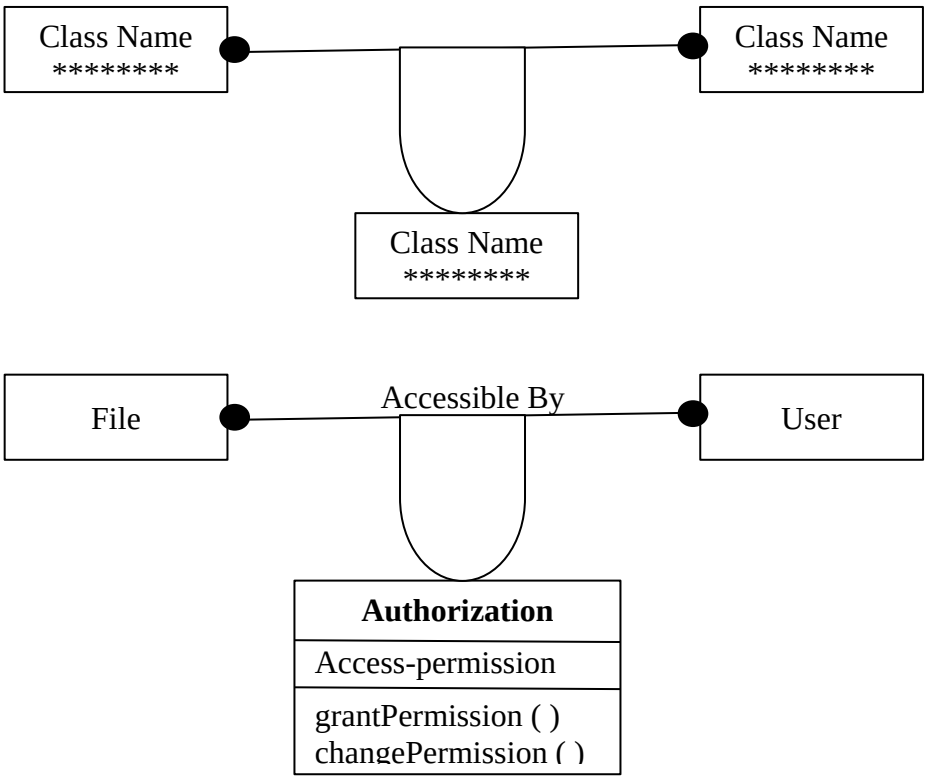
Sometimes, an attribute(s) cannot be associated with either of the two classes associated by the association. In such cases, the attribute(s) is associated with the association and is called as link attribute.



It is a property of the links in an association. The OMT notation for a link attribute is a box attached to the association by a loop, One or more link attributes may appear in the second region of the box. Sometime it is possible, for one-to-one and one-to-may associations, to fold link attributes into the class opposite to the "one" side. But as a rule, link attributes should not be folded into a class because future flexibility is reduced if the multiplicity of the association changes. Consider an example as shown in Figure File is accessed by a User. So, the classes File and User are related by association the association named “AccessedBy”. Many users can access one file and one user can access many files.

So, multiplicity of the association is many-to-many. Now, the attribute “access-permission” cannot be associated with either File class or with User class. This attribute can be associated with the link as shown in Figure. Hence, access-permission is link attribute.

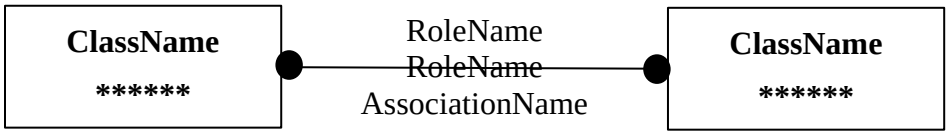
It is also possible to model an association as a class such class is called as link class as shown in Figure. Each link becomes one instance of the class. The notation for this kind of association is the same as for a link attribute and has a name and (optional) operations added to it.



Now, consider the example shown in Figure, where whole class is associated with the link. In this example, the class Authorization is a link class. It has one attribute “access-permission” and two methods grantPermission() and changePermission().

Role Names

A role is one end of an association. A binary association can have two roles, each of which may have a role name.



