

स्वाध्याय

स्वमन्थन

स्वावलम्बन

UTTAR PRADESH RAJARSHI TANDON OPEN UNIVERSITY
(Established vide U.P. Govt. Act No. 10, of 1999)



Indira Gandhi National Open University



UP Rajarshi Tandon Open University

BCA- 17
C++ And Object Oriented
Programming

**FIRST BLOCK : An Introduction to Object
Oriented Programming**
SECOND BLOCK : C++ - An Introduction

Shantipuram (Sector-F), Phaphamau, Allahabad - 211013

COURSE INTRODUCTION

Object Oriented Programming is one of the ways to manage the complexity of programming through computers. The machine and assembly language were not highly suitable for very complex programs/applications. Then, came high-level procedural languages such as FORTRAN, BASICS, PASCAL and C. These languages eliminated the close ties to the machine instructions. Most recent applications were written in a combination of these high-level languages and assembly languages. The programming involved the data structures and application was a collection of procedures that manipulate these structures.

This procedural approach to programming has worked well, however, the complexity of software is ever increasing with more powerful computer hardware. Any new applications today include features such as a window based graphical user interface, access to data stored in mainframe computers, and the ability to work in a networked environment. This complexity has forced the programmers to adopt a new programming paradigm: the object oriented programming (OOP). It is a new way of organising code and data that enables increased control over the complexity of the software development process.

The best way to learn C++ may be to learn Object Oriented Programming Concepts first and then implement such concepts using C++. Therefore, this course has been divided into two blocks.

The first block deals with the basic OOPs terminology such as abstraction, inheritance, and polymorphism. In addition, it provides an overview of various object oriented programming languages. It also provides a brief introduction to object oriented design. Block 2 covers the C++ is a programming language and how does it supports the object oriented programming paradigm.

Please note that you must write C++ classes and programs and run them in order to gain more in this course during your practical counselling sessions. Some of the suggested problems that you can attempt during those sessions are:

1. Write a function in C++ for swapping the value of two strings using reference parameters.
2. What is "this" pointer in C++? Give an example. Where is it used? Write example programs using "this" pointer.
3. Design a class to represent rectangle in C++. The basic functions to be designed in addition to constructor and destructor are: to find whether a given point is inside or outside or on the boundary of the rectangle. Also design the function for outputting a rectangle in visual form. The rectangle in visual form is to be filled with a colour and may be transparent or opaque.
4. Design a class ARRAY using pointers. Write a default constructor, copy constructor and overloaded assignment operator for the class. Also write the explicit destructor.
5. Design a class hierarchy for shapes of different types such as square, rectangle and circle. Write at least one polymorphic function. Show the run-time polymorphism by calling the polymorphic function from main().
6. Design a linked list as a template class.
7. Show multiple inheritance by an example.
8. Describe various access methods using examples.
9. Describe the use of "files" and "streams" classes in C++ using examples.

You may write more such similar Programs.

BLOCK INTRODUCTION

This being the first block of the course, an attempt has been made to define and consolidate concepts with the help of examples. The important concepts that one must be able to describe have been discussed in the block. Please note that this block is the backbone for your practical implementations, therefore, must be given maximum attention.

One must be clear about these basic concepts in order to use a lot of functions and facilities, which does exist in an Object Oriented System. This block describes the concept of Object Oriented Programming and introduces the concept of Object oriented design. The basic focus of the block being that you should be able to design basic objects prior to using an Object oriented Programming Language. This will facilitate you in better design and use of classes and the inheritance hierarchy.

This block consists of five units:

Unit 1 defines the basic concepts relating to objects and object oriented programming.

Unit 2 focuses on the various terminology of object oriented programming system.

Unit 3 provides the focus on the advanced concepts of object oriented systems such as dynamism and reusability.

Unit 4 provides an overview of various object oriented programming languages.

Unit 5 focuses on object oriented design. This unit provides basic information on a Object oriented design methodology: The Unified Modeling Language (UML).

Further Readings: Some of the important text books in these areas being:

1. B. Stroustrup: Object Oriented Programming in C++, Pearson Publication.
2. Barkakati, Object Oriented Programming in C++, PHI.

UNIT 1 WHAT IS OBJECT ORIENTED PROGRAMMING ?

Structure

- 1.0 Introduction
- 1.1 Objectives
- 1.2 Object Oriented Programming paradigm
 - 1.2.1 Object: The Soul of Object Oriented Programming
 - 1.2.2 Object Oriented Programming Characteristics
- 1.3 Advantages of Object Oriented Programming
- 1.4 Some Applications of Object Oriented Programming
 - 1.4.1 System Software
 - 1.4.2 DBMS
- 1.5 The Object Orientation
- 1.6 Object Oriented Languages
 - 1.6.1 Why C++ Succeeded
 - 1.6.2 Advantages of C++
- 1.7 Summary
- 1.8 Model Answers

1.0 INTRODUCTION

When Computers were initially introduced, the Engineers purely operated them. For a layman using computers was something like flying an aircraft so, the genesis of the computer revolution was in a complex machine. The genesis of our programming languages, thus, tends to look like that machine. However computers have evolved as user-friendlier tool, which is not so much of a machine, but a real world friend that support human expression. As a result, the tools are beginning to look less like machines and more like parts of our minds, and also like other expressive mediums such as writing, painting, sculpture, animation, and filmmaking. Object-oriented programming is part of this movement toward using the computer as an expressive medium.

This unit is an attempt to introduce you what is an object-oriented paradigm and why the industry should move from a procedural paradigm of programming to an object oriented paradigm.

1.1 OBJECTIVES

At the end of this unit, you should be able to:

- Describe the term: object, the heart of Object Oriented languages;
- Compare and contrast an object oriented language from a procedural language;
- Describe the advantages of using object oriented programming;
- Decide when to use procedural language and when to use a object oriented language; and

- Discuss the advantages and disadvantages of using C++, an object oriented language.

1.2 OBJECT-ORIENTED PROGRAMMING PARADIGM

Whenever we have a problem in hand, we have a very natural tendency to differentiate the Problem Space and Solution Space of the problem, i.e. the place where the problem exists and the place where we try to find out answer. When we use Computer to solve a problem, then Computer is the "Solution Space" i.e. the place where you model that problem and the "problem space" is the place where the problem that is being solved exists. All programming languages offer some level of abstraction, and the complexity level for its solution space. A problem that you are able to solve by any Programming language is directly related to the kind and quality of abstraction done by it. The term "Kind" in this context implies:

What is it that you are abstracting?

For example, Assembly language is an abstraction of the machine instruction set. Many higher level languages (such as Fortran, BASIC, and C) were abstractions of assembly language. These languages were improvements over assembly language, but their primary underlying abstraction model still required you to think in terms of the structure of the computer rather than the structure of the problem you are trying to solve. It was the job on the programmer to establish the association between the machine and the Problem by proposing the suitable modules, data structures and algorithms. The effort required to perform this mapping is extrinsic to the programming language. Also this kind of mapping produces programs that are difficult to write and expensive to maintain.

Thus, the focus of such programming paradigms is on processing, that is, the algorithms were needed to perform the desired computation on structured data. The programming languages support this paradigm by providing functions and facilities for passing arguments to these functions and returning values from functions. In other words, emphasis was to:

Decide the structure of data.

Decide which procedures are required. (What is needed?)

Use the best algorithm available (How to achieve it?)

For example, A Square root function is:

Given a double-precision floating-point argument, it produces the square root.

```
double sqrt (double arg)
```

```
{
```

```
// code for calculation
```

```
}
```

```
void f() - function does not return a value
```

```
{
```

```
double root2 = sqrt(2);
```

```
// Get and then print the square root
```

```
}
```

Code written within curly brackets express group. They indicate start and end of function bodies. From the point of view of programming, functions are used to create order in a maze of algorithms.

The alternative to modeling the machine is to model the problem you are trying to solve. The object-oriented approach goes a step further by providing tools for the programmer to represent solution entities with respect to the problem space. This representation is general enough that the programmer is not constrained to any particular type of problem. We refer to the entities in the problem space and their representations in the solution space as "objects." The idea is that the program should be allowed to adapt itself to the terminology used for the problem. It may appear to be a more flexible and powerful language abstraction than what you have had before. Thus, the idea behind Object Oriented Programming is to allow you to describe the problem in the terminology of the problem, rather than in terms of the computer where the solution will run. There is still a connection back to the computer, but how? That is what we are going to discuss further.

All the Programming languages have traditionally divided the world into two parts—data and operations on data. The data is a static entity and can be changed only by the valid operations. The functions that can operate on data has a finite life cycle of its own and can affect the state of data over their lifetime. Such a division is, of course, on the basis the way computers works. The operations or functions have meaning only when they can act on data or modify it. At some point, all programmers - including object-oriented programmers - must lay out the data structures that their programs will use and define the functions that will act on the data.

A procedural programming language like C, may offer various kinds of support for organising data and functions. Functions and data structures are the basic elements of procedural design. But Object-oriented programming tries to model the design of the program as real world philosophy. It groups operations and data into modular units called objects and lets you combine objects into structured networks to form a complete program. In an object-oriented programming language, objects and object interactions are the basic elements of design.

1.2.1 Object: The soul of Object Oriented Programming

Object-oriented Programming and Design is all about objects. Traditionally, code and data are apart. For example, in the "C" language, units of code are called functions or operations, while units of data are called structures. Functions and structures are not formally connected in "C". A "C" function can operate on more than one type of structure, and more than one function can operate on the same structure. However, it is not true for object-oriented programs. In Object Oriented Programming, the data and the operations are merged into a single indivisible unit - an Object.

An object has both state (data) and behavior (operations on data). In that way objects are not much different from ordinary physical entities. It is easy to see how a mechanical device embodies both state and behavior. For example, a simple non moving entity: an ordinary bottle combine state (how full the bottle is? is it open? how much warm its contents are?) with behavior (the ability to dispense its contents at various flow rates, to be opened or closed, to withstand temperatures). It is this resemblance of objects to real things that provides object's much of their power and appeal. They not only can model components of real systems, but also fulfil assigned roles as components in software systems. Therefore, object based programming scheme have advantages, we will discuss them later in this unit.

Objects are the physical and conceptual things we find in the world. Hardware, software, animals, and even concepts are all examples of objects. Everyone's world is built on Objects. For example, for a nuclear scientist plutonium, atoms, their speed are all objects for him. For a civil engineer bricks, columns, labour are the objects. Finally, for a software engineer developing windows based program windows, menus, buttons etc. are the objects for him. An object has its well-defined boundary in which it performs its functions while interacting with other objects with its external interface. Objects interact with each other via messages. (Please refer to figure 1). The concept of object being an entity can be described as, when we refer to some Object in real world we know it will be in some state (in time and place), for example, in a nuclear scientist's world the atom will always be in one or the other kind of state (static or moving and intact or exploded).

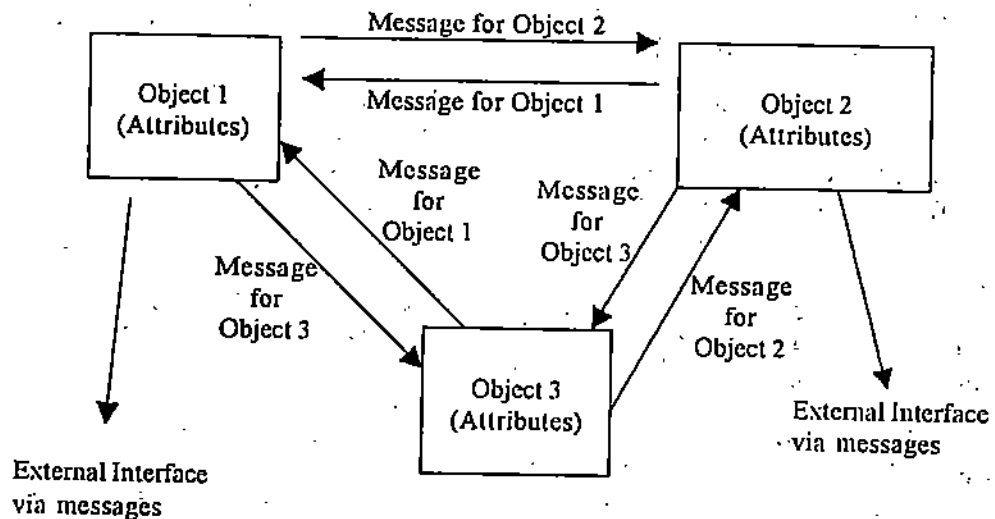


Figure 1: Objects and interfaces.

1.2.2 Object Oriented Programming Characteristics

The fundamental concept of Object Oriented Programming is that it allows combination of data and functions/methods/procedures which are working on that data, which did not exist in earlier procedure, based programming paradigms.

This fundamental unit is called Object. An Object's has a well-defined interface, which is the only way to access the Object's data. The data is thus well organized and hidden. Such hidden data is referred to as encapsulated. Data encapsulation and data hiding are basic and key terms used in OOPS.

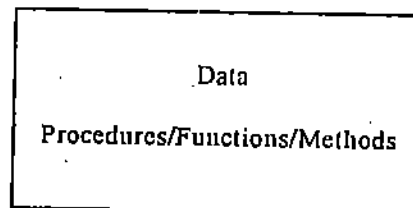


Figure 2: An Object

An Object Oriented programming system is composed of multiple objects (See Figure 3). When one object needs information from another object, a request is sent asking for specific information, (for example, a report object may need to know what is the today's date and will send a request to the date object).

These requests are called messages and each object has an interface that manages messages.

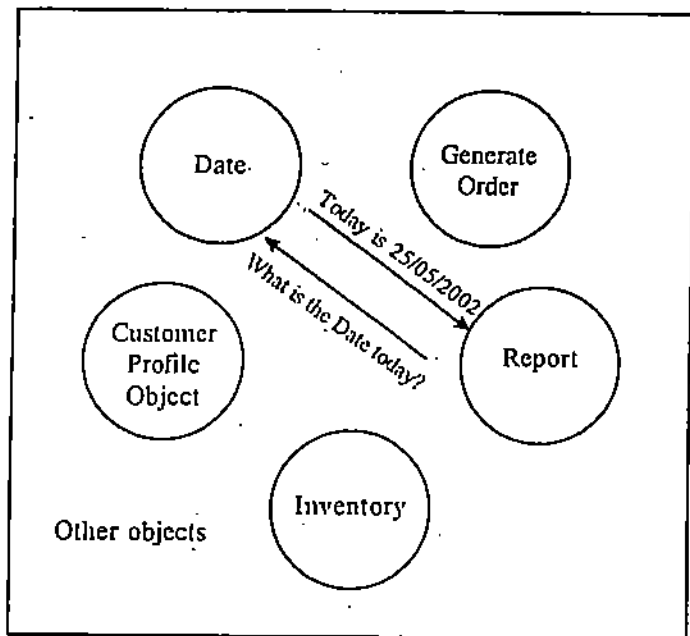


Figure 3: The object system and messages

A primary rule of object-oriented programming paradigm is that:

As the user of an object, you should never need to peek inside it."

Why should you not look inside an object?

All communications among the objects is done via messages. Messages define the interface to the object. The object that a message is sent to is called the receiver of the message. Everything an object can do is represented by its message interface. So you need not know anything about what is in the object in order to use it.

If you look inside the object, it may tempt you and you would like to tamper with the details of how the object works. Suppose you have changed the object and later the person who programmed and designed the object in the first place decided to change some of these details, then you may be in trouble. Your software may not be working correctly. But as long as you just deal with objects via their messages, the software is guaranteed to work. Thus, it is important that access to an object is provided only through its messages, while keeping the details hidden.

But why should we be concerned about the changes in the object design? Because software engineering experiences have taught us that software do change. A popular saying is that "Software is not written, it is re-written." Please remember that some of the costliest mistakes in computer history are because of software that failed when someone tried to change it.

So, that is an object. Let us now try to identify the basic characteristics of Object Oriented Programming?

The basic characteristics of any object-oriented language representing a pure approach to object-oriented programming are:

The basic Programming entity is the Object. An object can be considered to be a variable that stores data and can perform operation on the stored data itself.

2. An object oriented Program is a collection of objects for solving a problem. These objects send messages to each other. A message can be equated to a request to call a function of the receiver object.
3. Each object has its own memory or data that may be made up of other objects. Thus, object oriented Programs are suitable for Complex problem solving as they hide the complexity behind the simplicity of objects.
4. Each object can be related to a type, which is its class. An important consideration of a class is that it specifies the message interface that is the messages that can be send to that type/class of the objects.
5. All object of a particular class can receive the same messages but may behave differently. This leads to an important conclusion. Let us take an example, a circle object having center at $x=0$ and $y=0$ and a radius of 1 cm is of the class circle. However, it is also of the type shape. Thus, this object is bound to accept the messages that can be send to class shape. Similarly, a rectangle object is also of type rectangle and also of type shape and will follow messages send to class shape. Both these objects may be handled using the type shape, but may respond to a message differently on receiving the same message. This is one of the most powerful concepts of an Object oriented programming languages. The concept involves the concepts of inheritance and polymorphism. These concepts are discussed in unit 2 of this block.

1.3 ADVANTAGES OF OBJECT ORIENTED PROGRAMMING

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

Modular Design: The software built around OOP are modular, because they are built on objects and we know objects are entity in themselves, whose internal working is hidden from other objects and is decoupled from rest of the program.

Simple approach: The objects, we know, model real world, which results in simple program structure.

Modifiable: Because of its inherent properties of data abstraction and encapsulation (discussed in unit 2) the internal working of objects is hidden from other objects, thus any modification made to them should not affect rest of the system.

Extensible: The extension to the existing program for its adaptation to new environment can be done by simple adding few new objects or by adding new features in old classes/types.

Flexible: Software built on Object Oriented Programming, can be flexible in adapting to different situations because interaction between objects does not affect the internal working of objects.

Reusable: Objects once made can be reused in more than one program.

Maintainable: Objects are separate entities, which can be maintained separately allowing fixing of bugs or any other change easily.

Check Your Progress 1

1) An Object contains data and methods. Even a program written in C have those; then how is object oriented programming different than procedural programming.

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

2) How does two-object communicate?

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

3) What are the three most important advantages of OOP?

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

1.4 SOME APPLICATIONS OF OBJECT ORIENTED PROGRAMMING

OOPS has wide following since its inception, it is not only a programming tool, but also a whole modeling paradigm. In addition to general problem solving two of the upcoming object oriented paradigms that are emerging very fast are:

1.4.1 System Software

As an object-oriented operating system, its architecture is organized into frameworks of objects that are hierarchically classified by function and performance. By that we mean that the whole Operating system can be found as made up of objects. The Object Oriented Programming has been a great help for Operating system designers; it allowed them to break the whole Operating system into simple and manageable objects. It allowed them to reuse existing codes by putting similar objects in related classes. KDE (a well known desktop of Linux) developers have extensively used the concepts of Object Oriented Programming. Linux Kernel itself is a well known application of Object Oriented Programming.

1.4.2 DBMS

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

OODBMS have the following features.

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

Applications of OODBMS

The areas of OODBMS applications are:

- CASE
- CAD
- CAM
- Telecommunications
- Healthcare
- Finance
- Multimedia
- Text/document/quality management

Advantages of OODBMS

The objects do not require re-assembling from their component tables (in which they are initially stored) each time they are used thereby reducing processing overheads by increasing access speeds.

Paging is reduced.

Versioning is easier.

Navigation through the database is easier and more natural, with objects able to contain pointers to other objects within the database.

Reuse reduces development costs.

Concurrency control is simplified by the ability to place a single lock on an entire hierarchy.

Better data model as based on the 'real world' instead of the 'flattened' relational model.

Relationships and constraints on objects can be stored in the server application rather than the client application, therefore, any changes need only be made in one place, thus, reducing the need for and risks involved in making multiple changes.

Disadvantages of OODBMS

Late binding (discussed in unit 3); which may cause extensive searches through the inheritance hierarchies, may reduce speed of access.

There are as yet no formal semantics for OODBMS. Relational databases can be 'proved' correct by means of set theory and relational calculus.

The simplicity of relational tables is lost.

1.5 THE OBJECT ORIENTATION

Suppose, you want to add two number say, 1 and 2, in an ordinary, non-object-oriented computer language like C. You might write this as:

```
a = 1;
b = 2;
c = a + b;
```

The above code implies that take a number 'a', which has the value 1, and number 'b', which has the value 2, and add them together using the C language's built-in addition capability. Take the result, which happen to be 3 in this case, and places it into the variable called 'c'

Now, here's the same thing expressed in C++, which is a pure object-oriented language:

```
a = 1;
b = 2;
c = a + b;
```

You must be wondering that the above code looks exactly the same. You are right, looks the same, but the meaning is dramatically different.

In C++, this says, Take the object 'a' which has the value 1, and send it the message "+", which includes the argument 'b' which, in turn, has the value 2. Object 'a', receives this message and perform the action requested, which is to add the value of the argument to itself. Create a new object, give this the result, which in this case is 3 and assign this object to 'c'.

The reason is that objects greatly simplify matters when the data get more complex. Suppose you wanted a data type called list, which is a list of names. In C, list would be defined as a structure.

```
struct list {
<definition of list structure data here>
};
```

```
list a, b, c;
a = "Object Oriented";
b = "Programming";
```

Let's try to add these new a and b in the C language:

```
c = a + b;
```

Will it work? No. The C compiler will generate an error when it tries to compile this because it does not understand what to do with addition of two strings. C compilers just understand how to add number, but a and b are not numbers.

One can do the same thing in C++, but this time, list is defined and implemented as a class called a "String".

```
list a, b, c;
a = "Object Oriented";
```

```
b = "Programming";
```

```
c = a + b.
```

The first three lines simply create List objects 'a' and 'b' from the given strings. The addition may work if the list class was created with a function/method which specifically "knows" how to handle the message "+". For example, the message plus might simply be used for concatenation of two strings. Thus, the value of C may be—

"Object Oriented Programming"

Using Non-Object-Oriented Languages

It is also possible to use objects and messages in non-object-oriented languages. This is done using function calls. Among other things, such an implementation allows sophisticated client-server software to run "transparently" from within ordinary programming languages.

Suppose you want to add a "plus" function to a C program:

```
int plus(int arg1, int arg2)
```

```
{ return (arg1 + arg2); }
```

This has not really bought you anything yet. But suppose that instead of doing the addition on your own computer, you automatically sent it to a server computer to be performed:

```
int plus(int arg1, int arg2)
```

```
{ return server_plus(arg1, arg2); }
```

The function server_plus () in turn creates a message containing arg1 and arg2, and sends this message, via a network, to a special object which sits on a server computer. This object executes the "plus" function and sends the result back to you. It is an object-oriented computing via a back-door approach.

1.6 OBJECT-ORIENTED LANGUAGES

There are almost two dozen major object-oriented programming languages in use today. But the leading commercial OO languages are C++, Smalltalk and Java. We will discuss about some of these in unit 4 of this block. Let us discuss some of the reasons of success and advantages of C++ in this section as it has been selected by us as the language to be given in more details in this course. Java will be presented in CS-75 course.

1.6.1 Why C++ succeeded

C++ started as an extension to C language or more precisely we can say C++ started as turning C into an OOPL and it emerged out as Superset of C. But this is just the part of the reason for the success of C++. C++ has solved many other problems faced by C programmers in today's development scenarios. C++ has especially come as a major tool to the person who has made large investments in C.

The second reason is the main reason for the success of C++, in a nutshell, is economics: It still costs to move to OOP, but C++ may cost less.

C++ is aimed at enhancing productivity. The productivity enhancement is due to design, which helps you as much as possible and do not hinder you with any arbitrary rules and requirements. It is designed to follow a practical approach aimed at benefiting the programmer.

1.6.2 Advantages of C++

The basic advantages of C++ can be summed up as under:

- C++ has closed many holes in the C language and provides better type checking and compile-time analysis.
- You are forced to declare functions so that the compiler can check their use. The need for the preprocessor has virtually been eliminated for value substitution and macros, which removes a set of difficult-to-find bugs.
- C++ has a feature called references that allows more convenient handling of addresses for function arguments and returned values.
- The handling of names is improved through a feature called function overloading, which allows you to use the same name for different functions. A feature called namespaces also improves the control of names. There are numerous smaller features that improve the safety of C.
- **The learning curve for C programmers is very fast:** Most of the companies already have C programmers. They do not want that their programmer become ineffective in a day. C++ is an extension of C, thus, reduces the learning time. In addition, C++ compiler accepts C code.
- **Efficiency:** C++ allows greater control of program performance and also allows programmers to interact with assembly code as the case with C language. Thus, C++ is quite a performance-oriented language. It sacrifices some flexibility in order to remain efficient, however, C++ uses compile-time binding, which means that the programmer must specify the specific class of an object, or at the very least, the most general class that an object can belong to. This makes for high run-time efficiency and small code size, but it trades off some of the power to reuse classes.
- **Systems are easier to express and understand:** Since, C++ supports object oriented paradigm, thus, demonstrates the compatibility of good solution expression as it deals with higher-level concept like objects and classes rather than functions and data. It also produces maintainable code. The programs that are easier to understand are easier to maintain.
- **Good Library Support:** One of the fastest ways to create a program is to use already written code from the library. C++ libraries are easy to use and can be used in creating new classes, C++ guarantees proper initialization, clean up and call to library functions/classes, you can use the libraries by just knowing the message interfaces.
- **Source Code Reuse using templates:** Template feature reuses same source code with automatic modification for different classes. It is a very powerful tool that allows reuse of library code. Templates hide complexity of the code reuse for different classes on the user.
- **Error Handling:** C++ supports error-handling capabilities that catches the errors and reports them too. This feature provides control of error handling to the programmers in a similar way as being done for the libraries.
- **Programming in Large:** Many programming languages have their own limitation; some have limitations on line of code, some on recursion etc. however, C++ provides many features that supports the programming. Some of these features are:
 - Templates, namespaces and exception handling,
 - Strongly typed easy to use compiler,

- Small or large programs are allowed,
- Objects help in reducing complex problem to manageable one.

Check Your Progress 2

- 1) State True (T) or False (F)
 - (a) Object Oriented Programming can not be used for client server applications. True False
 - (b) The kernel of Linux Operating System is implemented using C. True False
 - (c) One of the advantages of object oriented database management system is that it reduces developmental costs. True False
 - (d) Object Oriented Programming C++ is slower than C. True False
 - (e) C++ library is not as good as C. True False
 - (f) Error handling cannot be done in C++ True False
- 2) What is the meaning of expression $C=a+b$ in the context of C++, where c , a and b all are complex numbers.
.....
.....
.....
.....

1.7 SUMMARY

Object-oriented programming offers a new and powerful model for writing computer software. The basic backbone being the object, which sends and receive messages, object oriented programming paradigm speeds up the program development and improves maintainability, reusability and modification of a program.

Object oriented programming requires a major shift in thinking by programmers. The C++ language offers an easier transition via C, but it still requires an Object oriented design approach in order to make proper use of this technology.

C++ is an object-oriented version of C. It is compatible with C (it is actually a superset), so that existing C code can be incorporated into C++ programs. C++ programs are fast and efficient, qualities which helped make C an extremely popular programming language. C++ has become so popular that most new C compilers are actually C/C++ compilers. However, to take full advantage of object-oriented programming, one must program (and think!) using objects.

1.8 MODEL ANSWERS

Check Your Progress 1

- 1) The main difference is in the approach. The data of objects can be modified by the functions of that object/class whereas in procedures/ functions there is concept of local data and global data. The basic

advantage of such OOP scheme, thus, is what operations that can be performed on objects are known and one can easily determine which elements/function/object has caused an error in data, if any.

- 2) Through interface messages.
- 3)
 - 1) Reusability
 - 2) Maintainability
 - 3) Modular hierarchical design

Check Your progress 2

- 1)
 - a) False
 - b) False
 - c) True
 - d) True
 - e) False
 - f) False
- 2) You as a programmer have to define a meaning for the message '+' as it is not directly defined in the language for complex number objects. How can you do it? You will learn about how to do such defining things using C++ in block 2.

UNIT 2 OBJECT ORIENTED PROGRAMMING SYSTEM

Structure

- 2.0 Introduction
- 2.1 Objectives
- 2.2 What is OOPS?
- 2.3 Class
- 2.4 Inheritance
- 2.5 Abstraction
- 2.6 Encapsulation & Information Hiding
- 2.7 Polymorphism
- 2.8 Summary
- 2.9 Model Answers

2.0 INTRODUCTION

Object Oriented Programming over the last decade has become a major trend in developing software and is accepted in both industry as well as research labs and academia. Object Oriented Programming System (OOPS) has come a long way and has seen many languages implementing it as a way of developing software. OOPS is implemented by languages in many flavours, some are purely object oriented (for example SMALLTALK) and some are a combination of traditional procedure based and Object-Oriented programming. OOPS have several advantages over earlier programming paradigms. In this unit, we will present a general description of the basic concepts of object-oriented programming.

Object oriented technologies can either confuse you or make you successful. It depends on your approach of using them and your understanding of the ultimate goal of object-oriented (OO) languages.

2.1 OBJECTIVES

After going through this Unit you will be able to:

- Describe the concepts of Object Oriented Programming;
- Define various terms used in Object Oriented Programming; and
- Describe the terminology like abstraction, encapsulation, inheritance, polymorphism.

2.2 WHAT IS OOPS?

Object Oriented Programming Systems (OOPS) is a way of developing software-using Objects. As described in the previous unit, Objects are the real world models, which are entities in themselves. That is they contain their own data and behaviour. An object resembles the physical world. When something

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

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

Physical Objects

ATM in Automated teller machines

Aircraft in an Air traffic control system

Countries in the political model

Elements of the Computer User Environment

Windows

Menus

Graphic Objects (lines, rectangles, circles)

The mouse, keyboard, disk drives, printer

Data Storage constructs

Arrays

Stacks

Linked Lists

Binary Trees

Human Entities

Employees

Students

Customers

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

One possible solution for employee type may be:

Object: Employee

Data: Name, DOB, Gender, Basic Salary, HRA, Designation, Department, Contact address, qualification, any other details.

Operations: Find_Age, compute_Salary, Find_address.

Create_new_employee_object, delete_an_old_employee_object.

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

The Objects are defined via the classes.

2.3 CLASS

Objects with similar properties are put together in a class. A class is a pattern, template, or blueprint for a category of structurally identical items (objects). OOPS programmers view Objects as instances of Class. A class is a blueprint from which objects can be created/instantiated.

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

Class in OOPS contains its members and controls outside access i.e. it provides interface for external access. The class acts as a guard and, thus, provides information hiding and encapsulation. These concepts are discussed later in the unit.

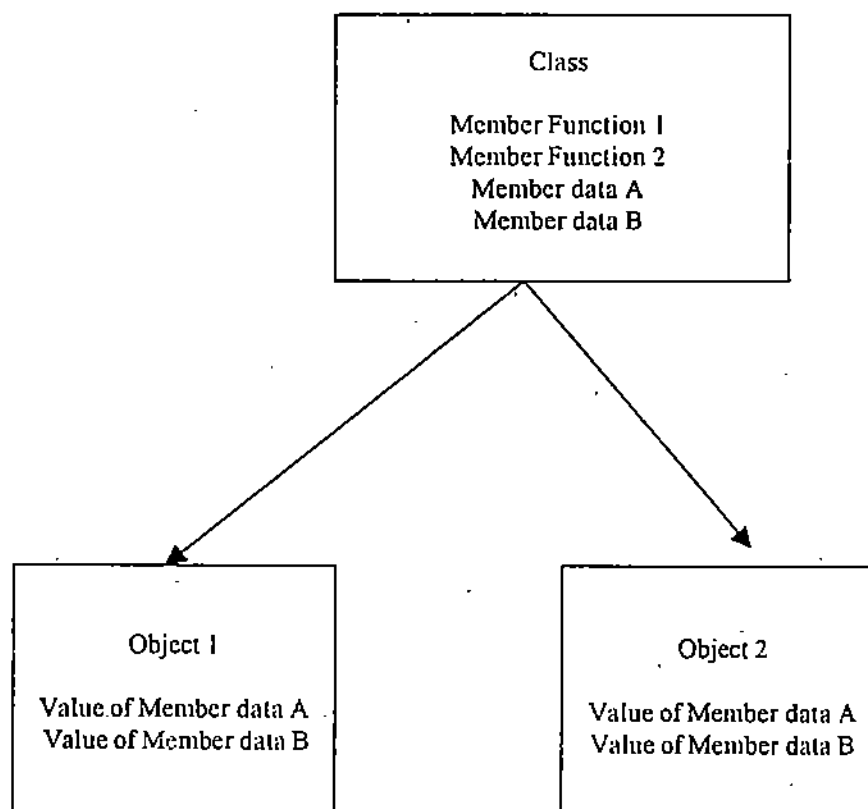


Figure 1: Class and Objects

All the objects share same member data functions but maintain separate copy of member data. (Please refer figure 1). You can use class for defining a user defined data type. A class serves as a plan, or a template that specifies what

data and what functions will be included in objects of that class. Defining the class does not create any objects, just as the mere existence of a type int does not create variables of type int.

A class is a description of a number of similar objects. A class has meaning only when it is instantiated. For example, we can use a class employee directly. We define a class employee and instantiate it.

Class Employee;

Employee John;

Now we can have various operations on John like compute_salary of John.

Check Your Progress 1

1) Which of the following cannot be put under the category of an object.

- Employers
- Manager
- Doubly linked list
- Quick sorting of numbers
- Square root of a number
- Students
- Word file

2) State True or False

- a) Two objects of same class share same data values. True False
- b) A class is an obstruction of an object. True False
- c) An instantiation of a class is an object. True False
- d) Objects are associated with one or more classes True False

2.4 INHERITANCE

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

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

The concept of inheritance allows the features of base class to be accessed by the derived classes, which in turn have their new features in addition to the old base class features. The original base class is also called the parent or super class and the derived class is also called as sub-class.

An example

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

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

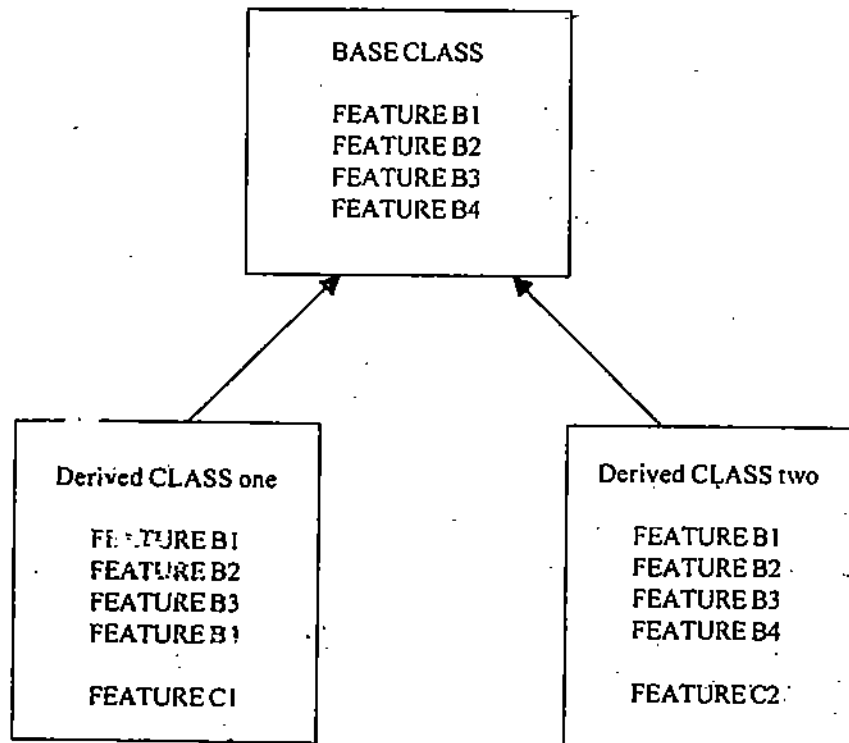


Figure 2: Inheritance

In the figure 2, Classes one and two are derived from base class. Note that both derived classes have their own features C1 and C2 in addition to derived features B1, B2, B3, B4 from base class.

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

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

Inheritance is of two types

Single Inheritance

When the derivation of a derived class is from one base class it is called single inheritance.

Multiple Inheritance

When the derivation of a derived class is from more than one base classes then it is multiple inheritance. The concept of inheritance is same in both type of inheritance, the only difference being in number of base classes.

Advantages of Inheritance

Reuse of existing code and program functionality: The programmer does not have to write and re-write the same code for logically same problems. They can derive the existing features from the existing classes and add the required characteristics to the new derived classes.

Much of the art of Object Oriented programming involves determining the best way to divide a program into an economical set of classes. In addition to speeding development time, proper class construction and reuse results in far fewer lines of code, which translates to less bugs and lower maintenance costs.

Less labour intensive: The programmers do not have to rewrite same long similar programs just because the application to be developed has slightly different requirements.

Well organized: The objects are well organised in a way that they follow some hierarchy.

An example of inheritance could be taken from bank. A bank maintains several kind of bank accounts e.g. Savings Account, Current Account, Loan Account etc., all these accounts at bank have certain common features like customer name, account number etc., at the same time, every type of account has its own characteristics e.g., loan account could have guarantors name, savings account could contain introducers name etc. All these accounts can be derived from the base class Bank Account and have separate derived classes for each of them.

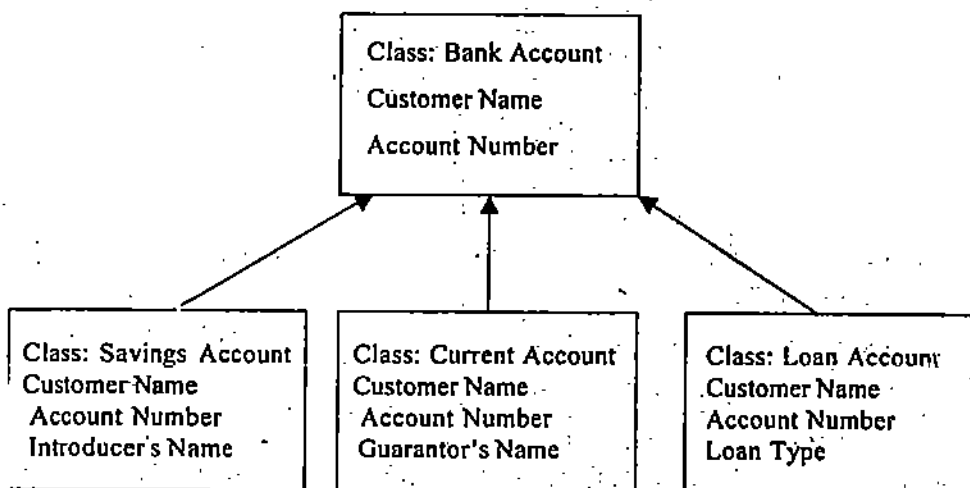


Figure 3: Example of Inheritance

2.5 ABSTRACTION

Whenever we have to solve a problem then first we try to distinguish between the important and unimportant aspects of the problem. This is abstraction, thus. Abstraction identifies patterns and frameworks, and separate important and non-important problem spaces. There could be many levels of Abstractions, like,

- Most important details,
- Less important details, and
- Unimportant details.

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

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

A programming language should facilitate the process of program invention and design by letting you encode abstractions to reveal the way things work. If the language inherently does not support Abstraction then it is task of the programmer to implement Abstraction. All programming languages provides a way to express abstractions. In essence, Abstraction is a way of grouping implementation details, hiding them, and giving them, at least to some extent, a common interface.

Abstraction in a procedural language

The principal units of abstraction in the C language are structures and functions. Both, in different ways, hide elements of the implementation. For example, C structures group data elements into larger units, which can then be handled as a single entity. One structure can include others, so a complex arrangement of constructs can be built from simpler structure constructs. A structure is an example of data abstraction.

Functions in C language encapsulate behaviours that can be used repeatedly. Data elements local to a function are protected within their own domain. Functions can reference (call) other functions, so quite complex behaviours can be built from smaller pieces. Functions in C language represent procedural side of abstraction.

Well-designed functions are reusable. Once defined, they can be called any number of times. The most useful functions can be collected in libraries and reused in different applications. All the user needs is the function interface and not the source code. Each function in C must have a unique name. Although the function may be reusable, its name is not.

C structures and functions are able to express Abstractions to certain extent, however, they maintain the distinction between data and operations on data.

Abstraction in an Object-Oriented language

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

With these changes, the structure has become an opaque token that other programmers never need to look inside. They can concentrate on what the functions do, not how the data is organized. You have taken the first step toward creating an object.

The next step is to provide support to Abstraction in the programming language and completely hide the data structure. In such implementations, the data becomes an internal implementation detail; and user only sees the functional interface. Because an object completely encapsulates their data (hide it), users can think of them solely in terms of their behaviour.

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

This progression from thinking about functions and data structures to thinking about object behaviour is the essence of object-oriented programming. It may seem unfamiliar at first, but as you gain experience with object-oriented

programming, you will find that it is a more natural way to think about things. By providing higher level of Abstraction, object-oriented programming languages give us a larger vocabulary and a richer model to program in.

Mechanisms of Abstraction

Thus, Abstraction is when we create an object we concentrate only on its external working while discarding unnecessary details. The internal details are hidden inside the object, which makes an object abstract. This technique of hiding details is referred to as data abstraction.

Objects in an object-oriented language have been introduced as units that implement higher-level abstractions and work as coherent role-players within an application. However, they could not be used this way without the support of various language mechanisms. Two of the most important mechanisms are: Encapsulation, and Polymorphism.

Check Your Progress 2

1) What is the need of inheritance?

.....

2) What is abstraction in the context of object oriented programming?

.....

3) State True (T) or False (F)

- a) Abstraction can be implemented using structures. True False
- b) Multiple Inheritance means that one base class have multiple sub-classes. True False
- c) Inheritance is useful only when it is single inheritance. True False
- d) Object oriented languages provide higher level of abstraction, that is, they are closer to real world objects. True False

2.6 ENCAPSULATION AND INFORMATION HIDING

To design effectively at any level of abstraction, you should not be involved too much in thinking about details of implementation rather you should be thinking in terms of units for grouping those details under a common interface.

For a programming unit to be truly effective, the barrier between interface and implementation must be absolute. The interface must encapsulate the implementation; hide it from other parts of the program. Encapsulation protects an implementation from unintended actions and inadvertent access.

In programming, the process of combining elements to create a new entity is encapsulation. For example, a procedure is a type of encapsulation because it combines a series of computer instructions. Its implementation is inaccessible

to other parts of the program and protected from whatever actions might be taken outside the body of the procedure. Likewise, a complex data type, such as a record or class, relies on encapsulation. Object-oriented programming languages rely heavily on encapsulation to create high-level objects.

In an Object oriented language, a class is clearly encapsulated as the data variables and the related operations on data are placed together in a class.

For example, in a windows based software, the window object contains Window's dimensions, position, colour etc. Encapsulated with these data are the functions which can be performed on Window i.e. moving, resizing of Window etc. The other part of this window program will call upon window object to carry out the necessary function. The calling or interacting with the window object will be performed by sending messages to it. The required action will be performed by the window object according to its internal structure. This internal working is hidden from the external world or from the other part of the software program.

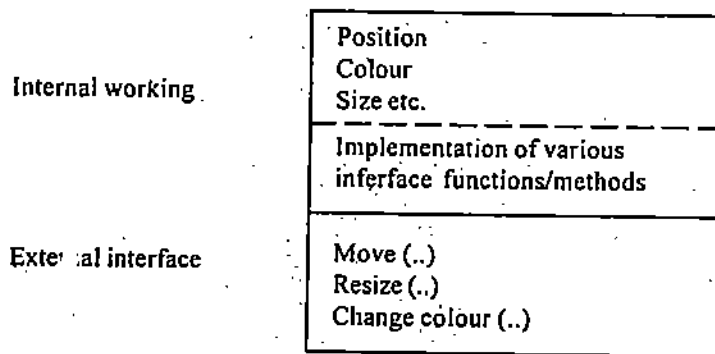


Figure 4 : Window Object

Based on the requirement and choice there are three types of access modes of the members of class:

- The data variables of the class A can only be accessed by the functions of the class A - this is private mode of access.
- The data variables of the class A can be accessed by any functions - this is public mode of access.
- The data variables of the class A can only be accessed by the functions with some special privileges - this is protected mode of access.

Thus, an object's variables are hidden inside the object and invisible outside it. The encapsulation of these instance variables is sometimes also called information hiding.

The process of hiding details of an object or function is information hiding. Information hiding is a powerful programming technique because it reduces complexity. One of the chief mechanisms for hiding information is encapsulation — combining elements to create a larger entity. The programmer can then focus on the new object without worrying about the hidden details. In a sense, the entire hierarchy of programming languages — from machine languages to high-level languages — can be seen as a form of information hiding.

Information hiding is also used to prevent programmers from changing intentionally or unintentionally — parts of a program.

It might seem, at first, that hiding the information in instance variables would constrain your freedom as a programmer. Actually, it gives you more room to act and free you from constraints that might otherwise be imposed. If any part of an object's implementation could leak out and become accessible or a concern to other parts of the program, it would tie the hands of both, the person who have implemented the Object and of those who would use the object. Neither could make modifications without first checking with the other.

For example, that you are interested in developing the object say "Pump" for the program that models use of water and you want to incorporate it in another program you are writing. Once the interface to the object is decided, you do not have to be concerned about fixing the bugs, and finding better ways to implement it, without worrying too much about the people using it.

You will decide all the functions and operations of pump and you will be providing a complete functional object i.e., pump for the other program through an interface. The programmer of the another program will solely depend on the interface and will not be able to break your code and change the functionality and implementation of the object pump. As a matter of fact rather s/he will not be even knowing about the implementation details of the object pump. S/he will simply be knowing the functional details of the object pump. Your program is insulated from the object's implementation. This way the information about the object pump will be hidden from all other modules except the one of which it is part.

Moreover, although those implementing or using the object pump would be interested in how you are using the class and might try to make sure that it meets your needs, they do not have to be concerned with the way you are writing your code. Nothing you do can touch the implementation of the object or limit their freedom to make changes in future releases. The implementation is insulated from anything that you or other users of the object might do.

2.7 POLYMORPHISM

The word polymorphism is derived from two Latin words poly (many) and morphs (forms). This concept of OOPS provides one function to be used in many different forms depending on the situation it is used. The polymorphism is used when we have one function to be carried out in several ways or on several object types. The polymorphism is the ability of different objects to respond in their own ways to an identical message.

When a message is sent requesting an object to do a particular function, the message names the function the object should perform. Because different objects can have different functions with the same name, the meaning of a message must be decided with respect to the particular object that receives the message. Thus, same message sent to two different objects can invoke two different methods.

The main advantage of polymorphism is that it simplifies the programming interface. It allows creation of conventions that can be reused from class to class. Instead of inventing a new name for each new function you add to a program, the same names that may be reused. The programming interface can be described as a set of abstract behaviours that may be different from the classes that implement them.

Overloading

The terms "polymorphism" and "argument overloading" refer basically to the same thing, but from slightly different points of view. Polymorphism takes a

pluralistic point of view and notes that several classes can have a method with the same name. Argument overloading takes the point of the view of the function name and notes that it can have different effects depending on what kind of object it applies to.

Operator overloading is similar. It refers to the ability to turn operators of the language (such as '=' and '+' in C) into methods that can be assigned particular meanings for particular kind of objects.

For example, we need to build a program, which will be used for addition. The input to the program should not depend on input variable, that is it should be able to produce the result of addition whether the input is of integer or float or character variable.