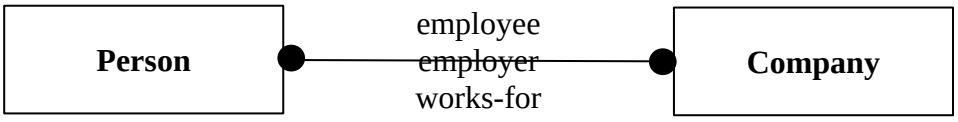
A role name is a name that uniquely identifies one end of an association. Roles provide a way of viewing a binary association as a traversal from one object to a set of associated objects. Each role on a binary association identifies an object or set of objects associated with an object at the other end. Figure shows how to represent roles in OMT methodology.

The use of role names is optional, but is often easier and less confusing to assign role names instead of, or in addition to, association names. Role names are necessary for associations between two objects of the same class.

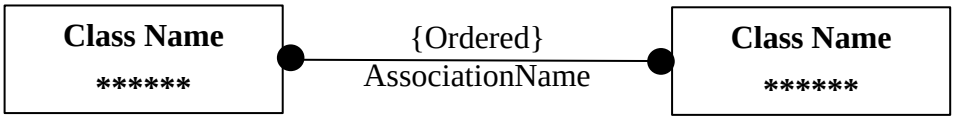
They are also useful to distinguish between two associations between the same pair of classes. We can follow these two guidelines: All role names on the far end of associations attached to a class must be unique. No role name should be the same as an attribute name of the source class. It is also possible to use role names for n-ary associations.

The role name is a derived attribute whose value is a set of related objects. Use of role names provides a way of traversing associations from an object at one end, without explicitly mentioning the association. For example, consider the association ‘a person works for a company’, in this employee and employer are role names for the classes Person and Company respectively as shown in Figure.

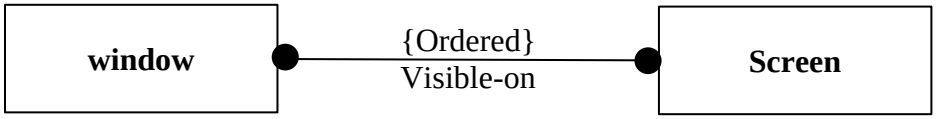
Ordering



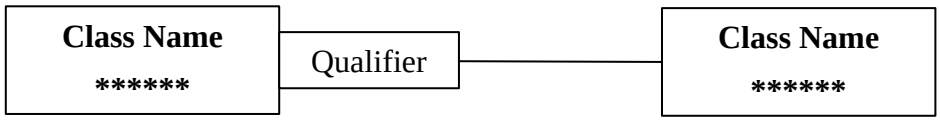
Usually the objects on the "many" side of an association have no explicit order, and can be regarded as a set. Sometimes the objects on the many side of an association have order. Writing {ordered} next to the multiplicity dot as shown in Figure indicates an ordered set of objects of an association.



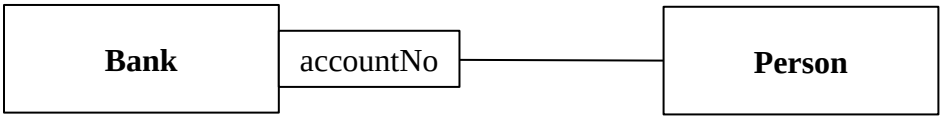
Consider the example of association between Window class and Screen class. A screen can contain a number of windows. Windows are explicitly ordered. Only topmost window is visible on the screen at any time. Figure shows this example.



Qualification



A qualifier is an association attribute. A qualified association relates two object classes and a qualifier. The qualifier is a special attribute that reduces the effective multiplicity of an association. One-to-many and many-to-many associations may be qualified. Figure shows how to represent a qualification.

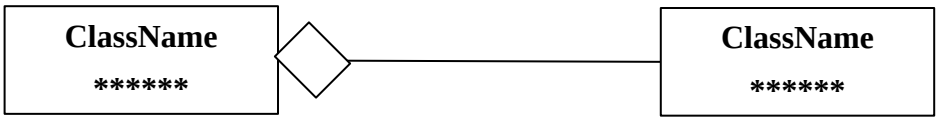


The qualifier is drawn as a small box on the end of the association line near the class it qualifies. The qualifier rectangle is part of the association, not of class. The qualifier distinguishes among the set of objects at the "many" end of an association. A qualified association can also be considered a form of ternary association. The advantage of the qualification is that it improves semantic accuracy and increases the visibility of navigation paths.