The Polymorphism allows the objects to act as black boxes, i.e. they have common external interface which will allow them to be called or manipulated in the same way. But their internal working and the output is different from each other which depends on the way in which they are invoked or manipulated.

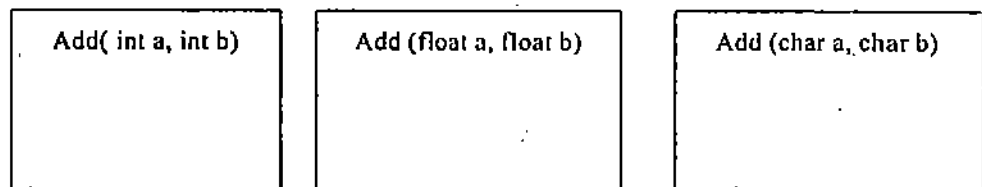


Figure 5: Example of Polymorphism

These addition functions acting as black boxes. All of them does the same thing and bears the same name but perform differently, depending on the arguments passed to them.

Polymorphism also permits code to be isolated in the function of different objects rather than be gathered in a single function that enumerates all the possible cases. This makes the code you write more extensible and reusable. When a new case comes along, you do not have to reimplement existing code, but only add a new class with a new function, leaving the code that is already written alone.

A very common example for the above is using the "draw" function of an object. We might have to draw a circle, or square or triangle etc. But for different drawing methods we will not be creating different methods like draw Circle or draw Square etc. rather we will be defining draw methods with appropriate arguments.

For example,

- Draw (float radius) - Call to this kind of argument will draw circle.....e.g., draw(float 5.2) will draw circle with radius 5.2.
- Draw (float a, float b), will draw square or rectangle.

The same can be extended to sub-classes also where from a base class shape sub-classes such as circle, square and rectangle can be created. Each sub-class will have its own implementation for the base class function draw (). Thus, enabling drawing of circle or square or rectangle based on the sub-class that invokes the message. Thus, polymorphism is a very strong mechanism that support reuse of similarities among object hiding dissimilarities under different behaviour as a result of same message to different objects. The implementation level details about polymorphism is given in block 2.

Check Your Progress 3

1) What is operator overloading? Is it different from polymorphism?

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

2) State True or False

- a) Encapsulation involves data hiding. True False
- b) A window-based environment has variables like position, colour, size that can be modified by any function. This is a valid example of information hiding. True False
- c) Information hiding increase maintenance related problems. True False
- d) Polymorphism can be implemented through any object oriented programming language. True False

2.8 - SUMMARY

In this unit, an overview of various concepts relating to object oriented system has been presented. The base of all the concepts in objects oriented programming is the "object". All the concepts are related to either its properties or behaviour. Class defines the behaviour and the data members of an object. An object encapsulates its data (generally) and generally external interface is the only media for communication with the objects. Inheritance and polymorphism are the concepts that have given major advantages to object oriented programming. Thus, the basic concepts discussed in this unit can be considered as the basic strengths of object oriented programming system.

2.9 MODEL ANSWERS

Check Your Progress 1

- 1) Quicksorting of numbers; square root of numbers.
- 2) a) False
- b) False
- c) True
- d) False

Check Your Progress 2

- 1) Inheritance helps in representing
 - 1) Classes in a hierarchy: Well-organized problem solution space.
 - 2) Extending existing classes thus, promotes reuse.
 - 3) Reducing the duplication of efforts.
- 2) Abstraction is to create model of behaviour of a real object and hiding its internal working details.

- 3) a) False
- b) False
- c) False
- d) True

Check Your Progress 3

- 1) Operator overloading is the way of defining the meaning to operator for a class by using methods/functions for operators. For example, the function for operator + can be written that defines concatenation of two string objects. Polymorphism is a generic concept that involves operator and function overloading. It can be used across several classes where same function name handle may be used for different types of object.
- 2) a) True
- b) False
- c) False
- d) True

UNIT 3 ADVANCED CONCEPTS

Structure

- 3.0 Introduction
- 3.1 Objectives
- 3.2 Dynamism
 - 3.2.1 Dynamic Typing
 - 3.2.2 Dynamic Binding
 - 3.2.3 Late Binding
 - 3.2.4 Dynamic Loading
- 3.3 Structuring Programs
- 3.4 Reusability
- 3.5 Organizing Object-Oriented Projects
 - 3.5.1 Large Scale Designing
 - 3.5.2 Separate Interface and Implementation
 - 3.5.3 Modularising
 - 3.5.4 Simple Interface
 - 3.5.5 Dynamic Decisions
 - 3.5.6 Inheritance of Generic Code
 - 3.5.7 Reuse of Tested Code
- 3.6 Summary
- 3.7 Model Answers

3.0 INTRODUCTION

Object Oriented design has become popular in the industry. In the previous two units of this block our focus was on discussion of some of the basic concepts of object-oriented programming. However, in this unit we would like to touch upon some of the advanced concepts that are useful for implementation of object-oriented programming paradigm. The unit include discussions on dynamism, the basis for overloading and polymorphism, the reusability that is the plank for building reusable classes using standards/tools like CORBA, COM/DCOM etc. and software project related issues. However, in this unit our objective is not to touch issues relating to object oriented modeling and design that will be covered in the last unit of the block.

3.1 OBJECTIVES

At the end of this unit you should be able to:

- Define the dynamism and its implications in the context of overloading polymorphism;
- Describe the reusability concepts, that is bad to better class design; and
- Discuss the features of object oriented projects.

3.2 DYNAMISM

Historically, the amount of memory allocation was determined at the compile and link time of the Program. The source code size was fixed and generally the memory allocated to a Program was as per the source code and could not be increased or decreased once memory is allocated. The approach was restrictive in nature. It not only put the restriction of maximum size of the code for various Programming Languages but also the program design and Programming techniques being followed. For example, a function for allocation and de-allocation of memory could not be thought of for such a system. However, development in hardware and software technology and with Functions (like malloc(), new) that dynamically allocate memory as a program runs opened possibilities that did not existed before.

Compile-time and link-time constraints are limiting in nature as they force resource allocation to be decided from the information obtained from the source code, rather than from the information that is obtained from the running Program.

Although dynamic allocation removes one such constraint that is allocating memory at run time to a data object, many others constraints equally as limiting as static memory allocation, remain. For example, the objects that make up an application must be bound to data types at compile time and the boundaries or size of an application are typically fixed at link time. The above constraints necessitates that every part of an application must be created and assembled in a single executable file. New modules and new types cannot be introduced as the program is being executed.

Object-oriented programming systems tries to overcome these constraints and try to make programs as dynamic as possible. The basic idea of dynamism is to move the burden of decision making regarding resources allocation and code properties from compile time and link time to run time. The underlying philosophy is to allow the control of resources indirectly rather than putting constrains on their actions by the demands of the computer language and the requirements of the compiler and linker. In other words, freeing the world of user's from the programming environments. But what is dynamism in the context of objects oriented-design?

There are three kinds of dynamism for object-oriented design. These are:

- Dynamic typing: That is, to determine the class of an object at run time.
- Dynamic binding : That is, the decision object involving a function in response to a call is moved to run time.
- Dynamic loading: That is, adding new components to a program as it is getting executed.

Let us discuss them in more detail in the following sub-sections.

3.2.1 Dynamic Typing

In general, the compilers give an error message if the code assigns a value to a variable of a different type that it cannot accommodate. Some typical warning messages in such cases may be as under:

“incompatible types in an assignment”

“assignment of integer from pointer does not have a cast”.

Type checking at compile time is useful as it tries to catch many expressions related errors, however, there are times when it can interfere with the benefits

Of mechanisms like polymorphism or the mechanisms where the type of an object is not known till run time.

Suppose, for example, that you want to send an object a message to perform the print method. As the case with other data elements, the object is also represented by a variable. In case the class of this variable is to be determined at compile time then it will not be possible to change the decision about what kind of object should be assigned to the variable at run time. Once class of variable is fixed in source code then it automatically fixes the version of print method that is to be invoked in response to the message.

However, if we delay the assignment/discovery of a class/type of a variable, then it provides a very flexible approach of class-to-type binding and results in assignment of any kind of object to that variable. Thus, depending on the class of the receiver object, the print message might invoke different versions of the method and produce very different results at run time.

Dynamic typing is the basis of dynamic binding (Please refer to next subsection). However, dynamic typing does more than that. It allows associations between objects to be determined at run time, rather than fixing them to be encoded in static compile time design. For example, a message could pass an object as an argument without declaring exactly the type/class of that object. The message receiver may then send a message to the object, again without ascertaining the class of the object. Because the receiver uses the object to do some of its own work, which is in a sense customized by the object of indeterminate type (indeterminate in source code, but, not at run time).

3.2.2 Dynamic Binding

In standard C program, one can declare a set of alternative functions, for example, the standard string-comparison functions,

```
int strcmp(const char *, const char *); /* case sensitive */
int strcasecmp(const char *, const char *); /* case insensitive*/
```

Let us declare a pointer to function `string_compare` that has the same return and argument types:

```
int (* string_compare)(const char *, const char *);
```

In the above case, you can determine the function assignment to pointer at run-time using the following code (through command line arguments)

```
if ( **argv == 'i' )
    string_compare = strcasecmp;
else
    string_compare = strcmp;
```

Thus, you can call the function through the pointer:

```
if (string_compare(s1, s2) )
```

This is a static time binding, where the procedures will be bound at compile time but for a string, which function to follow will be determined at program run time.

This is quite close to what in object-oriented programming is called dynamic binding. Dynamic binding means that delaying the decision of exactly which method to perform until the program is running.

Dynamic binding is not supported by all objects oriented languages. It can easily be accomplished through messaging. You do not need to go through the indirection of declaring a pointer and assigning values to it as shown in the example above. You also need not assign each alternative procedure a different name.

Messages are used for invoking methods indirectly. Every message must match a method implementation. To find the matching method, the messaging system must check the class of the receiver and locate the implementation of the method requested in the message. When such binding is done at run time, the method is dynamically bound to the message. When it is done by the compiler then the method is statically bound.

Dynamic binding is possible even if of dynamic typing does not exist in a programming language but it is not very useful. For example, the benefits of waiting until run time to match a method to a message when the class of the receiver is already fixed and known to the compiler are very little. The compiler could just as well find the method itself; such results will not be different from run-time binding results.

However, in case of the type/class of the receiver is dynamic the compiler cannot determine which method to invoke. The method can be bound only after the class of the receiver is bounded at run time. Thus, Dynamic typing, entails dynamic binding.

Dynamic typing opens the possibility that a message might have very different results depending on the class of the receiver because the run-time data may influence the outcome of a message.

Dynamic typing and binding opened the possibility of sending messages to objects that have not yet been designed. If object types need not be decided until run time, you can give more freedom to designers to design their own classes and name their own data types, and still your code may send messages to their objects. All you need to decide jointly is the message, that is, the interfaces of the objects and not the data types.

3.2.3 Late Binding

Some object-oriented programming languages (such as C++) require a message receiver to be statically typed in source code, but do not require the type to be exact as per the following rule:

"An object can be typed to its own class or to any class that it inherits from". The compiler, therefore, cannot differentiate whether a message receiver is an instance of the class specified in the type declaration OR an instance of a subclass, OR an instance of any further derived class. Since, the sender of the message does not know the class of the receiver, it does not know which version of the method named in the message will be invoked.

As a message is received by a receiver, it can either receive it as an instance of the specified class and the class can simply bind the method defined for that class to the message. As an alternative, the binding may be delayed to run-time. In C++ the binding decisions are delayed to run time for the methods, also called member functions, which are in the same inheritance hierarchy.

This is referred to as "late binding" rather than "dynamic binding". It is "dynamic" in the sense that it happens at run time, however, it carries with it strict compile-time type constraints, whereas, "dynamic binding" as discussed earlier is unconstrained.

3.2.4 Dynamic Loading

From the von Neumann architecture days, the general rule for running a program is to link all its parts together in one file the entire program is loaded into memory that may be virtual memory, prior to execution.

Some object-oriented programming environments have overcome this constraint. They allow different parts of an executable program to be kept in separate files. The program can be created from the bits and pieces as and when they are needed. The component of the programs are dynamically loaded and linked with the rest of program at run-time. The various facilities executed by a user determine the parts of the program that are to be kept in memory for execution purposes.

This is a very useful concept as, in general, only a small core of a large program is used by the users. Thus, only standard core program may be loaded in the memory for execution. Other modules may be called as per the requested of the user. Thus, the component/sections of program that are not executed at all are also not loaded in the memory.

Dynamic loading raises some interesting possibilities. For example, it encourages modular development with an added flexibility that the entire program may not to be developed before a program can be used. The programs can be delivered in pieces and you can update one part of it at any time. You can also create program that groups many different tools under a single interface, and load just the tools desired by the user. In addition, alternative sets of tools may also be offered for the given task, the user can select any one tool from the available sets. This will cause only the desired tool set to be loaded.

As per present one of the important benefits of dynamic-loading is that it makes applications extensible. A program designed by you can be added or customized as per your needs. Such examples are common when we look into operating systems like Linux and it utilities. Such program must provide some basic framework for extension: at run time these program find the pieces that have been implemented and load them dynamically.

One of the key requirements of dynamic loading is to make a newly loaded part of a program to work with parts already running, especially when the different parts of the program are written by different people. However, such a problem does not exist in an object-oriented environment because code is organized into logical modules with a clear distinction between implementation and interface. When classes are dynamically loaded, the newly loaded code can not clash with the already loaded code. Each class encapsulates its implementation and has an independent name space.

In addition, dynamic typing and dynamic binding concepts allows classes designed by others to fit effortlessly into your program design. Your code can send messages to other's objects and vice-versa neither of you has to know what classes the others have implemented. You only need to agree on the communication protocol

Loading and Linking

Dynamic loading also involves dynamic Linking of programs that requires various parts to be joined so that they can work together. The program is loaded into volatile memory at run time. Linking usually precedes loading. Dynamic loading involves the process of separately loading new or additional parts of the program and linking them dynamically to the parts of the program already running.

3.3 STRUCTURING PROGRAMS

An Object-oriented program has two basic kinds of structure:

- The first Structure is in the inheritance hierarchy of class definitions.
- The other structure is the pattern of message passing as the program runs. These messages create a network of object connections.

The inheritance hierarchy determines how objects are related by type. For example, in the program that models workers in a organization, it might turn out that Managers and Employees are of the same kind of object, except that manager controls a group of employees. Their similarities can be captured in the program design if the manager and employee classes inherit from a common class: Human Resources.

The network of object connections explains the working of the program. For example, Manager objects might send message requesting employee to do a piece of work. Employee object might communicate with the tools objects, etc. To communicate with each other in this way, objects must know about each others existence. These connections define a program structure.

Thus, the Object-oriented programs are designed by laying down the network of objects with their behaviours and basic patterns of interaction, and finally by arranging the hierarchy of classes. The object-oriented programming require structuring both in the activities of the program and its definition in terms of inheritance hierarchy of classes.

3.4 REUSABILITY

One of the major goal of object-oriented programming is to make reusable programs to the extent possible such that it can be used in many different situations and applications without re-implementation, even though it may be in slightly different form, from the earlier use.

Reusability of a program is influenced by a number of factors, such as:

- Reliability of code whether it is bug free or not
- Clarity of documentation
- Simplicity of programming interface
- Efficiency of code
- The richness of feature set of an object to cater for many different situations.

These factors can also be used to judge the reusability of any program irrespective of the language of implementation as well as class definitions. For example, efficient and well documented programs/functions would be better from the reusability point of view in comparison to the programs that are undocumented and unreliable.

The class definitions lend themselves to reusable code in a much better form than that of procedural functions. Functions can be made more reusable by passing data as arguments rather than using global variables. However, the reusability of functions is still constrained as per the following reasons:

- Function names are global variables by themselves. Each function must have a unique name. This makes it difficult to rely heavily on library code

when building a complex system. It makes the programming very extensive and, thus, hard to learn and difficult to generalize. On the other side, classes can share programming interfaces. With the same naming conventions used over and over again, a great deal of functionality can be packaged with a relatively small and easy-to-understand interface because of various concepts like inheritance, overloading and polymorphism.

- The second problem with functions are that they are selected from a library one at a time. The programmer needs to pick and choose the individual functions as per his needs. In contrast, objects are the packages of functionality and not just individual methods and instance values. They provide integrated services. Thus, an object-oriented library does not have functions that are to be joined by user for a solution. They have objects, which represent a solution to a problem.
- Functions are typically coupled to particular kinds of data structures for a specific program. The interaction between the data and function is major part of the interface. A library function is useful only to those who are using the same kind of data structures. However, an object hides the data, therefore, does not face such a problem. This is one of the main reasons why classes can be reused more easily than functions.

The data of an object is protected by access rights and cannot be altered by any other part of the program. Methods of a class are therefore responsible for integrity of data of an object. The external access cannot put an illogical or untenable state to data of an object. This makes an object data structure more reliable than that of the data passed to a function. Therefore, methods can rely on such reliable data and hence the reusable methods are easier to write.

Moreover, because the data of an object is hidden from an external user, a class can be redesigned to use a better data structure without affecting its interface. All programs that use the changed class can use the new version without changing any source code; no reprogramming is required.

3.5 ORGANIZING OBJECT-ORIENTED PROJECTS

Object-oriented programming allows restructuring of the program design in ways that benefit collaboration. It helps in eliminating the need of collaboration at low-level implementation details, instead it provides the structures that facilitate collaboration at a higher level. The features of the object model, such as code reusability, complexity controls etc. have provided the way, people work together.

3.5.1 Large Scale Designing

A program designed at high level of abstraction allows easier division of labor on logical lines; a project organization grows out of this design.

With an object-oriented design, it is easier to focus on common goals, instead of losing them during the implementation. It is also easier for everyone to visualize, how the module they are working on, fits into the whole program. Their collaborative efforts are, therefore, likely to be organized and oriented towards problem solution.

3.5.2 Separate Interface and Implementation

The inter connections among various components of an object-oriented program are normally worked out early in the design process. The interactions must be well defined prior to implementation.

During implementation phase only the interface of object needs to be monitored for. Since each class encapsulates its implementation and has its own name space, the object oriented projects need not be coordinated for implementation.

3.5.3 Modularizing

Object oriented system support modularity, which implies that a software can be broken into its logical components. Each of these logical components can be implemented separately. Thus, software engineers may be asked to work on different class or module separately.

The benefit of modularization is not only in implementation but also at the maintenance time. The class boundaries contain the problems that are related to a class. Thus, any bug can be tracked to a class and can be rectified.

An interesting outcome with respect to separating responsibilities by class is that each part can be worked on by specialist object. Classes can be updated periodically for performance optimization and as per the new technologies. Such updates need not be coordinated with other parts of the object oriented program. The improvements to a class implementation can be made at any time if its interface remains unchanged.

3.5.4 Simple Interface

The mechanisms such as polymorphism in object-oriented programming yields simpler programming interfaces, since it allows same names and conventions to be reused in any number of different classes. Thus, object oriented classes are easy to learn, and provide a greater understanding of the working of complete system, and an easier cooperation and collaboration mechanism for program development.

3.5.5 Dynamic Decisions

Since many object-oriented programming languages allow decisions dynamically viz., at run time, less information needs to be available at compile time (in source code) for allowing coordination between two objects/classes. Thus, there is less to coordinate and less to go wrong.

3.5.6 Inheritance of Generic Code

Inheritance in a way allows code reuse. It is advisable that in a project you define your classes as specialization of more generic classes. This simplifies programming code. It also simplifies the design as inheritance hierarchy describes the relationships among the different levels of classes and makes them easier to understand.

Inheritance also helps in the reliability of code. For example, the code in a super class is tested by all its subclasses, thus, is tested thoroughly and hence may be most reliable. Similarly, the generic classes of the library are tested by other subclasses written by other developers for other applications.

3.5.7 Reuse of Tested Code

You must reuse the code as much as possible. This reduces the workload and you need not start a project from scratch. There is more code available for reuse; classes themselves are designed keeping reusability in mind. This enhances the collaboration among the programmers working in different places or different organizations.

Classes and frameworks available in an object-oriented library may make substantial contribution to your software. This allows you to concentrate on

what you do the best and leave other tasks to the library creators. This also results in faster project completion with less effort.

An interesting facet in OOPS is that the increased reusability of object-oriented codes also increases its reliability. This is due to reuse of class in different situation; because the reused classes get tested in different situations and applications. The bugs of such classes must have already been diagnosed and fixed.

Check Your Progress

- 1) What is dynamic typing? If a language does not support dynamic typing then is it advisable to have dynamic binding?

.....

.....

.....

- 2) State True or False

- a) Late binding is same as dynamic binding True False
- b) Late binding is the basis of run-time polymorphism in C++
True False
- c) Dynamic Loading does not involve dynamic linking. True False
- d) Object are reusable as they support polymorphism. True False
- e) Object-oriented programming supports better reliability also.
True False

3.6 SUMMARY

In this unit our attempt was to cover some of the important object-oriented features in some more details. The features which have been covered include dynamic typing, dynamic binding and late binding. These concepts are very important from the viewpoint of providing flexibility in program design and development. The other concepts which has formed the basis of popularity of object oriented programming paradigm, are its reusability, reliability and modifiability. The unit discusses why object oriented programs are more reusable, reliable and modifiable/maintainable.

3.7 MODEL ANSWERS

Check Your Progress

- 1) Dynamic typing is the determination of type of an object at run-time. It is not useful to have dynamic binding without dynamic typing as if types are bound then function indirectly get bound to a type/class to which they will be bound at run-time (dynamic binding).
- 2) a) False
b) True
c) False
d) False
e) True

UNIT 4 INTRODUCTION TO OBJECT ORIENTED LANGUAGES

Structure

- 4.0 Introduction
- 4.1 Objectives
- 4.2 Objective-C
- 4.3 Python
- 4.4 C# (C Sharp)
- 4.5 Eiffel
- 4.6 Modula-3
- 4.7 SmallTalk
- 4.8 Object REXX
- 4.9 JAVA
- 4.10 BETA
- 4.11 Various Object Oriented Programming Languages Comparative Chart
- 4.12 Summary
- 4.13 Model Answers

4.0 INTRODUCTION

Object-oriented programming offers a new and powerful model for writing computer software. Object orientation (OO), or to be more precise, object-oriented programming, is a problem-solving method in which the software solution reflects objects in the real world. Objects are "black boxes" which send and receive messages. This approach speeds the development of new programs, and, if properly used, improves the maintenance, reusability, and modifiability of software.

Object Oriented programming requires a major shift in thinking by programmers, however. The C++ language offers an easier transition via C, but it still requires an object oriented design approach in order to make proper use of this technology. Smalltalk offers a pure object oriented environment, with more rapid development time and greater flexibility and power. Java promises much for Web-enabling object oriented programs.

There are almost two dozen major object-oriented programming languages in use today. But the leading commercial object oriented languages are far fewer in number. In this unit, let us look into the features of the mostly used object oriented programming languages. Some of the details provided in this unit are implementation oriented, therefore, it is advisable that you may read the unit again after having practical experiences.

4.1 OBJECTIVES

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

- Describe the importance of Object Oriented Programming Languages, and
- Discuss the features of various Object Oriented Programming Languages.

C is a low-level, block-structured language often used for systems programming. C++ is an Object-Oriented (OO) language developed as an extension of C and used for application development. Objective C is an Object Oriented programming language based on C and used for application and library programming. The Objective-C language is fully compatible with ANSI standard C and provides classes and message passing similar to Smalltalk. Objective-C was invented by Brad Cox who wrote the book "Object Oriented Programming: An Evolutionary Approach" in which he describes the language.

The compiler recognizes Objective-C source files by a ".m" extension, just as it recognizes files containing only standard C syntax by a ".c" extension. The most common set of libraries for Objective-C programming are the Next libraries like Foundation and Appkit.

Features of Objective-C

Objective-C includes, when compared to C, a few more keywords and constructs. Objective-C is a powerful, easy-to-learn, object-oriented extension to C. Unlike in C++, advanced object-oriented features like dynamic binding, run-time type identification, and persistence are standard features in Objective-C which apply universally and work well together. Moreover, the support for descriptive message names (as in SmallTalk) makes Objective-C code easy to read and understand. The GNU and NeXTSTEP C compilers support objective-C.

Syntax

@interface declares a new class. It indicates the name of the class, the name of its superclass, the protocols adhered to, the layout of the instance variables (similar to the definition of a struct, but including encapsulation information and declares the methods implemented by this class. A class' interface usually resides in a file called 'classname.h'.

@implementation defines a class. The implementation is no more than a collection of method definitions. Without an implementation, a class does not exist at run time. The implementation of a class usually resides in a file called 'classname.m'.

@category is a named collection of method definitions, which are added to an existing class. A category is not allowed to redefine a class existing methods.

Objective-C includes the predefined type 'id' which stands for a pointer to some object. Thus, 'id obj;' declares a pointer to an object. The actual class of the object being pointed to is almost irrelevant, since Objective-C does run-time type checking.

-Message declares a method called 'message'. The '-' indicates that the message can be sent to objects. A '+' instead indicates the message can be sent to class objects. A method is similar to a function in that it has arguments and a return value. The default return type is 'id'. If a method has nothing useful to return, it returns 'self', which is a pointer to the object to which the message was sent (similar to 'this' in C++).

Dynamic vs. Static C++ follows the Simula 67 orientation of OO programming, where Objective-C follows the Smalltalk school.

Objective-C have classes similar to Smalltalk. Objective-C is as close to Smalltalk as a compiled language allows. The following is a list of the features:

Compiling

- Objective-C is compiled ---Smalltalk is only partially compiled. The current Objective-C implementations are all much faster than any Smalltalk.
- Objective-C does hybrid typing : one can choose to represent a string as a 'char *' or as an object, whereas in Smalltalk, everything is an object. This is a reason for Objective-C being faster. On the other hand, if every bit of information in an Objective-C program would be represented by an object, the program would probably run at a speed comparable to Smalltalk and it would suffer from not having optimizations performed on the basic classes, like Smalltalk can do.

Messages

- You may add or delete methods and classes at runtime.
- Much of the syntax, i.e. Smalltalk uses method names like 'a:method:name:', as does Objective-C. In Objective-C, the message sending construct is enclosed in square brackets, like this: '[anObject aMessage: arg]' whereas Smalltalk uses something like 'anObject aMessage: arg'.
- The basic class hierarchy, that is, having class 'Object' in the very top, and letting most other classes inherit from it.

Forwarding

- Smalltalk normally uses 'doesNotUnderstand:' to implement forwarding, delegation, proxies etc. In Objective-C, these tasks are different:
- Forwarding/delegation: 'forward::' can be overridden to implement forwarding. On the NeXT, 'forward::' is even used when passing to super.
- Proxies: (Next) An instance of the NXProxy class forwards all methods and their arguments to the remote object via Mach messages.

Classes

- Objective-C has meta classes mostly like Smalltalk.
- Objective-C does not have class variables like Smalltalk, but pool variables and globals are easily emulated via static variables.

Other Features

- The possibility to load class definitions and method definitions (which extend a class) at run time.
- Objects are dynamically typed: Full type information (name and type information of methods and instance variables and type information of method arguments) is available at run time. A prime example of application of this feature is '-loadNibSection:owner:' method of NEXTSTEP's Application class.
- Persistence
- Remote objects
- Delegation and target/action protocols
- There is no innate multiple inheritance (of course some see this as a benefit).
- No class variables

4.3 PYTHON

Python is a language that has always aimed at consistency, simplicity, ease of understanding, and portability. It is simple (yet robust), object-oriented (yet can be used as a procedural language), extensible, scalable and features an easy to learn syntax that is clear and concise. Python combines the power of a compiled object language like Java and C++ with the ease of use. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and Rapid Application Development (RAD) in many areas on most platforms. By the way, the language is named after the BBC show "Monty Python's Flying Circus" and has nothing to do with reptiles. Making references to Monty Python skits in documentation is not only allowed, it is encouraged.

The Python interpreter and the extensive standard library are freely available in source or binary form for all major platforms from the Python Website, <http://www.python.org/> and can be freely distributed. The same site also contains distributions of pointers to many free third party Python modules, programs and tools, and additional documentation.

Features of Python

- The Python interpreter is easily extended with new functions and data types implemented in C or C++ (or other languages callable from C). Python is also suitable as an extension language for customizable applications.
- It offers much more error checking than C, and, being a very-high-level language, it has high-level data types built in, such as flexible arrays and dictionaries that would cost you days to implement efficiently in C. Because of its more general data types Python is applicable to a much larger problem domain than Awk or even Perl, yet many things are at least as easy in Python as in those languages.
- Python allows you to split up your program in modules that can be reused in other Python programs. It comes with a large collection of standard modules that you can use as the basis of your programs—or as examples to start learning to program in Python. There are also built-in modules that provide things like file I/O, system calls, sockets, and even interfaces to graphical user interface toolkits like Tk.
- Python is an interpreted language, which can save you considerable time during program development because no compilation and linking is necessary. The interpreter can be used interactively, which makes it easy to experiment with features of the language, to write throw-away programs, or to test functions during bottom-up program development.
- Python allows writing very compact and readable programs. Programs written in Python are typically much shorter than equivalent C or C++ programs, for several reasons:
 - the high-level data types allow you to express complex operations in a single statement;
 - statement grouping is done by indentation instead of begin/end brackets;
 - no variable or argument declarations are necessary.
- Python is extensible: If you know how to program in C, it is easy to add a new built-in function or module to the interpreter, either to perform critical

operations at maximum speed, or to link Python programs to libraries that may only be available in binary form (such as a vendor-specific graphics library).

- TCP/IP and UDP/IP Network programming using sockets
- Operating system interface
- GUI development with Tk using Tkinter
- Multithreaded programming
- Interactive Web/CGI/Internet applications
- Executing code in a restricted environment
- Inheritance, type emulation, operator overloading, and delegation in an OOP environment.

4.4 C# (C SHARP)

C# (C Sharp) is a modern, object-oriented language that enables programmers to quickly build a wide range of applications for the new Microsoft .NET platform, which provides tools and services that fully exploit both computing and communications. This is the premier language for the Next Generation Windows Services (NGWS) Runtime. This NGWS runtime is a runtime environment that not only manages the execution of code, but also provides services that make programming easier. Compilers produce managed code to target this managed execution environment. You get cross-language integration, cross-language exception handling, enhanced security, versioning and deployment support, and debugging and profiling services for free. C# derived from C and C++, however it is modern, simple, entirely object oriented and type safe.

Because of its elegant object-oriented design, C# is a great choice for architecting a wide range of components-from high-level business objects to system-level applications. Using simple C# language constructs, these components can be converted into XML Web services, allowing them to be invoked across the Internet, from any language running on any operating system.

More than anything else, C# is designed to bring rapid development to the C++ programmer without sacrificing the power and control that have been a hallmark of C and C++. Because of this heritage, C# has a high degree of fidelity with C and C++. Developers familiar with these languages can quickly become productive in C#. As C# is a modern programming language, it simplifies and modernizes C# in the areas of classes, namespaces, method overloading, and exceptional handling. Contributing to the ease of use is the elimination of certain features of C++: no more macros, no templates, and no multiple inheritance and pointers.

Features of C#

- Simple

C# is a simple language. Pointers are missing in C#. In C++ we have ::, .(dot) and -> operators that are used for namespaces, members and references. But C# does away with the different operators in favour of a single one: the .(dot) operator. All that a programmer now has to understand is the notion of nested names. C# provides a unified type system. This type system enables you to view every type as an object, be it a primitive data type or a full-blown class.

- **Modern**

C# is designed to be the premier language for writing NGWS applications. The entire memory management is no longer the duty of the programmer - the runtime of NGWS provides a garbage collector that is responsible for memory management in the C# programs. Exception handling is cross-language (another feature of runtime). It provides you metadata syntax for declaring capabilities and permissions for the underlying NGWS security model.

- **Object-Oriented**

C# supports all the key object-oriented concepts such as encapsulation, inheritance, and polymorphism. The entire C# class model is built on top of the NGWS runtime's Virtual Object System (VOS). Accidental overriding of methods is overcome in C#. C# supports the private, protected, public and internal access modifiers. C# allows only one base class. If the programmer needs the feel for multiple inheritance, he can implement interface.

- **Type-safe**

We cannot use uninitialized variables. For member variables of an object, the compiler takes care of zeroing them. For local variables, the programmer should take care of. C# does away with unsafe casts. Bounds checking is part of C#. Arithmetic operations could overflow the range of the result data type. C# allows you to check for overflow in such operations on either an application level or a statement level. Reference parameters that are passed in C# are type-safe.

- **Versionable**

In C#, the versioning support for applications is provided by the NGWS runtime. C# does its best to support this versioning. Although C# itself cannot guarantee correct versioning, it can ensure that versioning is possible for the programmer.

- **Compatible**

C# allows you to access to different API's with the foremost being the NGWS Common Language Specification (CLS). The CLS defines a standard for interoperability between languages that adhere to this standard. You can also access the older COM objects. C# supports the OLE automation. Finally, C# enables you to interoperate with C-style API's.

- **Flexible**

If you need pointers, you can still use them via unsafe code—and no marshalling is involved when calling the unsafe code.

4.5 EIFFEL

Eiffel is a pure object-oriented language created by Bertrand Meyer and developed by his company, Interactive Software Engineering (ISE) of Goleta, Canada. The language was introduced in 1986. Eiffel is named after Gustave Eiffel, the engineer who designed the Eiffel Tower.

Eiffel encourages object oriented programming development and supports a systematic approach to software development. Eiffel has an elegant design and programming style, and is easy to learn.

The Eiffel compiler generates C code, which you can then modify and re-compile with a C compiler. Its modularity is based on classes. It stresses reliability, and facilitates design by contract. It brings design and programming

closer together. It encourages the re-use of software components. Eiffel offers classes, multiple inheritance, polymorphism, static typing and dynamic binding, genericity (constrained and unconstrained), a disciplined exception mechanism, systematic use of assertions to promote programming by contract, and deferred classes for high-level design and analysis.

It is hard to generalise, but compared to C++, simple computation-intensive applications will run perhaps 15% slower. Large applications are often dominated by memory management rather than computation. ISE recently demonstrated that by simply adding a call to the garbage collector's "full-collect" routine at a time when there were known to be few live objects, performance became dramatically faster than a corresponding C++ version.

There are several significant language features of Eiffel:

- **Portable:** This language is available for major industry platforms, such as Windows, OS/2, Linux, UNIX, VMS, etc..
- **Open System** includes a C and C++ interface making it easily possible to reuse code previously written.
- **"Melting Ice Technology"** combines compilation, for the generation of efficient code, with bytecode interpretation, for fast turnaround after a change.
- **"Design by Contract"** enforced through assertions such as class invariants, preconditions and postconditions.
- **Automatic Documentation ("Short Form"):** Abstract yet precise documentation produced by the environment at the click of a button.
- **Multiple Inheritance:** A class can inherit from as many parents as necessary.
- **Repeated Inheritance:** A class inherits from another through two or more parents.
- **Statically Typed** ensure that errors are caught at compile time, rather than run time.
- **Dynamically Bound** guarantees that the right version of an operation will always be applied depending on the target object.
- **The Few Interfaces Principle** restricts the overall number of communication channels between modules in a software architecture: "Every module should communicate with as few others as possible".

4.6 MODULA-3

Modula-3 programming language is from Digital Equipment Corporation's Systems Research Center (SRC). Modula-3 is a modern, modular, object-oriented language. One of the principal goals for the Modula-3 language was to be simple and comprehensible, yet suitable for building large, robust, long-lived applications and systems. The language design process was one of consolidation and not innovation; that is, the goal was to consolidate ideas from several different languages, ideas that had proven useful for building large sophisticated systems.

The language features garbage collection, exception handling, run-time typing, generics, and support for multithreaded applications. The SRC implementation of this language features a native-code compiler; an incremental, generational, conservative; multithreaded garbage collector (whew!); a minimal

recompilation system; a debugger; a rich set of libraries; support for building distributed applications; a distributed object-oriented scripting language; and finally; a graphical user interface builder for distributed applications.

Features of MODULA-3

- Modula-3 has a particularly simple definition of an object. In Modula-3, an object is a record on the heap with an associated method suite. The data fields of the object define the state and the method suite defines the behaviour. The Modula-3 language allows the state of an object to be hidden in an implementation module with only the behaviour visible in the interface. This is different than C++ where a class definition lists both the member data and member function. The C++ model reveals what is essentially private information (namely the state) to the entire world. With Modula-3 objects, what should be private can be really be private.
- Modula-3 has the feature of garbage collection. Garbage collection really enables robust, long-lived systems. Without garbage collection, you need to define conventions about who owns a piece of storage. In 'C', a 'longjmp' may cause storage to be lost if the procedure being unwound doesn't get a chance to clean up. Exception handling in C++ has the same problems. In general, it is very difficult to manually reclaim storage in the face of failure. Having garbage collection in the language removes all of these problems. Better yet, the garbage collector that is provided with SRC implementation of Modula-3 has excellent performance. It is the result of several years of production use and tuning.
- Modula-3 provides such a standard interface for creating threads. In addition, the language itself includes support for managing locks. The standard libraries provided in the SRC implementation are all thread-safe. Trestle, which is a library providing an interface to X, is not only thread-safe, but itself uses threads to carry out long operations in the background. With a Trestle-based application, you can create a thread to carry out some potentially long-running operation in response to a mouse-button click. This thread runs in the background without tying up the user interface. It is a lot simpler and error prone than trying to accomplish the same thing with signal handlers and timers.
- Generic interfaces and modules are a key to reuse. One of the principal uses is in defining container types such as stacks, lists, and queues. They allow container objects to be independent of the type of entity contained. Thus, one needs to define only a single "Table" interface that is then instantiated to provide the needed kind of "Table"; whether an integer table or a floating-point table or some other type of table is needed. Modula-3 generics are cleaner than C++ parameterized types, but provide much of the same flexibility.
- Modula-3 provides a simple single-inheritance object system.
- Existing non-Modula-3 libraries can be imported. Many existing C libraries make extensive use of machine-dependent operations. These can be imported as "unsafe" interfaces. Then, safer interfaces can be built on top of these while still allowing access to the unsafe features of the libraries for those applications that need them.

4.7 SMALL TALK

Smalltalk is a purely object-oriented language which cleanly supports the notion of classes, methods, messages and inheritance. Smalltalk, a programming

language developed in the 1970s at Xerox's Palo Alto Research Center in California.

Unlike "hybrid" object-oriented languages C++, Smalltalk is considered to be a "pure" object-oriented language. Smalltalk is said to be "pure" for one main reason: Everything in Smalltalk is an object, whereas in hybrid systems there are things, which are not objects (for example, integers in C++ and Java).

The benefits of the "everything is an object" philosophy are great, and pure languages such as Smalltalk are considered to be more productive and (more importantly) more fun to program in.

Smalltalk is fundamentally tied to automatic dynamic memory management, and as such must be supported by an underlying automatic memory management system. In practical terms this means that a Smalltalk programmer no longer needs to worry about when to free allocated memory. When finished with a dynamically allocated object, the program can simply "walk away" from the object. The object is automatically freed, and its storage space recycled, when there is nothing else referencing it.

All Smalltalk code is composed of chains of messages sent to objects. Even the programming environment itself is designed within this metaphor. A large number of predefined classes are collectively responsible for the system's impressive functionality. Different from most other programming tools all of this functionality is always accessible to browsing and change, a fact which makes Smalltalk an extremely flexible system, which is easy to customize according to one's own preferences.

Smalltalk programs are considered by most to be significantly faster to develop than C++ programs. A rich class library that can be easily reused via inheritance is one reason for this. Another reason is Smalltalk's dynamic development environment. It is not explicitly compiled, like C++. This makes the development process more fluid, so that "what if" scenarios can be easily tried out, and classes definitions easily refined. But being purely object-oriented, programmers cannot simply put their toes in the O-O waters, as with C++. For this reason, Smalltalk generally takes longer to master than C++. But most of this time is actually spent learning object-oriented methodology and techniques, rather than details of a particular programming language. In fact, Smalltalk is syntactically very simple, much more so than either C or C++. There are many different versions of Smalltalk. You can think of them as different dialects of the Smalltalk language, much like there are different dialects of a human language.

Open Source & Free Smalltalk Versions

- Squeak Smalltalk
- GNU Smalltalk
- Little Smalltalk

Commercial Smalltalk Versions

- Dolphin Smalltalk
- Object Connect's Smalltalk MT
- Exept's Smalltalk/X
- Cincom's Visual Works Smalltalk
- Cincom's Object Studio Smalltalk
- IBM's Visual Age Smalltalk

- Pocket Smalltalk.
- QKS Smalltalk Agents.

4.8 OBJECT REXX

IBM Object REXX is an object-oriented programming language suited for beginners as well as experienced OO programmers. It is upward compatible with previous versions of classic REXX and provides an easy migration path to the world of objects. Because it can be used with REXX conventional programming, Object REXX protects your investment in existing REXX program code. It provides many programming interfaces to existing applications, such as DB2, C, and C++ applications. Object REXX runs on AIX, Linux, OS/2, Windows 2000, Windows 98, Windows Me and Windows NT.

Features of REXX

- Suitable for solving small automation problems and developing fully realized applications.
- Includes a rich set of system interfaces.
- Can be used for writing powerful command procedures for Windows.
- Includes the complete Object REXX Interpreter.
- Runs immediately without compilation or linkage.
- Can develop and debug Object REXX applications, including GUIs (Development Edition).
- Support for OLE/Active X.
- UNICODE conversion functions.
- Enhanced with full object orientation.
- Designed for object-oriented programming, and also allows REXX conventional programming.
- Provides a REXX API to develop external function libraries written in C.
- Includes applications developed mainly in C or C++.

4.9 JAVA

Java was developed by taking the best points from other programming languages, primarily C and C++. Java therefore utilizes algorithms and methodologies that are already proven. Error prone tasks such as pointers and memory management have either been eliminated or are handled by the Java environment automatically rather than by the programmer. Since Java is primarily a derivative of C++ which most programmers are conversant with, it implies that Java has a familiar feel rendering it easy to use.

Features of Java

- Object oriented

Even though Java has the look and feel of C++, it is a wholly independent language which has been designed to be object-oriented from the ground up. In

object-oriented programming (OOP), data is treated as objects to which methods are applied. Java's basic execution unit is the class. Advantages of OOP include: reusability of code, extensibility and dynamic applications.

- **Distributed**

Commonly used Internet protocols such as HTTP and FTP as well as calls for network access are built into Java. Internet programmers can call on the functions through the supplied libraries and be able to access files on the Internet as easily as writing to a local file system.

- **Interpreted**

When Java code is compiled, the compiler outputs the Java Bytecode which is an executable for the Java Virtual Machine. The Java Virtual Machine does not exist physically but is the specification for a hypothetical processor that can run Java code. The bytecode is then run through a Java interpreter on any given platform that has the interpreter ported to it. The interpreter converts the code to the target hardware and executes it.

- **Robust**

Java compels the programmer to be thorough. It carries out type checking at both compile and runtime making sure that every data structure has been clearly defined and typed. Java manages memory automatically by using an automatic garbage collector. The garbage collector runs as a low priority thread in the background keeping track of all objects and references to those objects in a Java program. When an object has no more references, the garbage collector tags it for removal and removes the object either when there is an immediate need for more memory or when the demand on processor cycles by the program is low.

- **Secure**

The Java language has built-in capabilities to ensure that violations of security do not occur. Consider a Java program running on a workstation on a local area network which in turn is connected to the Internet. Being a dynamic and distributed computing environment, the Java program can, at runtime, dynamically bring in the classes it needs to run either from the workstation's hard drive, other computers on the local area network or a computer thousands of miles away somewhere on the Internet. This ability of classes or applets to come from unknown locations and execute automatically on a local computer sounds like every system administrator's nightmare considering that there could be lurking out there on one of the millions of computers on the Internet, some viruses, trojan horses or worms which can invade the local computer system and wreak havoc on it.

Java goes to great lengths to address these security issues by putting in place a very rigorous multilevel system of security:

- First and foremost, at compile time, pointers and memory allocation are removed thereby eliminating the tools that a system breaker could use to gain access to system resources. Memory allocation is deferred until runtime.
- Even though the Java compiler produces only correct Java code, there is still the possibility of the code being tampered with between compilation and runtime. Java guards against this by using the bytecode verifier to check the bytecode for language compliance when the code first enters the interpreter, before it ever even gets the chance to run.

The bytecode verifier ensures that the code does not do any of the following:

- Forge pointers
- Violate access restrictions
- Incorrectly access classes
- Overflow or underflow operand stack
- Use incorrect parameters of bytecode instructions
- Use illegal data conversions.
- At runtime, the Java interpreter further ensures that classes loaded do not access the file system except in the manner permitted by the client or the user.

Sun Microsystems will soon be adding yet another dimension to the security of Java. They are currently working on a public-key encryption system to allow Java applications to be stored and transmitted over the Internet in a secure encrypted form.

- **Architecturally neutral**

The Java compiler compiles source code to a stage which is intermediate between source and native machine code. This intermediate stage is known as the bytecode, which is neutral. The bytecode conforms to the specification of a hypothetical machine called the Java Virtual Machine and can be efficiently converted into native code for a particular processor.

- **Portable**

By porting an interpreter for the Java Virtual Machine to any computer hardware/operating system, one is assured that all code compiled for it will run on that system. This forms the basis for Java's portability.

Another feature which Java employs in order to guarantee portability is by creating a single standard for data sizes irrespective of processor or operating system platforms.

- **High performance**

The Java language supports many high-performance features such as **multithreading, just-in-time compiling, and native code usage.**

- Java has employed multithreading to help overcome the performance problems suffered by interpreted code as compared to native code. Since an executing program hardly ever uses CPU cycles 100 % of the time, Java uses the idle time to perform the necessary garbage cleanup and general system maintenance that renders traditional interpreters slow in executing applications. [NB: Multithreading is the ability of an application to execute more than one task (thread) at the same time e.g. a word processor can be carrying out spell check in one document and printing a second document at the same time.]
- Since the bytecode produced by the Java compiler from the corresponding source code is very close to machine code, it can be interpreted very efficiently on any platform. In cases where even greater performance is necessary than the interpreter can provide, just-in-time compilation can be employed whereby the code is compiled at run-time to native code before execution.
- An alternative to just-in-time compilation is to link in native C code. This yields even greater performance but is more burdensome on the programmer and reduces the portability of the code.

- **Dynamic**

By connecting to the Internet, a user immediately has access to thousands of programs and other computers. During the execution of a program, Java can dynamically load classes that it requires either from the local hard drive, from another computer on the local area network or from a computer somewhere on the Internet.

4.10 BETA

BETA is a modern object-oriented language with comprehensive facilities for procedural and functional programming. BETA originates from the Scandinavian school of object-orientation where the first object-oriented language Simula was developed. Object-oriented programming originated with the Simula languages developed at the Norwegian Computing Center, Oslo, in the 1960s. The first Simula language, Simula I, was intended for writing simulation programs. Simula I was later used as a basis for defining a general-purpose programming language, Simula 67 (later renamed to Simula). A relatively small community has used Simula for number of years, although it has had a major impact on research in computer science.

The BETA language development process started out in 1975 with the aim to develop concepts, constructs and tools for programming, partly based on the Simula languages. The BETA language team consists of Bent Bruun Kristensen, Birger Møller-Pedersen, Ole Lehrmann Madsen, and Kristen Nygaard. Kristen Nygaard was one of the two original designers of the Simula languages. Currently, BETA is available on UNIX workstations, on PowerPC Macintosh and on Intel-based PCs.