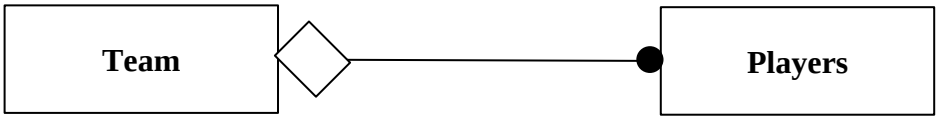
For example, a person object may be associated to a Bank object as shown in Figure. An attribute of this association is the accoutNo. The accountNo is the qualifier of this association.

Aggregation

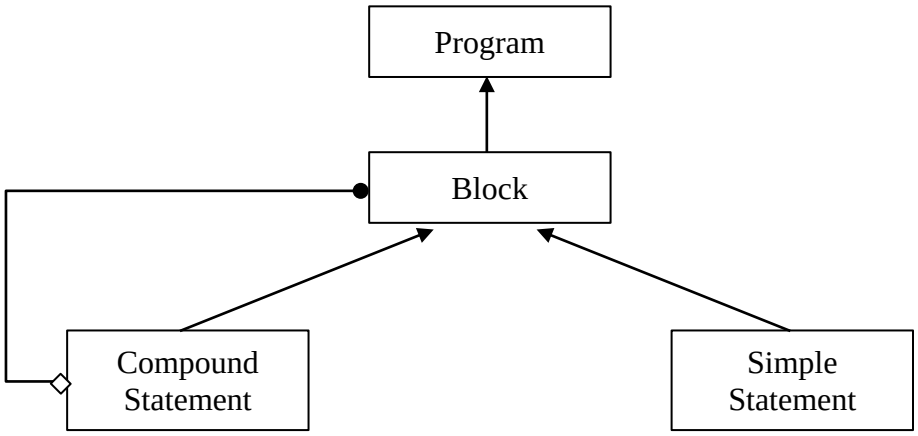
Aggregation is another relationship between classes. It is a tightly coupled form of association with some extra semantics. It is the “part-whole” or “a-part-of” relationship in which objects representing the component of something are associated with an object representing the entire assembly. Aggregations are drawn like associations, except a small hollow diamond indicating the assembly end of the relationship as shown in Figure. The class opposite to the diamond side is part of the class on the diamond side.



For example, a team is aggregation of players. This can be modelled as shown in Figure. Aggregation can be fixed, variable or recursive.



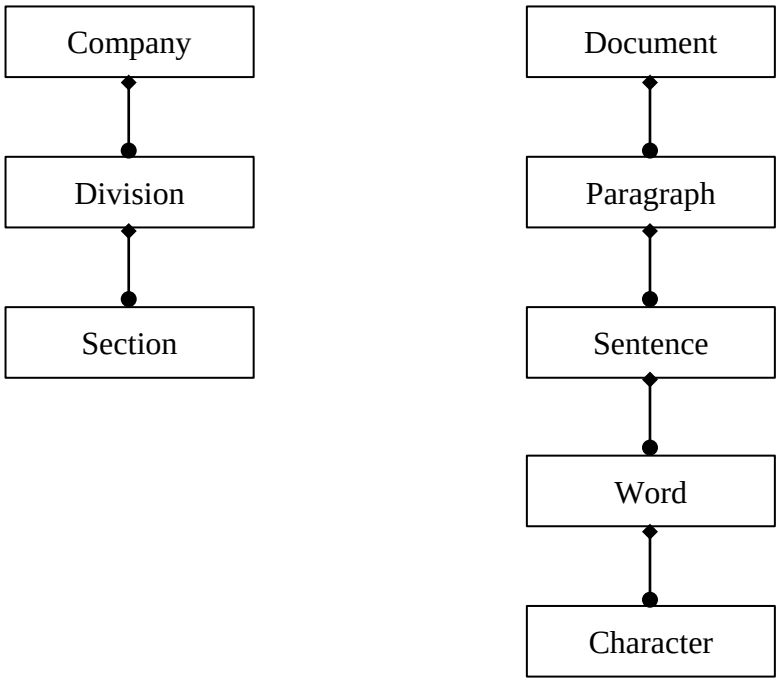
- In a fixed aggregation number and subtypes are fixed i.e. predefined.
- In a variable aggregation number of parts may vary but number of levels is finite.
- A recursive aggregate contains, directly or indirectly, an instance of the same aggregate. The number of levels is unlimited. For example, as shown in Figure, a computer program is an aggregation of blocks, with optionally recursive compound statements. The recursion terminates with simple statement. Blocks can be nested to arbitrary depth.