On UNIX, the platforms supported are: Sun Sparc (Solaris), HP 9000 (series 700) and Silicon Graphics MIPS machines running IRIX 5.3 or 6.

The definition of the BETA language is in the public domain. This definition is controlled by the original designers of the BETA language: Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. This means that anyone or any company may create a compiler, interpreter, or whatever having to do with BETA.

BETA has powerful abstraction mechanisms than provide excellent support for design and implementation, including data definition for persistent data. The abstraction mechanisms include support for identification of objects, classification, and composition. BETA is a strongly typed language (like Simula, Eiffel, and C++), with most type checking being carried out at compile-time.

The abstraction mechanisms include class, procedure, function, coroutine, process, exception, and many more, all unified into the ultimate abstraction mechanism: the pattern. In addition to the pattern, BETA has subpattern, virtual pattern, and pattern variable.

BETA does not only allow for passive objects as in Smalltalk, C++, and Eiffel. BETA objects may also act as coroutines, making it possible to model alternating sequential processes and quasi-parallel processes. BETA coroutines may also be executed concurrently with supported facilities for synchronization and communication, including monitors and rendezvous communication.

Features of BETA

BETA replaces classes, procedures, functions, and types by a single abstraction mechanism, called the pattern. It generalizes virtual procedures to virtual patterns, streamlines linguistic notions such as nesting and block

structure, and provides a unified framework for sequential, coroutine, and concurrent execution. The resulting language is smaller than Simula in spite of being considerably more expressive.

The pattern concept is the basic construct. A pattern is a description from which objects may be created. Patterns describe all aspects of objects, such as attributes and operations, as seen in traditional object-oriented languages, but also aspects such as parameters and actions, as seen in procedures.

Objects are created from the patterns. Objects may be traditional objects as found in other languages, but they may also be objects which correspond to procedure or function activations, exception occurrences, or even coroutines or concurrent processes.

Objects may be created statically or dynamically and the objects are automatically garbage collected by the runtime system when no references exist to them any longer.

Patterns may be used as superpatterns to other patterns (the subpatterns). This corresponds to traditional class hierarchies, but since patterns may describe other types of objects, inheritance is a structuring means available also for procedures, functions, exceptions, coroutines, and processes.

Patterns may be virtual. This corresponds to traditional virtual procedures but again the generality of the pattern construct implies that also classes, exceptions, coroutines, and processes may be virtual.

Virtual patterns in the form of classes are similar to generic templates in other languages. The prime difference is that the generic parameters (that is, the virtual class patterns) may be further restricted without actually instantiating the generic template. The generality of the pattern also implies that genericity is available for classes, procedures, functions, exceptions, coroutines, and processes.

Patterns may be handled as first-order values in BETA. This implies the possibility of defining pattern variables which can be assigned pattern references dynamically at runtime. This gives the possibilities for a very dynamic handling of patterns at runtime.

Exception handling is dealt with through a predefined library containing basic exception handling facilities. The exception handling facilities are fully implemented within the standard BETA language in the form of a library pattern, and the usage is often in the form of virtual patterns, inheriting from this library pattern.

Garbage collection is conducted automatically by the BETA runtime system when it is discovered that no references to the object exist. The garbage collection mechanism is based on generation-based scavenging. The implemented garbage collection system is very efficient.

4.11 VARIOUS OBJECT ORIENTED PROGRAMMING LANGUAGES COMPARATIVE CHART

Language	BETA	C++	Eiffel	Java	Object Pascal	Ruby	Smalltalk
Inheritance	simple	multiple	multiple	simple	simple	simple	simple
Generic classes (templates)	yes	yes	yes	no	no	no	no

Language	BETA	C++	Eiffel	Java	Object Pascal	Ruby	Smalltalk
Strong typing	yes	yes	yes	yes	yes	no	no
Polymorphic	yes	yes	yes	yes	yes	yes	yes
Multithreading	yes	possible	possible	yes	possible	possible	possible
Garbage collection	yes	no	yes	yes	no	yes	yes
Pre-/postconditions	indirect (through patterns)	no	yes	no	no	no	no
Speed	+	+++	++	-	++	-	--
hybrid/OOP	hybrid (pattern)	hybrid	OOP	OOP	hybrid	OOP	OOP
Compiler/interpreter	compiler	compiler	compiler	interpreter	compiler	interpreter	interpreter
Special environment	browser/pretty-printer	-	browser/pretty-printer	-	-	-	class-browser
Special concept	pattern	-	design by contract (Bertrand Meyer)	-	-	everything is an object	everything is an object

Check Your Progress 1

1) What is Object Oriented Programming?

.....

.....

.....

.....

.....

.....

.....

2) What are Object Oriented Tools?

.....

.....

.....

.....

.....

.....

3) What is Object Oriented Analysis and Design?

.....

.....

.....

.....

.....

4.12 SUMMARY

Object-orientation is a new programming concept, which should help you in developing high quality software. Object-orientation makes developing of projects easier. Complex software systems become easier to understand, since object-oriented structuring provides a closer representation of reality than other programming techniques. In a well-designed object-oriented system, it should be possible to implement changes at class level, without having to make alterations at other points in the system. This reduces the overall amount of maintenance required. Through polymorphism and inheritance, object-oriented programming allows you to reuse individual components. In an object-oriented system, the amount of work involved in revising and maintaining the system is reduced, since many problems can be detected and corrected in the design phase. This unit have presented an overview of some of the important object oriented languages.

4.13 MODEL ANSWERS

Check Your Progress 1

- 1) A type of programming in which programmers define not only the data type of a data structure, but also the types of operations (functions) that can be applied to the data structure. In this way, the data structure becomes an object that includes both data and functions. In addition, programmers can create relationships between one object and another. For example, objects can inherit characteristics from other objects.

One of the principal advantages of object-oriented programming techniques over procedural programming techniques is that they enable programmers to create modules that do not need to be changed when a new type of object is added. A programmer can simply create a new object that inherits many of its features from existing objects. This makes object-oriented programs easier to modify.

- 2) Object-oriented tools allow you to create object-oriented programs in object-oriented languages. They allow you to model and store development objects and the relationships between them.
- 3) OO Analysis - Examination of requirements from the perspective of the classes/objects found in the problem domain.

OODesign - Uses OO decomposition and a notation for depicting logical/physical and static/dynamic models of the system.

UNIT 5 AN INTRODUCTION TO UNIFIED MODELING LANGUAGE (UML)

Structure

- 5.0 Introduction
- 5.1 Objectives
- 5.2 What is UML?
 - 5.2.1 Goals of UML
 - 5.2.2 Why use UML?
 - 5.2.3 History of UML
 - 5.2.4 Why do We Need UML at All?
- 5.3 Definitions
- 5.4 The UML Diagrams
 - 5.4.1 Use case Diagrams
 - 5.4.2 Class Diagrams
 - 5.4.3 Interaction Diagrams
 - 5.4.4 State Diagrams
 - 5.4.5 Activity Diagrams
 - 5.4.6 Physical Diagrams
- 5.5 Summary
- 5.6 Model Answers

5.0 INTRODUCTION

In the previous units, we have discussed about the concepts of object oriented programming and languages supporting OOP. One of the major asset of object oriented programming is its reusability. The reusability requires proper design of classes and inheritance hierarchy. To deal with business problems one need to do proper design to ensure quick reusable and reliable solution. Object oriented modeling is an approach to analyse system behaviours, thus, proposing a realistic and proper solution or design to a problem. There are many modeling techniques in this regard. However, a detailed discussion on all such techniques is beyond the scope of this unit. For the purpose of an introduction to object oriented analysis and design, we have selected unified modeling language (UML) for software development. This unit does not attempt to provide a detailed analysis or design methodology but is an attempt to make you familiar with some diagrammatic tools that with your knowledge of Software Engineering may be used as a first cut object analysis and design.

5.1 OBJECTIVES

At the end of this unit you will be able to:

- Describe the UML concept;
- Present simple design for simple problem, and
- Identify various diagrams used in UML.

5.2 WHAT IS UML?

The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems.

5.2.1 Goals of UML

The primary goals in the design of the UML were:

- 1) Provide users with a ready-to-use, expressive visual modeling language so that they can develop and exchange meaningful models.
- 2) Provide extensibility and specialization mechanisms to extend the core concepts.
- 3) Be independent of particular programming languages and development processes.
- 4) Provide a formal basis for understanding the modeling language.
- 5) Encourage the growth of the OO tools market.
- 6) Support higher-level development concepts such as collaborations, frameworks, patterns and components.
- 7) Integrate best practices.

5.2.2 Why Use UML?

As the strategic value of software increases for many companies, the industry looks for techniques to automate the production of software and to improve quality and reduce cost and time-to-market. These techniques include component technology, visual programming, patterns and frameworks. Businesses also seek techniques to manage the complexity of systems as they increase in scope and scale. In particular, they recognize the need to solve recurring architectural problems, such as physical distribution, concurrency, replication, security, load balancing and fault tolerance. Additionally, the development for the World Wide Web, while making some things simpler, has increased these architectural problems. The Unified Modeling Language (UML) was designed to respond to these needs.

5.2.3 History of UML

The development of UML began in late 1994 when Grady Booch and Jim Rumbaugh of Rational Software Corporation began their work on unifying the Booch and OMT (Object Modeling Technique) methods. In the Fall of 1995, Ivar Jacobson and his Objectory company joined Rational and this unification effort, merging in the OOSE (Object-Oriented Software Engineering) method.

As the primary authors of the Booch (Grady Booch's work has involved the use of application of data abstraction and information hiding with an emphasis on iterative software development), OMT, and OOSE (Object Oriented Software Engineering) methods, Grady Booch, Jim Rumbaugh and Ivar Jacobson were motivated to create a unified modeling language for three reasons.

First, these methods were already evolving toward each other independently. It made sense to continue that evolution together rather than apart, that would further confuse users.

Second, by unifying the semantics and notation, they could bring some stability to the object-oriented marketplace, allowing projects to settle on one mature modeling language and letting tool builders focus on delivering more useful features.

Third, they expected that their collaboration would yield improvements in all three earlier methods, helping them to capture lessons learned and to address problems that none of their methods previously handled well.

The efforts of Booch, Rumbaugh, and Jacobson resulted in the release of the UML 0.9 and 0.91. Several organizations saw UML as strategic to their business such as IBM, ObjecTime, Platinum Technology, Ptech, Taskon, Reich Technologies, these companies joined the UML partners to contribute their ideas, and together the partners produced the UML 1.1.

What is the benefit of UML for users?

UML is based on OMT, Booch, OOSE and other important modeling languages available. It is a fusion of OMT, Booch and OOSE techniques for software development and software architecture. Those who have been trained on these three languages will have little trouble getting to work with UML. It provides the opportunity for new integration between tools, processes, and domains. Also it enables developers to focus on delivering business value and provides them a paradigm to accomplish this.

5.2.4 Why do we need UML at all ?

As we have described above UML as "UML is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems." The question arises why do we need to carry out such an exercise? To answer this we need to understand the problem domain, its solution domain and the problem solving approach.

Problem can be termed as Requirements of Organisations or their customers. The Solutions of the problem is the desired services or products, as the case may be. To deliver value added solutions (maximum quantity and minimum cost and within the minimum time), organisations must capture knowledge, communicate and leverage knowledge. By capturing knowledge we mean acquiring it, by communicating it we mean share it and by leveraging it we mean utilizing it. The main aspect we need to deal here is "communication or sharing" of knowledge. There is an old adage, which says "A picture is worth thousand words", when we express knowledge by visual tools we are able to deliver (communicate) its contents in a better manner. That's what UML is all about-expressing captured knowledge.

The problem occurs within business context (domain). The solution must also fit in organisations IT infrastructure. The problem must be fully understood in terms of business requirements and the information system must be fully understood of how it meets those requirements. As we conceptualize the problem and work towards its solution we capture knowledge (models), make decisions (architectural views) about how we will address different issues and communicate information (diagrams). UML does it all.

5.3 DEFINITIONS

UML concept as we know is based on Object Oriented approach; we need to define certain OO concepts before marching into UML's world.

Objects

Objects are the real world models. They are well defined representational constructs. Objects are the physical and conceptual things we find in the world. When something is called as an object in our world, we associate it with a name, properties etc. Objects encapsulate structural characteristics known as attributes. They contain their own data and programming.

Classes

Classes are descriptions of objects, they contain objects with similar characteristics. Classes encapsulates behavioural characteristics called as operations.

Links

Links are representational constructs that define how the classes are related to each other. Links are objects.

Associations

Associations are representational constructs that describes links. Associations are classes.

5.4 THE UML DIAGRAMS

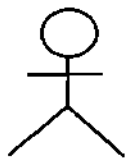
The UML defines various types of Diagrams : Class, Object, Use Case, Sequence, Collaboration, Statechart, Activity, Component and Deployment diagrams.

5.4.1 Use Case Diagrams

Use Case diagrams describe the functionality of a system and users of a system. These diagrams contain following items:

Actors—Which are users of system, including Human beings and other system components

Use Cases—Which includes services provided to Users of the system.



Actor



Use Case

An actor represents a user or another system that will interact with the system you are modeling. A use case is an external view of the system that represents some action the user might perform in order to complete a task.

When to Use Use Cases Diagrams

Use cases are used in almost every project. They are helpful in exposing requirements and planning the project. During the initial stage of a project most use cases should be defined, but as the project continues more might become visible.

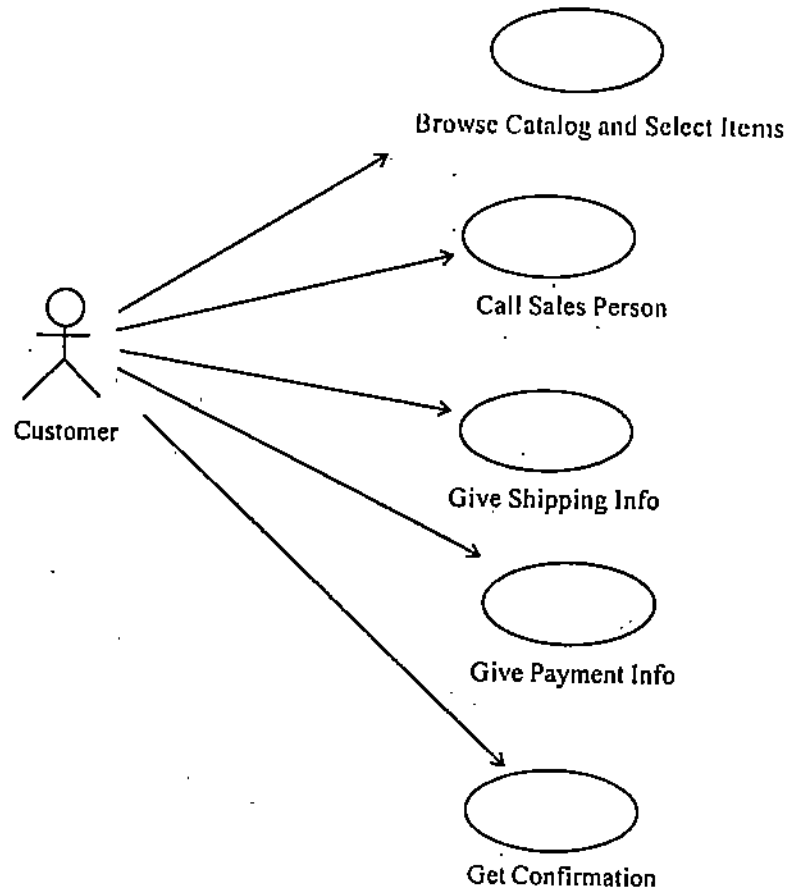
How to Draw Use Cases Diagrams

Use cases are a relatively easy UML diagram to draw.

Start by listing a sequence of steps a user might take in order to complete an action. For example, a user placing an order with a sales company might follow these steps.

- 1) Browse catalog and select items.
- 2) Call sales representative.
- 3) Supply shipping information.
- 4) Supply payment information.
- 5) Receive confirmation number from salesperson.

These steps would generate this simple use case diagram:



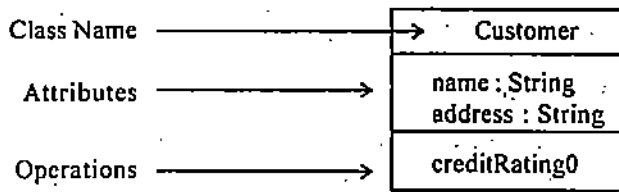
This example shows the customer as an actor because the customer is using the ordering system. The diagram takes the simple steps listed above and shows them as actions the customer might perform.

From this simple diagram the requirements of the ordering system can easily be derived. The system will need to be able to perform actions for all of the use cases listed. As the project progresses other use cases might appear. The customer might have a need to add an item to an order that has already been placed. This diagram can easily be expanded until a complete description of the ordering system is derived capturing all the requirements that the system will need to perform.

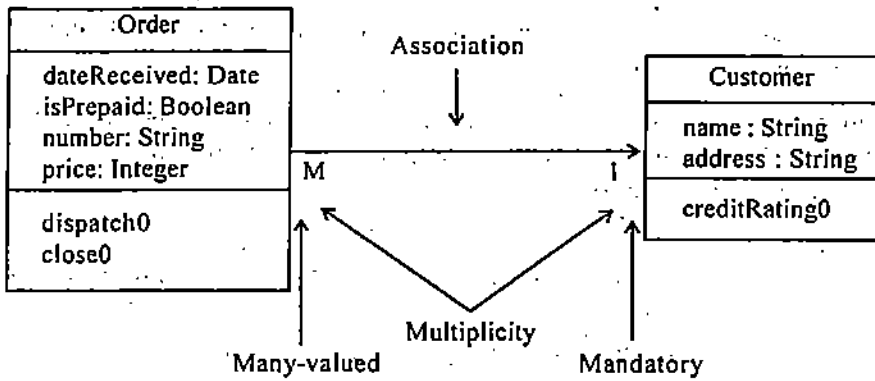
5.4.2 Class Diagrams

Class diagrams are widely used to describe the types of objects in a system and their relationships. Class diagrams model class structure and contents using design elements such as classes, packages and objects. Class diagrams describe three different perspectives when designing a system, conceptual, specification, and implementation. These perspectives become evident as the diagram is created and help solidify the design.

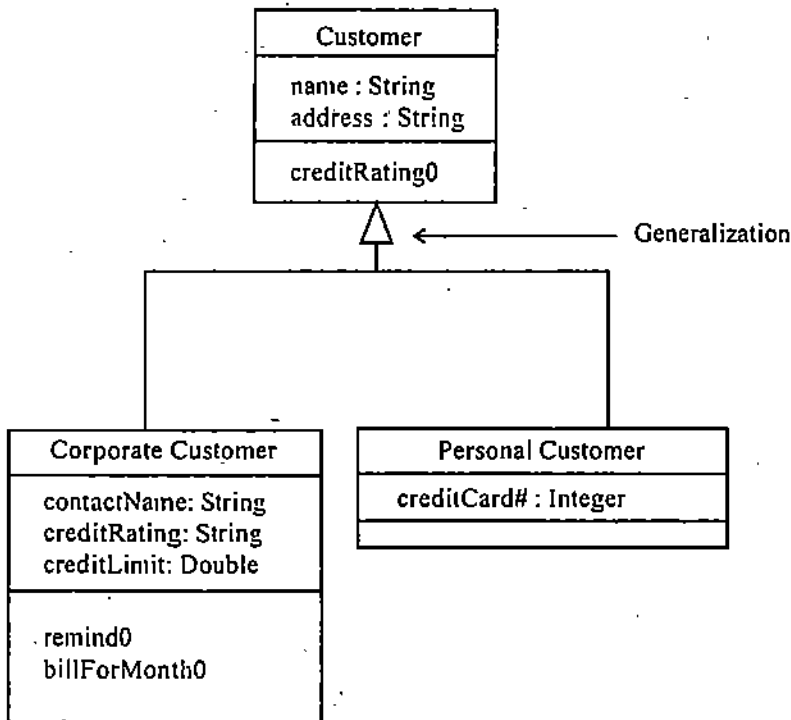
Classes are composed of three things: a name, attributes, and operations/ methods. Below is an example of a class.



Class diagrams also display relationships such as containment, inheritance, associations and others. Below is an example of an associative relationship:



The association relationship is the most common relationship in a class diagram. The association shows the relationship between instances of classes. For example, the class Order is associated with the class Customer. The multiplicity of the association denotes the number of objects that can participate in the relationship. For example, an Order object can be associated to only one customer, but a customer can be associated to many orders.



Another common relationship in class diagrams is a generalization. A generalization is used when two classes are similar, but have some differences. Look at the generalization below:

In this example the classes Corporate Customer and Personal Customer have some similarities such as name and address, but each class has some of its own attributes and operations. The class Customer is a general form of both the Corporate Customer and Personal Customer classes. This allows the

designers to just use the Customer class for modules and do not require in-depth representation of each type of customer.

When to Use Class Diagrams

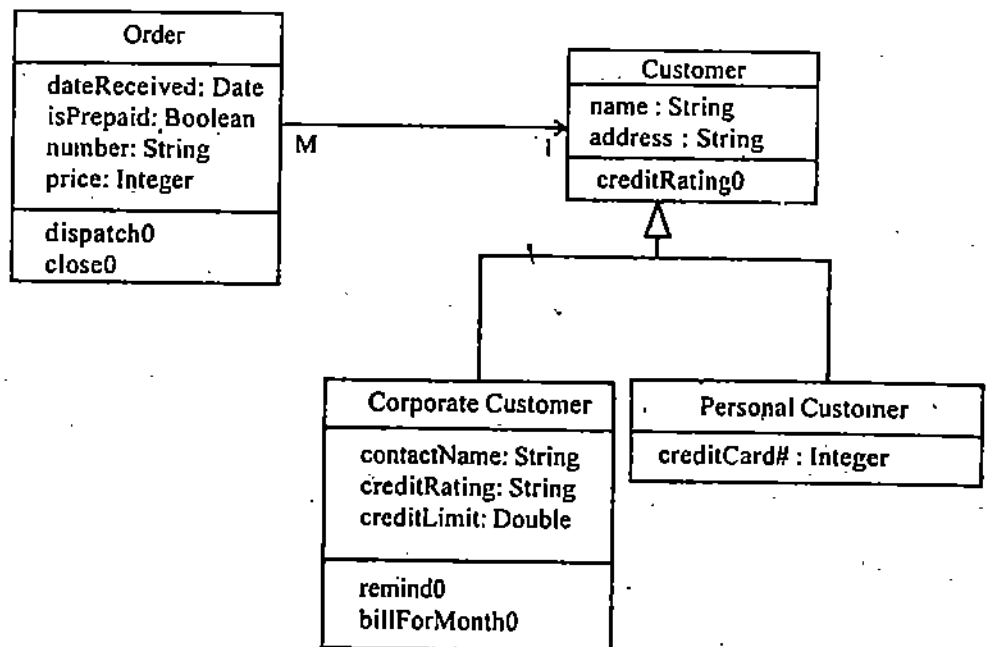
Class diagrams are used in nearly all Object Oriented software designs. Use them to describe the Classes of the system and their relationships to each other.

How to Draw Class Diagrams

Class diagrams are some of the most difficult UML diagrams to draw. To draw detailed and useful diagrams a person would have to study UML and Object Oriented principles for a long time.

Before drawing a class diagram consider the three different perspectives of the system the diagram will present; conceptual, specification, and implementation. Try not to focus on one perspective and try to see how they all work together.

When designing classes consider what attributes and operations it will have. Then try to determine how instances of the classes will interact with each other. These are the very first steps of many in developing a class diagram. However, using just these basic techniques one can develop a complete view of the software system.



5.4.3 Interaction Diagrams

Interaction diagrams model the behaviour of use cases by describing the way groups of objects interact to complete the task. The two kinds of interaction diagrams are sequence and collaboration diagrams.