One more example of aggregation is shown in Figure. A company is composed of zero, one or more divisions. A division is composed of zero, one or more sections.

Another example of aggregation is shown in Figure. A document is composed of zero, one or more paragraphs. A paragraph is composed of

zero, one or more sentences. A sentence is composed of one or more words. A word is composed of one or more characters.



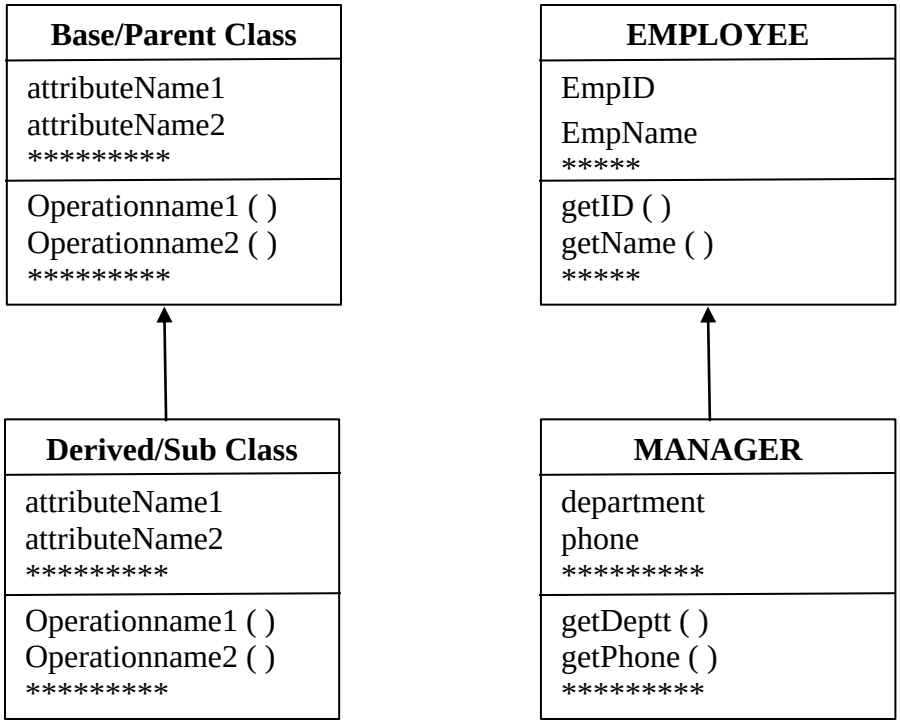
Check Your Progress

Whatis the stronger form of aggregation and how it is represented?

Inheritance

The inheritance concept was invented in 1967 for Simula. Inheritance is a way to form new classes using classes that have already been defined. Inheritance is intended to help reuse existing code with little or no modification. The new classes, known as derived classes (or child classes or sub classes), inherit attributes and behaviour of the pre-existing classes, which are referred to as base classes (or parent classes or super classes) as shown in Figure.

The inheritance relationship of sub- and super classes gives rise to a hierarchy. Inheritance is a “is-a” relationship between two classes.For example, Student is a Person; Chair is Furniture; Parrot is a Bird etc. in all these examples, first class (i.e. Student, Chair, Parrot) inherits properties from the second class (i.e. Person, Furniture, Bird).



Example of an inheritance : Manager is an Employee. Manager class inherits features from Employee class as shown in Figure. There are several reasons to use inheritance as enumerated below:

Inheritance for Specialization

One common reason to use inheritance is to create specializations of existing classes. In specialization, the derived class has data or behaviour aspects that are not part of the base class. For example, Square is a Rectangle. Square class is specialization of Rectangle class. Similarly, Circle is an Ellipse. Here also, Circle class is specialization of Ellipse class. Another example, a BankAccount class might have data members such as accountNumber, customerName and balance. An InterestBearingAccount class might inherit BankAccount and then add data member interestRate and interestAccrued along with behaviour for calculating interest earned.

Another form of specialization occurs when a base class specifies that it has a particular behaviour but does not actually implement the behaviour. Each non-abstract, concrete class which inherits from that abstract class must provide an implementation of that behaviour. This providing of actual behaviour by a subclass is sometimes known as implementation or reification. For example, there is a class Shape having operation area(). The operation area() cannot be implemented unless we have concrete class. So, Shape class is abstract class. Rectangle is a Shape. Now, Rectangle is a concrete class, which can implement the operation area().

Inheritance for Generalization

Generalization is reverse of specialization. For instance, a "fruit" is a generalization of "apple", "orange", "mango" and many others. One can consider fruit to be an abstraction of apple, orange, etc. Conversely, since apples are fruit (i.e., an apple is-a fruit), apples may naturally inherit all the properties common to all fruit, such as being a fleshy container for the seed of a plant.

Another example: Vehicle is a generalization of Car, Truck, Bus etc. Car, Truck, Bus etc. share some properties such as “number of wheels”, speed, capacity etc. these common properties are abstracted out and put into another class say Vehicle, which comes higher in the hierarchy.

Inheritance for Extension

In this case, inheritance extends the existing class functionalities by adding new operations in the derived class. It can be distinguished from generalization that the later must override at least one method from the base and the functionality is tied to that of the base class. Extension simply adds new methods to those of the base class and functionality is less strongly tied to the existing methods of the base class. For example, StringSet class inherits from Set class, which specializes for holding string values. Such a class might provide additional methods for string related operations – for instance - search by prefix, which returns a subset of all the elements of the set that begin with a certain string value. These operations are meaningful to the derived class but are not particularly relevant to the base class.

Inheritance for Restriction

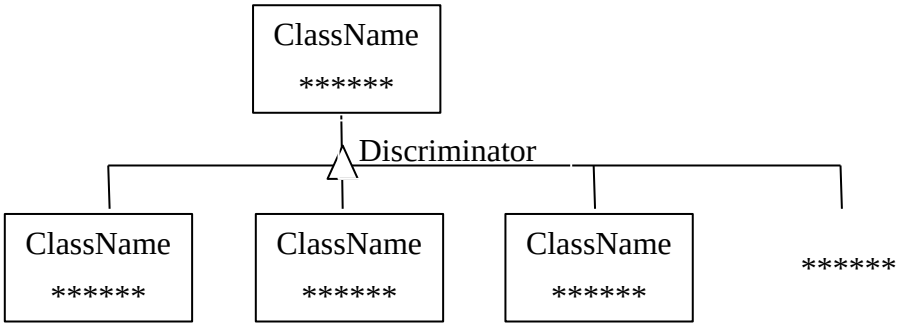
In this case, the derived class does not implement the functionality, which a base class has. In other words, inheritance for restriction occurs when the behaviour of the derived class is smaller or more restrictive than the behaviour of the base class. For example, an existing class library provides a double-ended queue (deque). Elements can be added or removed from either end of the deque, but the programmer wishes to write a stack class, enforcing the property that elements can be added or removed from only one end of the stack. Here, the programmer can make the Stack class a derived class of the existing Deque class and can modify or override the undesired methods so that they produce an error message if used.

Inheritance for Overriding

When a class replaces the implementation of a method that it has inherited is called overriding. Overriding introduces a complication: which version of the method does an instance of the inherited class use the one that is part of its own class, or the one from the parent (base) class. The answer varies between programming languages, and some languages provide the ability to indicate that a particular behaviour is not to be overridden.

Generalization and Inheritance

Generalization is the relationship between a class and one or more refined versions of it. The class being refined is called the *superclass* and each refined version is called a *subclass*. Attributes and operations common to a group of subclasses are attached to the superclass and shared by each subclass. Each subclass is said to*inherit* the features of its superclass. Generalization is sometimes called the "is-a" relationship, because each instance of a subclass is an instance of the superclass as well. The OMT notation for generalization is shown in figure.



The notation for generalization is a triangle connecting a superclass to its subclasses. The discriminator is an attribute of enumeration type that indicates which property of an object is being abstracted by a particular generalization relationship. Only one property should be discriminated at once. The discriminator is an optional part of a generalization relationship. It is not good to nest subclasses too deeply, because they can be very difficult to understand.