When to Use: Interaction Diagrams

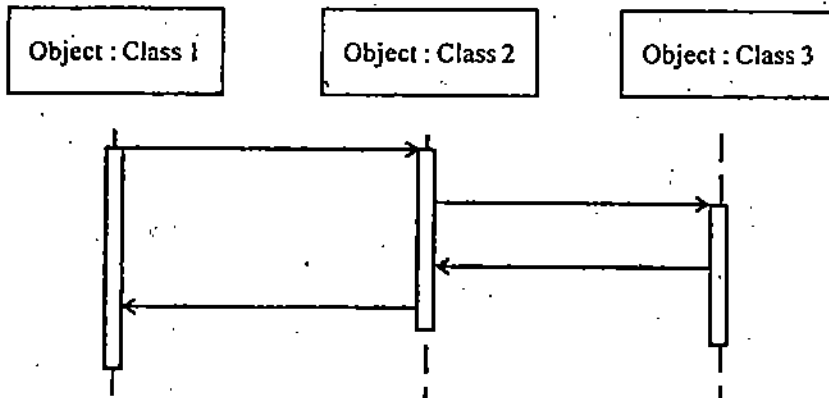
Interaction diagrams are used when you want to model the behaviour of several objects in a use case. They demonstrate how the objects collaborate for the behaviour. Interaction diagrams do not give an in-depth representation of the behaviour. If you want to see what a specific object is doing for several use cases use a state diagram. To see a particular behaviour over many use cases or threads use an activity diagram.

How to Draw Interaction Diagrams

Sequence diagrams, collaboration diagrams, or both diagrams can be used to demonstrate the interaction of objects in a use case. Sequence diagrams generally show the sequence of events that occur. Collaboration diagrams demonstrate how objects are statically connected. Both diagrams are relatively simple to draw and contain similar elements.

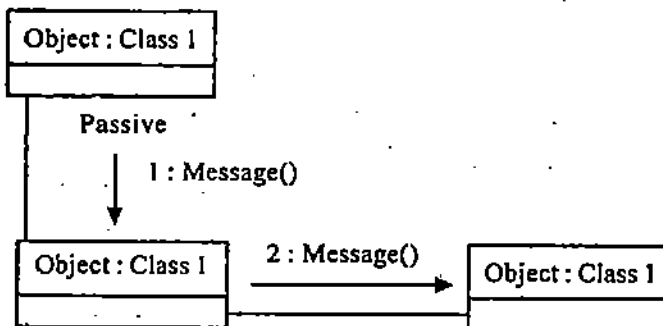
Sequence diagrams

Sequence diagrams describe interactions among classes. These interactions are exchange of messages. Sequence diagrams demonstrate the behaviour of objects in a use case by describing the objects and the messages they pass. The example below shows an object of class 1 start the behaviour by sending a message to an object of class 2. Messages pass between the different objects until the object of class 1 receives the final message.



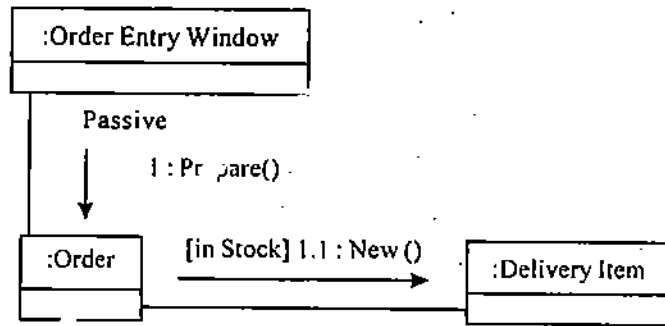
Collaboration diagrams

Collaboration diagrams describe interactions among classes and associations. Collaboration diagrams are also relatively easy to draw. They show the relationship between objects and the order of messages passed between them.



The objects are listed as icons and arrows indicate the messages being passed between them. The numbers next to the messages are called sequence numbers. As the name suggests, they show the sequence of the messages as they are passed between the objects. There are many acceptable sequence numbering schemes in UML. A simple 1, 2, 3... format can be used, as the example shows, or for more detailed and complex diagrams a 1, 1.1, 1.2, 1.2.1... scheme can be used.

The following example shows a simple collaboration diagram for the placing an order use case. This time the names of the class appear after the colon, such as Order Entry Window; following the object Name: className naming convention, the class name is shown to demonstrate that all of objects of that class will behave the same way.



5.4.4 State Diagrams

State diagrams are used to describe the behaviour of a system, i.e., the states and responses of a class. They describe the behaviour of a class in response to external stimuli. State diagrams describe all of the possible states of an object as events occur. Each diagram usually represents objects of a single class and track the different states of its objects through the system.

These diagrams contain the following elements:

States which represents the state of an object during its life cycle in which it satisfies some condition.

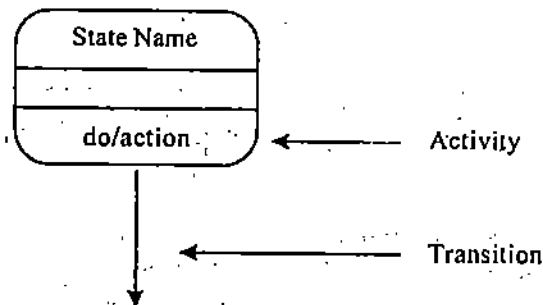
Transitions which represents relationship between different states of an object.

When to Use State Diagrams

State diagrams are used to demonstrate the behaviour of an object through many use cases of the system. Only use state diagrams for classes where it is necessary to understand the behaviour of the object through the entire system. Not all classes will require a state diagram and state diagrams are not useful for describing the collaboration of all objects in a use case. State diagrams are combined with other diagrams such as interaction diagrams and activity diagrams.

How to Draw State Diagrams

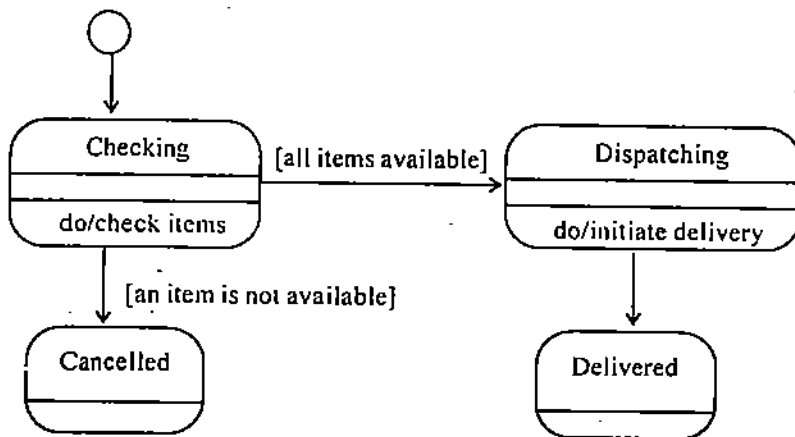
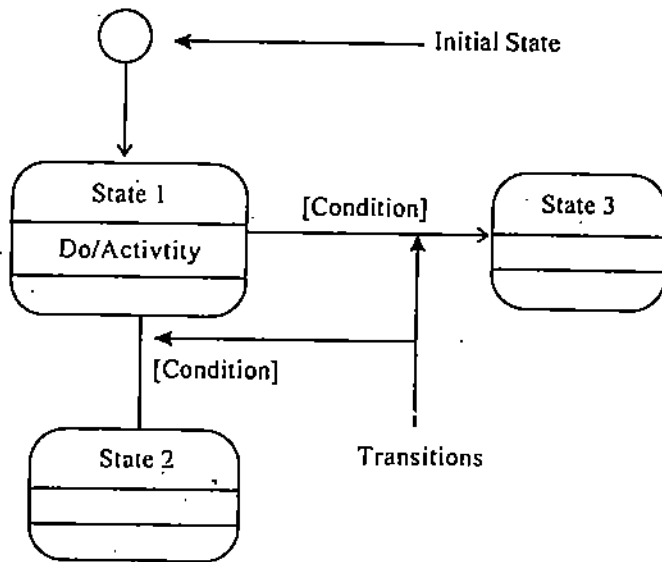
State diagrams have very few elements. The basic elements are rounded boxes representing the state of the object and arrows indicating the transition to the next state. The activity section of the state symbol depicts what activities the object will be doing while it is in that state.



All state diagrams begin with an initial state of the object. This is the state of the object when it is created. After the initial state the object begins changing states. Conditions based on the activities can determine what the next state the object transitions to.

Below is an example of a state diagram might look like for an Order object. When the object enters the Checking state it performs the activity "check items." After the activity is completed the object transitions to the next state based on the conditions [all items available] or [an item is not available]. If an

item is not available the order is cancelled. If all items are available then the order is dispatched. When the object transitions to the Dispatching state the activity "initiate delivery" is performed. After this activity is complete the object transitions again to the delivered state.



5.4.5 Activity Diagrams

Activity diagram describe the activities of a class. It is similar to state diagram but describes the behaviour of a class in response to internal processing instead of external stimuli. Activity diagrams describe the workflow behaviour of a system. Activity diagrams are similar to state diagram because activities are the state of doing something. The diagrams describe the state of activities by showing the sequence of activities performed. Activity diagrams can show activities that are conditional or parallel.

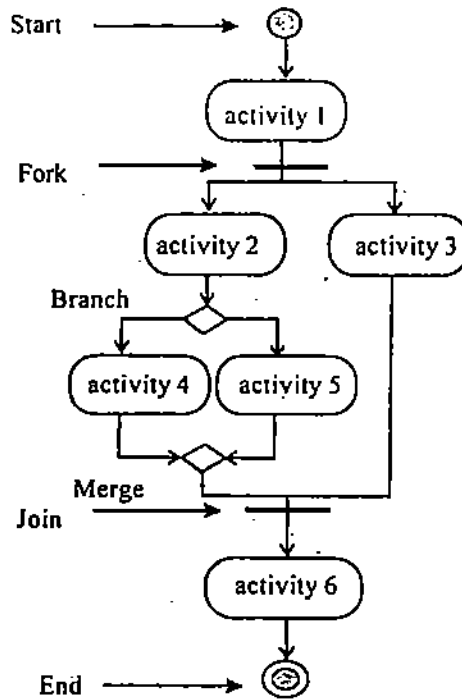
When to Use Activity Diagrams

Activity diagrams should be used in conjunction with other modeling techniques such as interaction diagram and state diagram. The main reason to use activity diagrams is to model the workflow behind the system being designed. Activity Diagrams are also useful for analyzing a use case by describing what actions need to take place and when they should occur; describing a complicated sequential algorithm; and modeling applications with parallel processors.

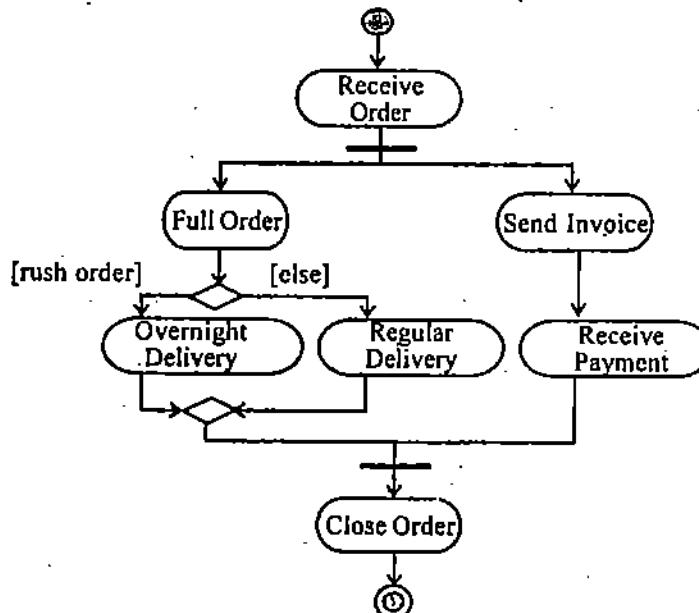
However, activity diagrams should not take the place of interaction diagram and state diagram. Activity diagrams do not give details about how objects behave or how objects collaborate.

How to Draw Activity Diagrams

Activity diagrams show the flow of activities through the system. Diagrams are read from top to bottom and have branches and forks to describe conditions and parallel activities. A fork is used when multiple activities are occurring at the same time. The diagram below shows a fork after activity 1. This indicates that both activity 2 and activity 3 are occurring at the same time. After activity 2 there is a branch. The branch describes what activities will take place based on a set of conditions. All branches at some point are followed by a merge to indicate the end of the conditional behaviour started by that branch. After the merge all of the parallel activities must be combined by a join before transitioning into the final activity state.



Below is a possible activity diagram for processing an order. The diagram shows the flow of actions in the system's workflow. Once the order is received the activities split into two parallel sets of activities. One side fills and sends the order while the other handles the billing. On the Fill Order side, the method of delivery is decided conditionally. Depending on the condition either the Overnight Delivery activity or the Regular Delivery activity is performed. Finally the parallel activities combine to close the order.



5.4.6 Physical Diagrams

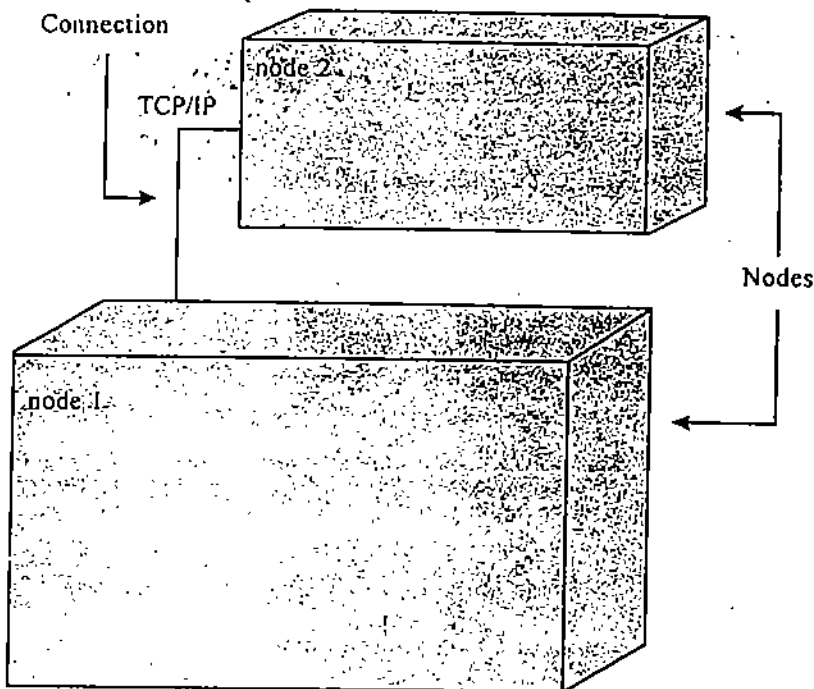
There are two types of physical diagrams: deployment diagrams and component diagrams. Deployment diagrams show the physical relationship between hardware and software in a system. Component diagrams show the software components of a system and how they are related to each other. These relationships are called dependencies.

When to Use Physical Diagrams

Physical diagrams are used when development of the system is complete. Physical diagrams are used to give descriptions of the physical information about a system.

How to Draw Physical Diagrams

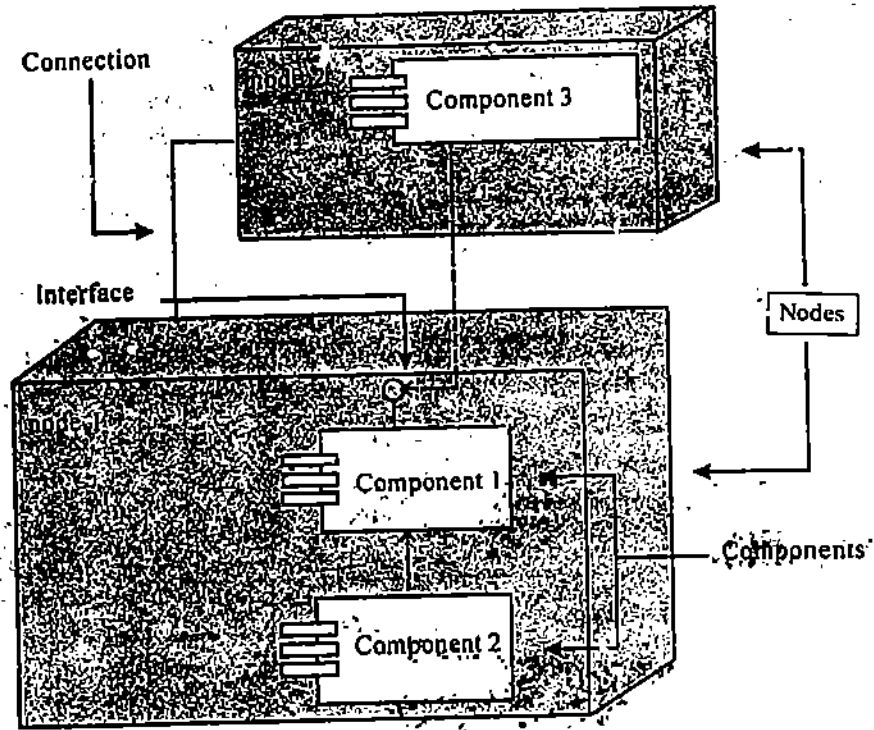
Many times the deployment and component diagrams are combined into one physical diagram. A combined deployment and component diagram combines the features of both diagrams into one diagram.



The deployment diagram contains nodes and connections. A node usually represents a piece of hardware in the system. A connection depicts the communication path used by the hardware to communicate and usually indicates a method such as TCP/IP.

The component diagram contains components and dependencies. Components represent the physical packaging of a module of code. The dependencies between the components show how changes made to one component may affect the other components in the system. Dependencies in a component diagram are represented by a dashed line between two or more components. Component diagrams can also show the interfaces used by the components to communicate to each other.

The combined deployment and component diagram below gives a high level physical description of the completed system. The diagram shows two nodes which represent two machines communicating through TCP/IP. Component 2 is dependant on component 1, so changes to component 2 could affect component 1. The diagram also depicts component 3 interfacing with component 1. This diagram gives the reader a quick overall view of the entire system.



Check Your Progress

State True (T) or False (F)

- 1) a) UML is used for waterfall model based implementation. True False
- b) UML increases burden on designers and implements. True False
- c) If you are using UML then additional documentation tool is not needed. True False
- d) An association in UML is an object. True False
- e) An actor is a user who interact with the system to be modeled. True False
- f) Class diagrams are very close to E R diagrams True False
- g) State diagrams can be used to represent interaction. True False
- h) Activity diagram describe class and objects. True False

5.5 SUMMARY

In this last unit, we have discussed about a modeling language which is getting popular in object oriented analysis and design. The unit presents the basic objectives of the methodology and various types of diagrams represented in UML. It is advisable that you may explore the further readings and web site to look for more example of UML.

5.6 MODEL ANSWERS

Check Your Progress

- 1) a) False
- b) False
- c) True
- d) False
- e) True
- f) True
- g) False
- h) False



UTTAR PRADESH RAJARSHI
TANDON OPEN UNIVERSITY

BCA-17 C++ and Object Oriented Programming

Block

2

C++ - AN INTRODUCTION

UNIT 1

Overview of C++ 5

UNIT 2

Classes and Objects 25

UNIT 3

Operator Overloading 38

UNIT 4

Inheritance-Extending Classes 48

UNIT 5

Streams and Templates 56

FACULTY OF THE SCHOOL

Prof. Manohar Lal
Director

Prof. M.M. Pant

Shri Shashi Bhushan
Reader, IGNOU

Shri Akshay Kumar
Reader, IGNOU

P. V. Suresh
Lecturer, IGNOU

Shri V. V. Subrahmanyam
Lecturer, IGNOU

COURSE CO-ORDINATOR

Shri Akshay Kumar
Reader, IGNOU

BLOCK WRITERS

Shri Akshay Kumar
Reader, IGNOU

P. V. Suresh
Lecturer, IGNOU

EXPERT COMMITTEE FOR BCA

Prof. P.S. Grover
Professor of Computer Science
University of Delhi
Delhi

Dr. S.C. Mehta
Sr. Director
Manpower Development Division
Department of Electronics
Govt. of India
New Delhi

Prof. R.G. Gupta
School of Computer and
Systems Science
Jawaharlal Nehru University
Delhi

Brig. V.M. Sundaram
Coordinator
DoE-ACC Centre
New Delhi

Dr. G. Haider
Director
Information Technology Centre
TCIL, Delhi

Dr. Sugata Mitra
Principal Scientist
National Institute of
Information Technology
New Delhi

Prof. Karmeshu
School of Computer and
Systems Sciences
Jawaharlal Nehru University
Delhi

Prof. H.M. Gupta
Department of Electrical
Engineering
Indian Institute of Technology
Delhi

Prof. Sudhir Kaicker
Director
Computer Centre
Jawaharlal Nehru University
Delhi

Prof. L.M. Patnaik
Indian Institute of Science
Bangalore

Prof. S. Sadagopan
Department of Industrial
Engineering
Indian Institute of Technology
Kanpur

Prof. M.M. Pant
School of Computer
and
Information Sciences
IGNOU, New Delhi

PRINT PRODUCTION

H. K. Som
IGNOU

July, 2002

© Indira Gandhi National Open University, July, 2002

ISBN-81-266-0497-2

All rights reserved. No part of this work may be reproduced in any form, by mimeograph or any other means, without permission in writing from the Indira Gandhi National Open University.

BLOCK INTRODUCTION

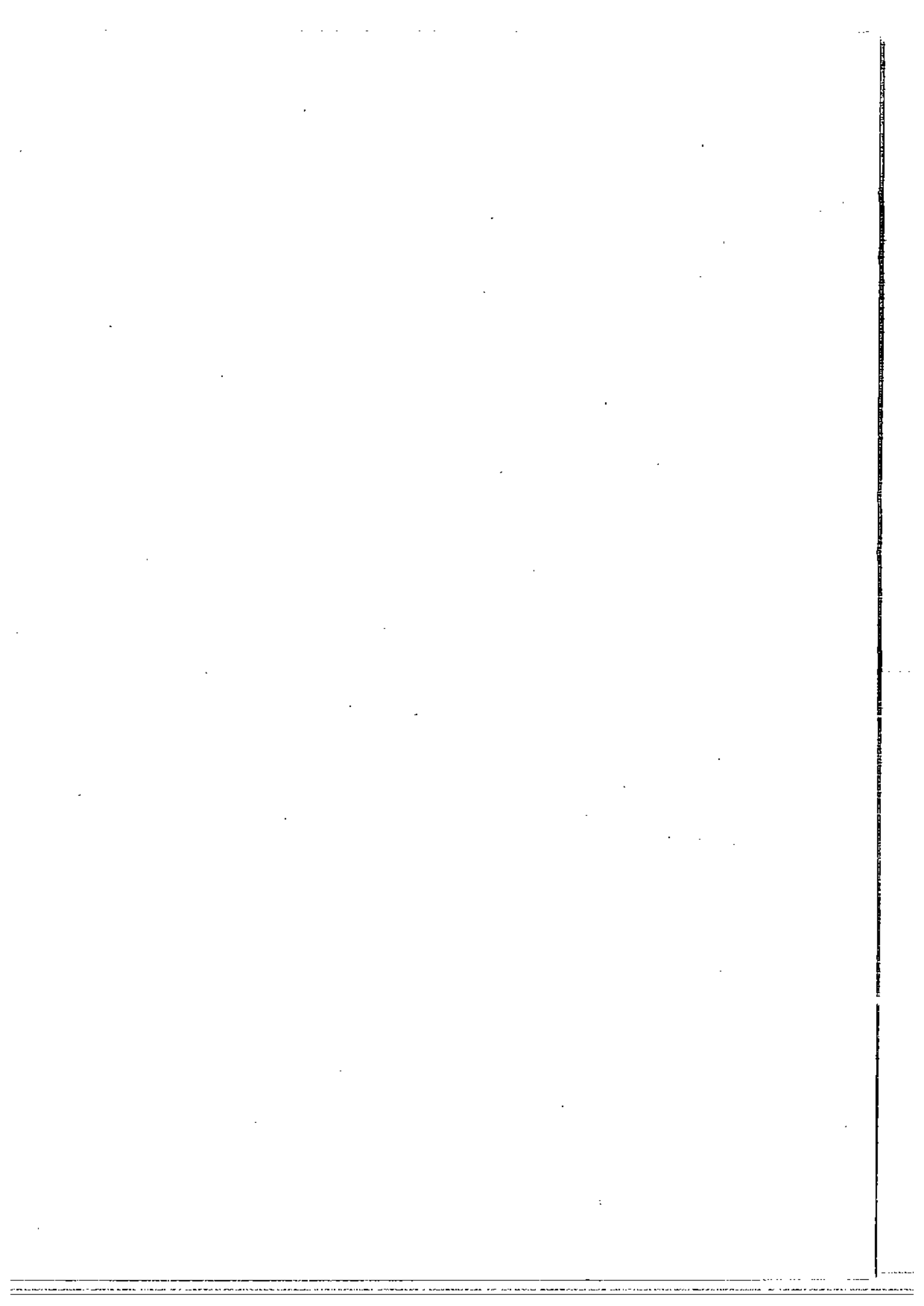
This block provides an in-depth coverage of C++ Programming Language. The C++ Programming language at present is one of the most commonly used object oriented Programming Language. However, it is not the safest of object oriented programming language. The main concept, which makes C++ slightly unsafe, is the pointer feature. Most of the latest Object Oriented Programming Languages such as JAVA and C# (pronounced C-Sharp) removes such direct features and are much simpler to use. They have also attempted to achieve platform independence through intermediate codes.