Rules for a correct Object diagram :

An Object Diagram should comply with the following rules to be a correct diagram:

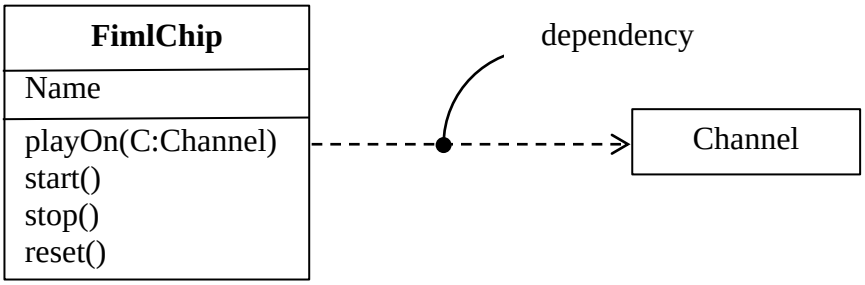
- Attributes should be pure data values, not objects;
- No classes with the same names
- Associations should be at least between 2 classes
- No attributes & operations with the same names in one class
- Do not bury pointers or other object references inside objects as attributes. Instead model these as associations;
- Link attributes must not be collapsed into classes;
- Every subclass has at least one superclass
- Each operation must have a target object as an implicit argument
- Object models should be documented;

Relationships : When you build abstractions, you'll discover that very few of your classes stand alone. Instead, most of them collaborate with others in a number of ways. Therefore, when you model a system, not only must you identify the things that form the vocabulary of your system, you must also model how these things stand in relation to one another.

In object-oriented modelling, there are three kinds of relationships that are especially important: *dependencies,* which represent using relationships among classes (including refinement, trace, and bind relationships); *generalizations,* which link generalized classes to their specializations; and *associations,* which represent structural relationships among objects. Each of these relationships provides a different way of combining your abstractions. Building webs of relationships is not unlike creating a balanced distribution of responsibilities among your classes. Over-engineer, and you'll end up with a tangled mess of relationships that make your model incomprehensible; under-engineer, and you'll have missed a lot of the richness of your system embodied in the way things collaborate.

A *relationship* is a connection among things. In object-oriented modelling, the three most important relationships are dependencies, generalizations, and associations. Graphically, a relationship is rendered as a path, with different kinds of lines used to distinguish the kinds of relationships.

Dependency: A *dependency* is a using relationship that states that a change in specification of one thing (for example, class Event) may affect another thing that uses it (for example, class Window), but not necessarily the reverse. Graphically, a dependency is rendered as a dashed directed line, directed to the thing being depended on. Use dependencies when you want to show one thing using another.