This block is divided into five units.

Unit-1 provides an overview of C++ Programming Language where various programming paradigms have been introduced again. In addition, the concepts like functions, macros in C++ have been introduced. Unit 2 discusses issues relating to classes and objects in C++. Unit 3 focuses on the operator overloading in C++. Unit 4 discusses inheritance in C++. Unit 5 discusses streams and templates.

References

1. Bjarne Stroustrup, The C++ Programming Language, Third edition. Pearson, Publication.
2. N. Barkakati, Object Oriented Programming in C++, Prentice Hall of India.



UNIT 1 OVERVIEW OF C++

Structure

- 1.0 Introduction
- 1.1 Objectives
- 1.2 Programming Paradigms
 - 1.2.1 Procedural Programming
 - 1.2.2 Modular Programming
 - 1.2.3 Data Abstraction
 - 1.2.4 Object Oriented Programming
- 1.3 C++ Programming Language: A re-visit of Concepts of C/C++
- 1.4 Functions and Files
 - 1.4.1 How to make a Library
 - 1.4.2 Functions
 - 1.4.3 Macros
- 1.5 Summary
- 1.6 Model Answers
- 1.7 Further Readings

1.0 INTRODUCTION

In this unit, some important features of C++ have been discussed. But, all these topics will be dealt in the remaining units more elaborately. The Object Oriented Programming Paradigm features and the need for such a paradigm have also been introduced in this unit. Though there is an advantage of saving the time for transfer of control and storing return addresses in the case of Macros, several disadvantages were also present and they were discussed in this unit. The time to write a program is reduced due to the presence of library routines. There are instances when we use some routines frequently which are not part of library. We usually write the routine repeatedly when we are using it in different programs. To avoid this, we can always make a library of routines. The method of making a library of our own routines is also discussed in this unit.

1.1 OBJECTIVES

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

- Differentiate different programming paradigms
- Define various statements of C++
- Use functions in C++
- Make a library using C++
- Define macros in C++

1.2 PROGRAMMING PARADIGMS

Programming is an art. This art of programming has seen many evolutions. The basic idea for evolution was to develop simpler, maintainable, dependable, efficient, reusable programs. This section traces the evolution of various programming paradigms.

1.2.1 Procedural Programming

In this paradigm, we divide a problem into sub-problems recursively and then we write a procedure for each sub-problem. Procedures communicate with each other by parameters. In this paradigm, data structures are declared locally or globally. The procedures can contain local data. Pascal and C support this programming approach. For example, if you want to implement a stack, you divide the problem into smaller problems like how to push value in a stack, how to pop a value and how to find whether stack is full or empty. Then you write functions for Push, Pop, stack empty and stack full and call these functions using its parameters to implement a stack in a program. The stack size is declared in main program using a data structure (may be an array).

1.2.2 Modular Programming

In this paradigm, the program is divided into modules. Each module will contain all the procedures, which are related to each other and the data on which they act. Modular programming is the other name for Data hiding principle. The reason for this name is that the data in a module cannot be accessed by the procedures in another module unless it has been specified that it can be done so. Modula 2 supports this notion. For example, the complete set of operations of a stack including the data structure and functions may reside in a module.

The advantages of this paradigm are that we can have different files for a simple program. Each file can be separately compiled and an executable file can be made from them. So, if there is any error in one module, the entire program need not be compiled. Only the module in which the error is present can be recompiled separately. This will reduce the total compilation time.

C enables modular programming by providing provision for including files and separate compilation facilities. The context of module is supported by the class concept of C++.

1.2.3 Data Abstraction

Data Abstraction = Modular programming + Data hiding principle.

This concept is supported by C++. In C++, classes can be defined in which the data can be specified as Private. This Private data can be accessed only by the member functions of the class. Then, we can define the class as a user defined data type which is the other name of Data Abstraction.

So, in this paradigm, we have to decide the types we want and then, we have to provide a full set of operations for each type. For example, in the data structure of stack, the data stored in it cannot be accessed or modified by any other class except the functions of stack class then it can be classified as Data Abstraction.

1.2.4 Object Oriented Programming

When we define data types (user defined data types), we may find commonality among them. Also, when we think of defining a new class, we may find that there is another class which is already possessing most of the features of the class, we wish to have. Under such a situation, we can define a class with only additional features and explicitly state that it includes all the features of another class, which has been already defined.

This concept is known as inheritance. The object oriented programming paradigm is made up of Abstraction and Inheritance.

So, OOP = Data Abstraction + Inheritance.

In this paradigm, we have to decide the classes we want; provide a full set of operations for each class and then we have to make commonality explicit by using inheritance. These concepts will be further dealt in more details in this Block.

Check Your Progress 1

1) The various programming paradigms are _____

2) C++ supports:

(a) Procedural Programming

(b) Modular Programming

(c) Object Oriented Programming

Mark the correct answer(s).

3) Data abstraction is a _____ of Object Oriented Programming paradigm.

1.3 C++ PROGRAMMING LANGUAGE: A RE-VISIT OF CONCEPTS OF C/C++

In this section, we will define the term expression and discuss different type of C++ operations.

An expression is composed of one or more operations. The objects of an operation are referred to as operands. Operators represent the operations.

Operators that act on one operand are referred to as 'Unary' operators.
Example: -6 (unary minus).

Operators that act on two operands are referred to as 'Binary' operators.
Example: $x * y$ (multiplication) $2 + 5$ (addition).

The evaluation of an expression results in one or more operations, yielding a result. When two or more operations are combined, the expression is referred to as a 'compound' Expression. The "precedence" and "associativity" of the operators determine the order of the operator evaluation.

The simplest form of an expression consists of a single literal constant of a variable. This "operand" is without an operator. The result is the operand's value. For example, here are three simple expressions:

3.14159

"malancholia"

UpperBound.

The result of 3.14159 is 3.14159. Its type is Double. The result of malancholia is the address in memory of the first element of the string. Its type is char*. The result of UpperBound is its rvalue (that is the value bounded with the variable. In other words, the present value stored in the location held by this variable). Its type is determined by its definition.

Arithmetic Operators

The following table lists the Arithmetic operators:

Operators	Function	Use
*	Multiplication	expr * expr
/	Division	expr / expr
%	Modulus(Remainder)	expr % expr
+	Addition	expr + expr
-	Subtraction	expr - expr

Division between two integers results in an integer output. If the quotient contains a fractional part, it is truncated, example:

$$21/6 = 3$$

The modulus operator (%) can be applied only to integers. The left operand of the % is dividend. The divisor is the operator's right operand. Example:

```
86.3 % 5          // error: floating point operand
65 % 3            // OK: Result is 2
40 % 8            // OK: Result is 0
```

Equality, Relational and Logical Operators

The equality, relational and logical operators evaluate to either true or false. A true condition yields 1; a false condition yields 0. The following table lists the Equality, Relational and Logical Operators:

Operator	Function
!	logical NOT
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
==	equality
!=	inequality
&&	logical AND
	logical OR

Except of ! all are binary operators.

The logical AND ("&&") operator evaluates to true only if both its operators evaluate to true. The logical OR ("||") operator evaluates to true if either of its operands evaluates to true. The operators are evaluated from left to right.

Evaluation stops as soon as the truth or falsity of the expression is determined. The logical NOT ("!") operator evaluates to true if its operand has a value of Zero. Otherwise, it evaluates to false.

Assignment operators

The left operand of the assignment operator ("=") must be an lvalue. The term lvalue is derived from the variable position to the left of the assignment operation. You might think of lvalue as meaning location value. The effect of an assignment is to store a new value in the storage associated with left operand. For example, given the following three definitions:

```
int i, *ip, ia[4];
```

the following are legal assignments:

```
ip = &i;
i = ia[0] + 1;
ia[*ip] = 1024;
*ip = i*2 + i*a[i];
```

The data type of the result is the type of its left operand. Assignment operators can be concatenated provided that each of the operands being assigned is of the same general data type. For example,

```
int i, j;
i = j = 0; // OK: each assigned 0
0 is assigned to j and i. The order of evaluation is right to left.
```

The compound assignment operator also provides a measure of notational compactness. For example,

```
int arrayprod (int ia[], int sz)
{
    int prod = 0;
    for (int i = 0; i < sz; ++i)
        prod *= prod[i]
    return prod;
}
```

The general syntactic form of the compound assignment operator is

```
a op = b;
```

where op = may be one of the following ten operators: +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=. Each compound operator is equivalent to the following long hand assignment:

```
a = a op b;
```

so, a+=b is equivalent to a = a + b.

Increment and Decrement Operators

Two special operators are used in C++, namely incrementer and decrementer. These operators are used to control the loop in an effective manner. There are two types of incrementers : Prefix incrementer (++i) and postfix incrementer (i++).

In prefix incrementer, first it is incremented and then the operations are performed. On the other hand, in postfix incrementer, first the operations are performed and then it is incremented. However, the result of the incremented value will be same in both the cases. For example:

```
i = 8;
x = ++i;
cout << x; // 9 will be printed
j = 7;
y = j++;
cout << y; // 7 will be printed
cout << j; // 8 will be printed.
```

The decrementer is also similar to incrementer. The symbol '--' is used for decrementing. There are two types of decremeters. They are prefix decrementer (--i) and postfix decrementer (i--). For example:

```
i = 8;
x = --i;
cout << x; // 7 will be printed
j = 7;
y = j--;
cout << y; // 7 will be printed
cout << j; // 6 will be printed.
```

The sizeof Operator

The 'sizeof' operator returns the size, in bytes, of an expression or type-specifier. It may occur in either of two forms:

```
sizeof (type-specifier);  
sizeof expr;
```

For example,

```
sizeof (short); // returns the storage allocated to a short integer which is  
                // machine dependent  
sizeof (stack); // stack is a class. So, storage occupied by it, will be returned.
```

The Arithmetic If Operator

The Arithmetic If operator, the only ternary operator in C++, has the following syntactic form:

```
expr1 ? expr2: expr3;
```

expr1 is always evaluated. If it evaluates to a true condition - that is, any non-zero value then expr2 is evaluated, otherwise expr3 is evaluated. The following program illustrates the usage of this operator.

```
# include <iostream.h>  
void main(.)  
{  
    int i = 10, j = 20;  
    cout << "The larger value of " << i << "and" << j << "is" << (i > j ? i : j)  
    << endl;  
}
```

When compiled and executed, the program generates the following output:

The larger value of 10 and 20 is 20.

Comma Operator

A Comma expression is a series of expressions separated by a Comma(s). These expressions are evaluated from Left to Right. The result of a Comma expression is the value of Right most expression. In the following example, each side of the arithmetic if operator is a Comma expression. The value of the first Comma expression is 1; the value of second is 0.

```
main( )  
{ int ival = (ia! = 0) ? ix = index( ), ia[ix] = ix, 1: (set-array (ia), 0);  
}
```

The above expression sets an array if it does not exist or gets and assign a value to an index element equal to the index value and returns 1.

The Bitwise Operators

A bitwise operator interprets its operand(s) as an ordered collection of bits. Each bit may contain either a 0 (off) or a 1 (on) value. A bitwise operator allows the programmer to test and set individual bits.

The operands of the bitwise operation must be of an integral type. (Is char an intigral type? Yes) The following table lists the bitwise operators:

Operator	Function
~	bitwise NOT
<<	Left Shift
>>	Right Shift
&	bitwise AND
^	bitwise XOR
	bitwise OR

The bitwise NOT operator (~) flips the bits of its operands. Each 1 bit is set to 0 and 0 bit is set to 1. For example, unsigned char bit = 0227

1	0	0	1	0	1	1	1
---	---	---	---	---	---	---	---

On applying bitwise NOT operator will give:

0	1	1	0	1	0	0	0
---	---	---	---	---	---	---	---

The bitwise left shift operator ("<<") shifts the bits to the left keeping the same number of bits by dropping shifted bits off the end and filling in with zeroes from the other end.

For example, let x = 33 (0010 0001) (8 bits). Now, x << 1 results in 0100 0010.

The bitwise Right Shift operator (">>") shifts the bits to the Right keeping the same number of bits by dropping shifted bits off the end and filling in with zeroes from other end.

For example, let x = 33 (0010 0001) (8 bits). An operation, x >> 3 results in (0000 0100).

The bitwise AND operator ("&") takes two integral operands. For each bit position, the result is a 1-bit if both operands contain 1 bit; otherwise, the result is a 0-bit. This operator is different from logical AND operator ("&&").

For example:

unsigned char result;

unsigned char b1 = 0145

0	1	1	0	0	1	0	1
---	---	---	---	---	---	---	---

unsigned char b2 = 0257

1	0	1	0	1	1	1	1
---	---	---	---	---	---	---	---

Result = b1 & b2

0	0	1	0	0	1	0	1
---	---	---	---	---	---	---	---

The bitwise XOR (exclusive or) operator ("^") takes two integral operands. For each bit position, the result is a 1-bit if either but not both operands contain a 1-bit. Otherwise, the result is 0-bit. For example, the result on b1^b2 operation would be:

b1^b2 =

1	1	0	0	1	0	1	0
---	---	---	---	---	---	---	---

The bitwise OR operator ("|") takes two integral operands. For each bit position, the result is a 1-bit if either or both operands contain a 1 bit; otherwise, the result is a one bit. For example, the result on b1 | b2 operation would be:

b1 | b2 =

1	1	1	0	1	1	1	1
---	---	---	---	---	---	---	---

Precedence

Operator precedence is the order in which operators are evaluated in a compound expression. Operators that have the same precedence are evaluated from left to right.

::	Scope resolution	class_name :: member
::	Global	::name
.	Member selection	object.member
→	Member selection	pointer → member
[]	Subscripting	pointer [expr]
()	Function call	expr (expr_list)
()	value construction	type (expr_list)

++ --	Post increment Post decrement	lvalue ++ lvalue --
sizeof sizeof ++ --	Size of object Size of type Pre increment Pre decrement	sizeof expr sizeof (type) ++lvalue -- lvalue
~ ! - + & * new delete delete [] ()	Complement Not Unary minus Unary plus Address of Dereference Create (allocate) Destroy (de-allocate) Destroy array Cast (type conversion)	~ expr ! expr - expr + expr & lvalue * expr new type delete pointer delete [] pointer (type)expr
* ->*	Member selection Member selection	Object .* pointer-to-pointer Pointer -> *pointer-to-pointer
* / %	Multiply Divide Modulo(remainder)	expr * expr expr / expr expr % expr
+ -	Add (plus) Subtract (minus)	expr + expr expr - expr
<< >>	Shift left Shift right	expr << expr expr >> expr
< <= > >=	Less than Less than or equal Less than or equal Greater than or equal	expr < expr expr <= expr expr > expr expr >= expr
== !=	Equal Not equal	expr == expr expr != expr
&	Bitwise AND	expr & expr
^	Bitwise exclusive OR	expr ^ expr
	Bitwise exclusive OR	expr expr
&&	Logical AND	expr && expr
	Logical inclusive OR	expr expr
? :	Conditional expression	expr ? expr :expr
= *= /= %= += -= <<= >>= &= = ^=	Simple assignment Multiply and assign Divide and assign Modulo and assign Add and assign Subtract and assign Shift left and assign Shift right and assign AND and assign Inclusive OR and assign Exclusive OR and assign	lvalue = expr lvalue *= expr lvalue /= expr lvalue %= expr lvalue += expr lvalue -= expr lvalue <<= expr lvalue >>= expr lvalue &= expr lvalue = expr lvalue ^= expr
Throw	Throw exception	throw expr
,	Comma (sequencing)	expr, expr

Reserved Words

The following identifiers are reserved for use as keywords, and may not be used otherwise:

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

In addition, identifiers containing a double underscore(`__`) are reserved for use by C++ implementations and standard libraries and should be avoided by users.

The ASCII representation of C++ programs uses the following characters as operators or for punctuation:

!	%	^	&	*	()	-	+	=	{	}		~
[]	\	;	'	:	"	<	>	?	,	.	/	

And the following character combinations are used as operators:

→	++	—	*	*→	<<	>>	<=	>=	==	!=	&&
	*=	/=	%=	+=	-=	<<=	>>=	&=	^=	=	::

Each is a single token.

In addition, the processor uses the following tokens:

##

Type Conversion

Converting one predefined type into another typically will change size and/or interpretation properties of the type but not the underlying bit pattern. The size may widen or narrow, and of course the interpretation will change.

There are two ways of type conversion:

Implicit type conversion

Using assignment statement. For example,

```
long lval = 3.14159;  
int i = lval; // i = 3
```

However, such type of type conversion is restricted to pre-specified type conversions only. Such type conversion sometimes result in surprising results, for example, `int i = 2/3` will assign a zero to `i`.

Explicit type conversion

The notation used will be.

```
type(expr)  
(type)expr
```

These are referred to as `typecast`. For example,

int(3.14159) results in 3

The if statement

The syntax of the IF statement is as follows:

```
if (expression)  
Statement;
```

The expression must be enclosed in parenthesis. The statement may be a compound statement. For example,

```
if (x > 5)  
x = 0;
```

The statement will be executed only if the expression is true. The syntax of the If-else statement is as follows:

```
if (expression)  
    Statement-1;  
else  
    Statement-2;
```

If expression is true statement-1 is executed. If expression is false, statement-2 is executed. So, depending on the truth-value of expression, either statement-1 or statement-2 is executed. For example,

```
if (x > 5)  
    ++y;  
else  
    -- z;
```

The statement 1 and 2 may be another if statement if the need so be. The if-else statement introduces a source of potential ambiguity referred to as the dangling-else. The problem occurs, when a statement contains more if-else clauses. The question is "with which if does the additional else clause properly match up? Consider,

```
if (row < 5)  
    if (col < 10)  
        cout << "valid";  
else  
    cout << "Invalid order"
```

The indentation indicates the programmer's belief that the 'else' is associated with outer 'if'. But, it is wrong. To avoid this ambiguity, a rule has been made that an else is associated with the last unmatched 'if'. So, the above statements will be executed in accordance with the following indentation;

```
if (row < 5)  
    if (col < 10)  
        cout << "valid";  
    else  
        cout << "Invalid";
```

The Switch-Statement

The syntax of the switch statement is as follows:

```
switch(expression) {  
    case constant-1  
        statement;  
    case constant-2  
        statement;
```

```

case statement-n
    statement;
default: statement
} // end of switch

```

The expression can be any valid expression except a floating-point value. The value of expression is compared with each constant for a match. The statement corresponding to matched case and the statements of the following cases are executed. If the value of expression is unmatched with all of the constant values, then the statement corresponding to default clause is executed. For example,

```

x = 5;
switch(x)
{
case 1 : cout << "one"; break;
case 5 : cout << "five"; break;
case 8 : cout << "eight"; break;
default : cout << "matching did not occur";
}

```

So, the output will be:

five

In the absence of the break statement, the output will be:

five eight matching did not occur

The WHILE statement

The syntax of the while statement is of the following form:

```

while (expression)
    statement;

```

So, whenever expression becomes true, the statement will be executed.

The FOR statement

The syntactic form of the FOR loop is as follows:

```

for (initialization; expression-1; expression-2)
    Statement(s);

```

The order of the evaluation is as follows:

```

initialization
expression-1
if expression is true {
    statement(s);
    expression-2;
}
else
    exit loop;

```

Both initialization and expression-2 can be NULL statements.

Examples:

```

for (int k=0; k<5; ++k)

```

```
For (;value>2; ++count)
```

```
For (; colour=green;)
```

Expression-1 must always evaluate to either 1 or 0. Only when expression is true, the statement(s) is/are executed.

The do while statement

The syntactic form of the do while loop is as follows:

```
do  
    Statement(s)  
while(expression);
```

The order of evaluation is as follows:

```
Statement(s)
```

Expression: If the expression is true then the statement(s) will be executed again.

So, the statement is executed before the expression is evaluated. The expression will always evaluate to True (1) or False (0). So, unlike other loop structures, the body is always executed atleast once.

The break Statement

The break statement will transfer control to the first statement after the body of the latest FOR, WHILE, DO or SWITCH in which it is present. For example,

```
while (1)  
{  
    cin>>height;  
    if (height>10000)  
        break;  
}  
cout << "Reduce Altitude";
```

The Continue Statement

The continue statement will terminate the current iteration of the WHILE, FOR or DO loop statement.

```
for(i=0; i<st-strength; ++i)  
{  
    if (marks[(i+1)]>=50)  
        continue;  
    else  
        // communicate to the student that  
        // he has failed  
}
```

The GOTO statement

The goto statement is an unconditional jump statement. The syntax of the goto statement is as follows:

```
goto Label;
```

Both the goto and Label should appear in the same function. A colon should always follow the Label. For example,

```
void matrix(int i)
```

```
if(i = 1)
goto Multiply;
```

```
Multiply;
}
```

Constraints on GOTO statement

There should be atleast one statement followed by LABEL. For example,

```
{goto X;
.
.
.
X ; // a Null statement is used
}
```

Between the goto and Label statements, there should be no explicit or implicit initialise statements. The constraint is applicable only in the case of forward jumps. In the case of backward jumps, this rule is not applicable.

However, the rule regarding forward jump is not applicable, if the GOTO statement jumps on the entire block containing the initialise statement.

Example:

```
goto Process
{
i=0;
j=k;
```

```
Process
```

A good programming practice is to avoid goto statement in a program.

Check Your Progress 2

1) What are the problems in the following program segment?

```
main ( )
{
int i, j, k;
float x;
x = i;
if (i > x) goto mult;
else
j = i;
Mult
k = i * j;
}
```

2) What will be the value of following expressions of C++ for the values of int a = 5, b = 6, c = 7, d = 8

- $a + b * c \% d$
- $a++ * b++ * a++ * b++$
- $++a * b-- * a++ * --b$
- $a \& b$
- $c | d$
- $\text{sizeof } a$

- 3) How many times the following loops will be executed for the values of `int i = 7, j = 250, k = 55`
- (a) `int count = 0;`
`for (i < k; i += 5)`
`{printf ("%d", count)`
`count++`
`}`
- (b) `do {i++;`
`count ++;`
`} while (j > i).`

1.4 FUNCTIONS AND FILES

In this section we will mainly discuss how to make our own library of functions which will be used frequently.