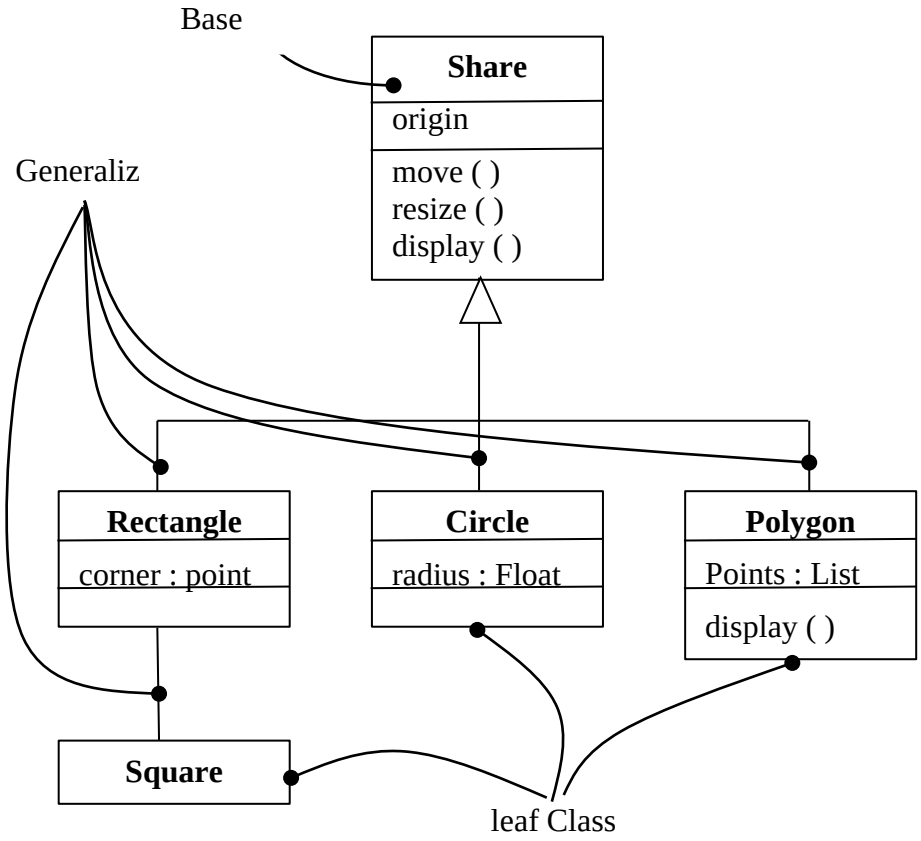


A dependency can have a name, although names are rarely needed unless you have a model with many dependencies and you need to refer to or distinguish among dependencies. More commonly, you'll use stereotypes to distinguish different flavours of dependencies.

Generalization : A *generalization* is a relationship between a general thing (called the superclass or parent)and a more specific kind of that thing (called the subclass or child). Generalization is sometimes called an "is-a-kind-of" relationship: o modelling (like the class BayWindow) is-a-kind-of a more general thing (for example, the class Window). Generalization means that objects of the child may be used anywhere the parent may

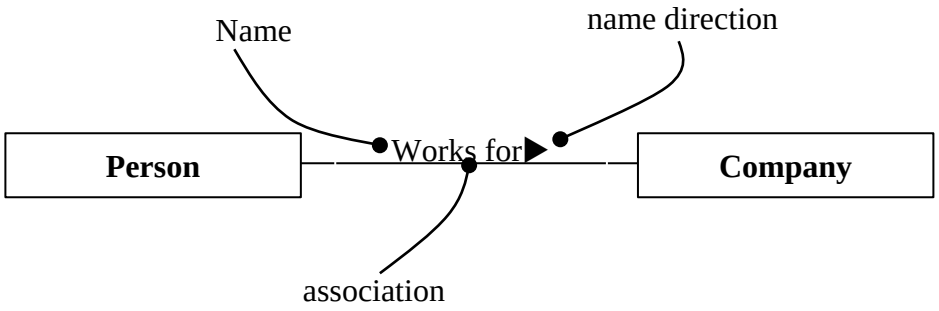
appear, but not the reverse. In other words, generalization means that the child is substitutable for the parent. A child inherits the properties of its parents, especially their attributes and operations. Often- but not always- the child has attributes and operations in addition to those found in its parents. An operation of a child that has the same signature as an operation in a parent overrides the operation of the parent; this is known as polymorphism. Graphically, generalization is rendered as a solid directed line with a large open arrowhead, pointing to the parent. Use generalizations when you want to show parent/child relationships.

A class may have zero, one, or more parents. A class that has no parents and one or more children is called a root class or a base class. A class that has no children is called a leaf class. A class that has exactly one parent is said to use single inheritance; a class with more than one parent is said to use multiple inheritance.



Association : *Associations and dependencies (but not generalization relationships) may be reflective.* An *association* is a structural relationship that specifies that objects of one thing are connected to objects of another. Given an association connecting two classes, you can navigate from an object of one class to an object of the other class, and vice versa. It's quite legal to have both ends of an association circle back to the same class. This means that, given an object of the class, you can link to other objects of the same class. An association that connects exactly two classes is called a binary association. Although it's not as common, you can have associations that connect more than two classes; these are called n-ary

associations. Graphically, an association is rendered as a solid line connecting the same or different classes. Use associations when you want to show structural relationships. Although an association may have a name, you typically don't need to include one if you explicitly provide role names for the association, or if you have a model with many associations and you need to refer to or distinguish among associations. This is especially true when you have more than one association connecting the same classes.



Check Your Progress

By which termshigher-level and lower –level entities are designated in generalization.

11.8 SUMMARY

Modelling gives us the ability to visualise the system before it is implemented. It also gives the convection that system that is to be implemented is analysed properly. There are various diagrams with which we model our system. UML is the universal language for modelling of object oriented systems. It also gives us the template to guide us in constructing a system. We can easily simulate the real life problems of object objected systems. Class diagram, object diagram, use case diagram etc. are the various diagrams using in this process.

11.9 EXERCISE

1. Explain Object Oriented modelling with its need.
2. What are the principles of modelling?
3. How the real life problems are modelled in object oriented system. Explain the basic building blocks used in this representation.
4. Give an example of object oriented system and represent it using object oriented modelling.
5. Explain the class diagram and object diagram with example.

ROUGH WORK

ROUGH WORK