1.4.1 How to make a Library?

Let us assume that you want to have a function, which will receive parameters of type `double`, computes their sum and returns that sum which is of type `double`.

Also, let us have a function, which will receive an integer parameter and returns a 0 (false) if it is odd or a 1 (true) if it is even. Let the names of above functions be `sum` and `even`. Let the files in which they are placed are `sum.c` and `even.c`.

Let us make a library with these two files and let the name of library be `lib.a`. Now, applying the following sequence of steps in UNIX environment can make the library `lib.a`:

```
$cc -c sum.c even.c // The result is the equivalent object files
$ar cr lib.a sum.o even.o // An archive called lib.a is made which
// contains the object
// codes of two functions.
$ranlib lib.a // Now, the library is indexed for faster access
```

In the above code, `$` is the system prompt

Now, the library can be used alongwith any program say, `bill.c` in the following way:

```
$cc bill.c lib.a
```

The advantage of using a library is that only the functions, which are needed, will be used and the linker will look after it. If we do not archive them into a library, we have to specify the files individually, which will be cumbersome as well as error-prone.

1.4.2 Functions

In C++, any function has to be declared and defined before it is called.

The format of a function declaration is as follows:

```
Return-type functionname (Argument-type1, Arguement-type2.....);
```

For example,

```
void swap(int*, int*);
```

The format of a function definition is as follows:

```
return-type    function-name(Argument-type1, Argument-type2.....)
{
//function body
}
```

For example,

```
void swap(int* u, int* v)
```

//Program to swap the values of two integers u and v using third location temp.

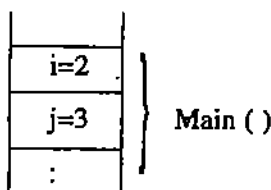
```
{
    int temp;
    temp = *u;
    *u = *v;
    *v = temp;
}
```

Arguments to a function can be passed by value or by reference.

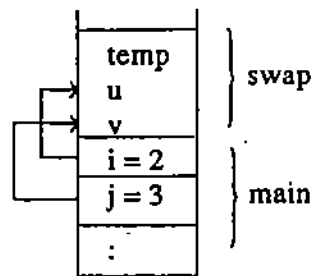
Let us demonstrate the call by value mechanism with a simple example as follows:

```
# include <iostream.h>
//swap function prototype
void swap(int, int);
void main( )
{ int i = 2;
  int j = 3;
  swap(i, j);
  cout << i << j;
}
//swap function implemented here
void swap (int u, int v)
{   int temp = u;
    u = v;
    v = temp;
    return
}
```

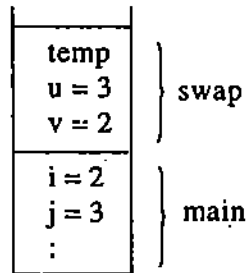
The output will be 2 (the value of i) & 3 (the value of j) that is no exchange has taken place in original values although the values of u & v must have changed. The reason is, in call by value, temporary storage will be allocated to arguments parameters.



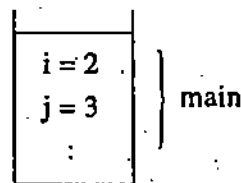
Start of Main before swap statement execution



On call to Swap function i is passed to u and j is passed to v



On complete execution of swap but before return.



On return from the call the values in main() remains unchanged.

The values of i & j in main() remain the same since the memory in which they reside is different from the storage locations where they are passed as arguments. This is the reason why the same values are reflected in the output.

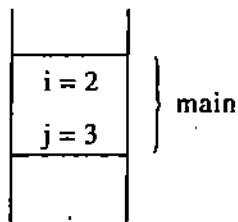
Consider the following example for demonstration of call by reference.

```
# include <iostream.h>
void swap (int&, int&);
//indicating call by reference
void main ( )
{
    int i = 2;
    int j = 3;
    swap (i, j);
    cout << "i =" << i << " j" << j
}

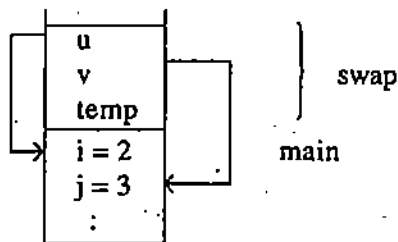
void swap (int& u, int& v)
{
    int w=u;
    u = v;
    v = w;
}
```

The output will be i = 3 j=2. That is the interchange in the main has occurred which was the desired purpose.

The reason is that during the compilation and execution of main(), the locations i & j are created and they were assigned the values 2 & 3 as follows:

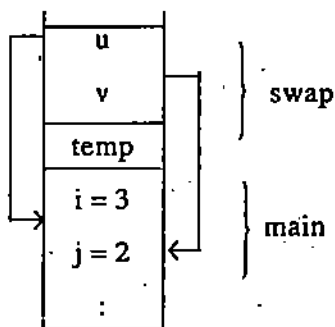


When there is a call to swap, the control will be transferred to procedure swap. Since the argument passing is by call by reference, the memory location u & v will refer to i & j. It can be illustrated as shown in the following figure:



u is a pointer/alias to i and v is alias to j.

After the completion of execution of procedure swap, the control passes to the main () program and the location of i & j will contain 3 & 2.



On interchange, u & v will change the values of i & j.

The locations in all the above situations are the same. In this way, the output will be i=3 j=2 in call by reference.

We can declare an argument as **const** if we want that it should not be modified by the called function. It is also possible to change the type of the argument. For example,

```
int check(const int& u)
{
    if u > 0
        return 1;
    else return 0;}

```

At the same time, if there is need for any type conversion between a formal parameter & an actual parameter then the argument must be declared **const** to the reference argument. It cannot be converted if it is not declared **const**.

Arrays can be passed as arguments to a function in C++ as in C.

Overloading means using the same name for different operators on different types. For example, '+' is an overloaded operator in most of the languages since '+' is the sign used for addition of two integers as well as two floating point numbers. The same is the case with '-' operator.

Function names can also be overloaded in C++ as demonstrated by an example as follows:

```
void search (const char*);
```



```
void search (int*);

# include <iostream.h>
void main( )
{
    int j[5];
    char k[5];
    // code for reading elements
    search (j); // second function is used
    search (k); // first function is used
}
```

When a function name is overloaded, the overloaded functions should have different number of arguments or different types of arguments. If not, the compiler cannot take a decision regarding which function to use.

Type conversion will take place if the matching of arguments does not take place and in such cases, error messages will result. We can also have unspecified number of arguments by using `va_arg`, `va_end`, `va_list`, `va_start` of library `<stdarg.h>`. We can have pointers to a function as in C language.

1.4.3 Macros

A Macro can be defined as a segment of code with a name which replaces the occurrences of name in the code. For example:

```
# define Msoft Bill Gates
```

So, in the program, whenever the token "Msoft" is encountered, "Bill Gates" replaces it.

Macros can be used effectively for defining a symbolic name to a constant. It improves the readability of the program. However, the same work can be done using constant declarations, for example:

Macros	Constant Variable Definition
Example <pre># define PI 3.14159 # define MAXSIZE 5000</pre>	Example <pre>const PI = 3.14159; const MAXSIZE = 5000;</pre>
Advantage the symbol PI will be taken as value and no space will be reserved by compiler for them. But symbols cannot be identified to a type	Symbols declared here can be identified to a type. Thus, helps in identifying compilation errors.

The capabilities of macros is far beyond just symbols. However, they should be used with cautions.

- There will be problems with macros when there were recursive calls.
- Another problem is with precedence.

For example,

```
#define cube(a) a × a × a
```

Now, let us assume that there is a statement in the program as follows:

```
int k = cube (k + 3);
```

Now, this statement will be expanded as follows:

```
int k = (k + 3 * k + 3 * k + 3)
```

which is certainly not $(k + 3)^3$ as per precedence of operator.

Due to these and many other reasons, C++ provides `const`, and template mechanisms so that the use of Macros can be minimised.

Check Your Progress 3

- 1) What is overloading of functions? How does compiler resolve which of the overloaded function has been called?

.....

.....

.....

- 2) Implement SWAP functions using call by value and call by reference and print intermediate results to check the justification given in the section above.

.....

.....

.....

- 3) Write a macro to define a max function for two variable.

.....

.....

.....

1.5 SUMMARY

In this unit, we have introduced various programming paradigms. Object Oriented Programming has become popular since the paradigm perfectly fits the real life situations.

We have also discussed various control statements namely- IF THEN, IF THEN ELSE, SWITCH, WHILE, DO, FOR and the circumstances when we have to prefer a particular statement through one can be expressed in the form of other.

We are also able to know about the declaration and use of functions. A function can have more than one return statement.

We have seen the method of defining macros and a few disadvantages of them. We will explore the "Inline functions" in the remaining sections of the block.

In the next unit, we will also describe the Declaration and usages of classes, concept of overloading, inheritance and also how C++ supports Object Oriented Programming.

1.6 MODEL ANSWERS

Check Your Progress 1

- 1) Procedural Programming, Modular Programming, Object Oriented Programming
- 2) (a) & (c)
- 3) Part.

Check Your Progress 2

- 1)
 - no return type to main ()
 - an assignment & comparison of dissimilar type not necessarily will give syntax error as C++ as it is not very strong typed language
 - use of goto is not recommended.
- 2) Write and run a small C++ program and compare your theoretical results with those obtained from program.
- 3) Write and run a small C++ program and compare your theoretical results with those obtained from program.

Check Your Progress 3

- 1) More than one function with the same name but different number of or type of arguments. The different number of arguments or type of arguments is used by compiler for resolving which function has been actually called.
- 2) Do it through a program and check results with different set of data.
- 3) # define max(a,b)(a > b)? a : b

1.7 FURTHER READINGS

- 1) B. Stroustrup, The C++ Programming Language, Third edition, Pearson/ Addison-Wesley Publication.
- 2) N. Barkakati, Object Oriented Programming in C++, Prentice Hall of India.

UNIT 2 CLASSES AND OBJECTS

Structure

- 2.0 Introduction
- 2.1 Objectives
- 2.2 Definition and Declaration of a Class
- 2.3 Scope Resolution Operation
- 2.4 Private and Public Member Functions
- 2.5 Creating Objects
- 2.6 Accessing Class Data Members and Member Functions
- 2.7 Arrays of Objects
- 2.8 Objects as Function Arguments
- 2.9 Summary
- 2.10 Model Answers

2.0 INTRODUCTION

In the previous unit we discussed different programming paradigms as well as the syntax of various C++ statements. We also learnt how to make our own library. In this unit we will discuss Class, as important Data Structure of C++. A Class is the backbone of Object Oriented Computing. It is an abstract data type. We can declare and define data as well as functions in a class. An object is a replica of the class to the exception that it has its own name. A class is a data type and an object is a variable of that type.

2.1 OBJECTIVES

At the end of this unit, you will be able to:

- Define the basic concept of a class
- Explain private and public clauses
- Create objects
- Write programs with functions taking objects as arguments.

2.2 DEFINITION AND DECLARATION OF A CLASS

A class is a user-defined type. The definition of a class includes declaring a data object of that type, specifying the data items as well as functions which operate on these data items, indicating if the data can be accessed by functions which are out of scope of class by indicating "public or private" and other details.

We shall consider the class "Queue" as an example:

```
class Queue
{
    int front, rear;
    int queue_array[10];
public:
    int isqueueempty( );
    int isqueuefull( );
```

```
void insert(int);  
void delete( );  
void print( );  
};
```

In the above Class front, rear, queue_array are data members. Whenever, we define and declare a class, all the data as well as member functions are "private" by default.

"Private" means that they can be accessed only by the functions within the class. These functions are known as **Member functions**.

"Public" means that they can be accessed by the functions which are outside the class also. Since, everything in a class is Private by default, we can specifically indicate that something is public as we have done in the case of above member functions.

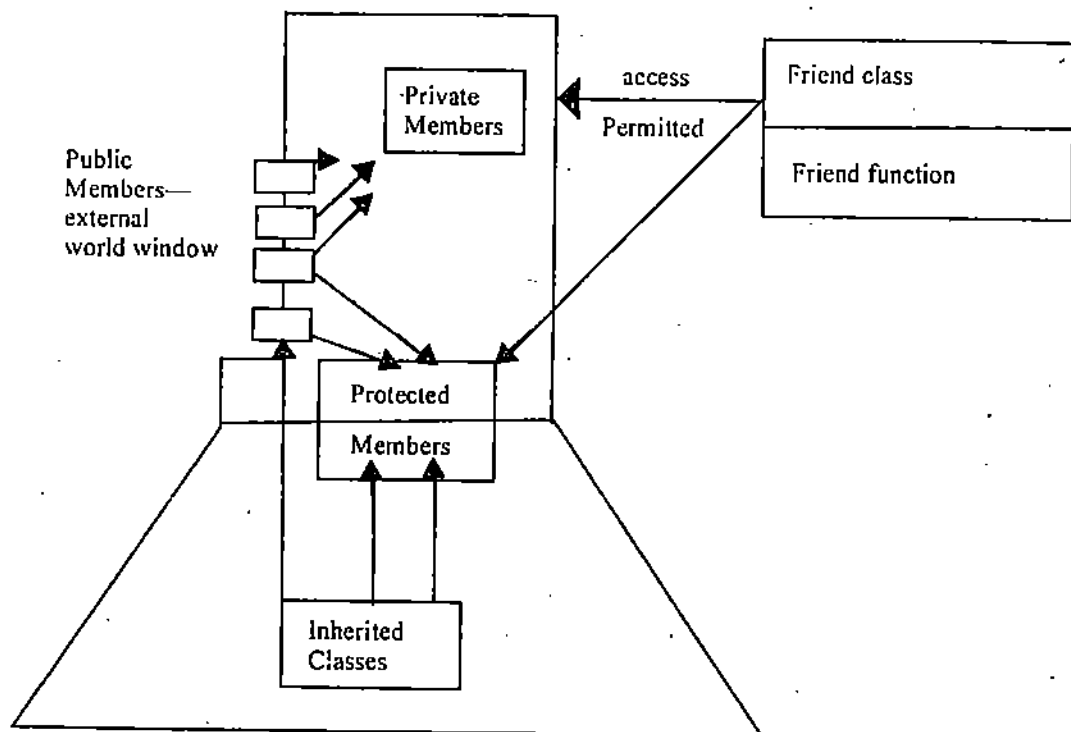


Figure 1 : Public and Private Members of a class

So, in the above example, the Private members front, rear, queue_array can be accessed only by the five member functions. (Any way we can override this. We will discuss about it later).

The ability to declare the members of a class as private gives rise to the concept of "data hiding".

Encapsulation is the mechanism that binds together code and the data. It manipulates and keeps both safe from outside interference and misuse. The concept of class in C++ supports encapsulation by enabling both data as well as functions to be declared and defined in the same class and declaration of all members as "private" by default.

We can declare arrays within a class just like any other data item. Queue_array in the class "Queue" is an example.

2.3 SCOPE RESOLUTION OPERATION

The scope resolution operator is denoted by "::".

A class is identified by the external world through the public member functions, constructor and destructors, which is also referred to as the interface of a class. The interface of class, in general, needs to specify the names of member functions and the parameters to be passed to values to be returned. Therefore, in general the interface of the class is kept separately from the implementation of its member functions. One of the key reason for that is, a user of a class need not know the implementation details, it just need to know how a class can be used.

Since, implementation/definition may be defined separately, therefore, we need a mechanism to identify with what class a function implementation is associated. The scope resolution operator specify the scope of a function that is to what class it is associated.

For example, any function of the Queue class in Section 2.2 may be implemented as:

```
Queue::isqueuempty( )
{
    ...
    ...
}
```

The scope resolution operator can also be used for defining the global variables also.

Please note here that the functions can also be defined inline that is during the class definition itself.

Consider the following example:

```
# include <iostream.h>
int x;
void main( )
{
    f ( );
}
void f ( )
{
    int x; // hides the Global x
    x = 1; // assigns to local x
    // if we want to make any assignments to global x in f ( ), it is not
    // possible
}
```

But with the help of scope resolution operator "::", we can do that as follows:

```
void f ( )
{
    int x;
    x = 1; // assigns to local x
    :: x = 2; // assigns to global x
}
```

So, whenever a variable is prefixed by ::, it refers to Global variable.

So, with the help of Scope Resolution operator, we can use a hidden global name.

2.4 PRIVATE AND PUBLIC MEMBER FUNCTIONS

All the member functions, which are declared in a class, are private by default unless specifically indicated as public (refer to Queue class where all the member functions are declared public).

Other member functions of the same class and friends of it can only call all private functions. Private functions, by themselves cannot be invoked. (Refer to figure 1). For example,

```
class one
{
    int x, y;
    one()
    //one () is a private function by default.
    {
        x = 1;
        y = 2;
    }
    void print()
    {
        cout << x << y;
    }
};
```

```
one x1; //wrong
        // the procedure one() cannot be executed as it is private.
```

If we change it to public, then it will be executed.

Consider the following:

```
class one
{
    int x, y;
    void print()
    {
        cout << x << y;
    }

public:
    one()
    {
        x = 1;
        y = 2;
    }
};

one x1;
void main()
{....}
```

Now, one() will be executed. But, any call such as x1.print() is invalid as it is a private member function. So, we can invoke it from some other member function of the same class follows:

```
class one
{
    int x, y;
    void print()
    {
```

```
cout << x << y;
}
public:
one( )
{
x= 1;
y = 2;
print ( ); // valid call to print within one( )
}
};
one x1;
void main( )
{
}
```

Also, a private member function can be called from another private member function. Anyway, the first member function, which starts the chain should be a public member function.

Check Your Progress 1

- 1) Define a string data type with the following functionality:
- A constructor having no parameters
 - Constructors which initialize strings as follows:
 - A constructor that creates a string of specific size
 - Constructor that initializes using a pointer string
 - A copy constructor
 - Define the destructor for the class
 - It has overloaded operators (This part of question will be taken up in the later units).
 - There is operation for finding length of the string.

2.5 CREATING OBJECTS

If a class has a constructor, then it is called whenever an object of that class is created otherwise a default constructor for the class is called. If the class has the destructor, then it is called whenever an object of that class is to be destroyed otherwise a default destructor for the class is called.

We can create objects in the following ways. We can create an automatic object, which is created each time its declaration is encountered during the execution of the program and is destroyed each time the block in which it occurs is left. For example,

```
class example1
{
    .
    .
    .
};
void main( )
{
    f()
}

void f()
{
    example1 ex1;
    .
    .
    .
}
```

Now, in the above example, the object ex1 is created whenever the control is transferred to f() and the statement declaring ex1 is executed. ex1 is destroyed, once the execution of f() is complete. Here ex1 is automatic object. Please note that in this example no constructor or destructor of the object is defined, therefore, it will use default constructor & destructor generated by the compiler itself.

We can create a static object, which is created once during the start of program and is destroyed once the execution of program is completed.

Consider the following example:

```
class example
{
    .
    .
    //
    .
    .
};
void main()
{
    f();
    f();
    // last statement
}
```

```

void f()
{
static example ex1;
.
.
.
}

```

Now, ex1 is created once during the first call to f(). It is not destroyed after the completion of execution of f() since it is a **static object**. Also, since it is not created once again, the data stored in it which was present at the end of previous call is not lost when f() is called again. It is destroyed only once and that is done after the completion of execution of the program. In this case object still be created by default constructor, however, it will be run only once during the lifetime of the object.

We can create an object on the free store using new operator and destroy it using delete operator. Consider the following example:

Class example1

```

{
.
.
.
};
main()
{
example1*p= new example1;
delete p;
}

```

P is a pointer to an object of type example1. The object as well as the pointer, which points to it, is destroyed when delete is executed. A user can design his own new and delete operators.

An object can also be created as a member of another class. Consider the following example:

Class example1

```

{
.
.
.
}

```

Class example 2

```

{
example1 ex1; // ex1 is an object which is member of another class
.
.
.
};

```

Whenever an object of example2 is created and if there were constructors for both or either without taking arguments, then the constructor of object ex1 is executed followed by the execution of constructor of example2.

If there were arguments to one or both the constructors, then they have to be explicitly called. The same is case with destruction. Whenever an object of example2 has to be destroyed, the object ex1 is destroyed followed by the destruction of object of example2.

An object can also be created as an array element. Consider the following example:

```
Class ex
{
.
.
.
}
```

```
ex ten[10];
```

If the ex contains constructors, then there should be a default constructor, which need not be called with a parameter. The reason is, there is no way to specify an argument along with an array.

2.6 ACCESSING CLASS DATA MEMBERS AND MEMBER FUNCTIONS

If a data member is public, then it can be accessed by the following syntax
objectname.datamember

If the data member is private then it can be accessed by the member functions only.

Any public member function can be accessed as follows:

```
object_name.member function_name
```

If the member function is private, then it cannot be accessed as mentioned for public member functions. A private member function can be accessed only through another member function. (Please refer to figure 1).

Consider the following example which illustrates the above concepts. The following is a linear search program which will receive a key value as well as a list of integers and gives the position of the key value in the list if it is present.

```
# include<iostream.h>
class data
{
    int list[10];
    int size;
    public:
    data () //constructor of class data
    {
        cout << "size of list is atmost 10" <<"Enter the size of list";
        cin >> size; //Accessing private data
        for(int i = 0; i<size; ++i)
        {
            cout << "Enter the element";
            cin >> list [i];
        }
    }
    // end of constructor
    void search(int p)
    {
```

```

int check = 1;
int i = 0;
while ((check) && (i < size))
//Check is not equal to 0 and i is less than size
{
    if (p == list[i])
        check = 0;
    else
        ++i;
}

if (check == 0) // key is found as check is assigned to zero value
    cout << "key matched at the following position" << ++ i;
else
    cout << "No matching element found";
}
}; // end of class data

void main()
{
    //Calls the constructor defined in the process
    data d1;
    int key;
    cout << "Enter the key value";
    cin >> key;
    d1.search(key); // Accessing member function. Notice the way
    member function has been
    // Called by the object d1 which is of class data.
}

```

2.7 ARRAYS OF OBJECTS

Consider the following class:

```
class C1
```

```
{
```

```
};
```

Now, an Array of objects of C1 can be declared as C1 c [10];

So, we have declared 10 objects of type C1. Also, we can create an array of objects C1 on the free store as follows:

```
C1 c = new C1[10];
```

If C1 is having a constructor, then it should have a default argument or no arguments. During the execution of above statement, each constructor belonging to each object is executed in sequence one after another.

The way of accessing data members or member functions of the objects are similar to the method we have outlined in the previous sections except that the prefix of data member as well as member function name will be Array_name[index].

So, if you are accessing data item (say, i assuming that i is public) of object c[3], then it is indicated as c[3].i

When we are creating objects on free store; we can delete them as follows:

```
Delete[] c; // Since c is an Array
```

```
Delete X; // if X is a single object.
```

2.8 OBJECTS AS FUNCTION ARGUMENTS

We can pass objects as arguments to a function. We can send them by reference or by value.

Consider the problems of adding two matrices:

This can be done by sending objects by reference as follows:

```
#include <iostream.h>
#define MAXROW 50
#define MAXCOL 50
int row, col;
class matrix
{
int m[MAXROW][ MAXCOL];
public
    void in_element(void);
    void print(void);

friend matrix add(const matrix&, const matrix&)
//The addition will be carried out by a friend function.
//This function however will not change the value of its arguments,
//hence const been used before each argument.
};

void matrix::in_element( ) //please note use of scope resolution operator
{
    cout << "the matrix is:";
    for (int i = 0; i < row; ++ i)
        for(int j = 0; j< col; ++ j)
            cin >> m[i][j];
}

void matrix::print( )
{
    cout << "the matrix is:";
    for (int i = 0; i < row; ++i)
    {
        cout << "\n";
        for (int j = 0; j < col; ++j) //Print the matrix in tabular form.
            cout << m[i][j] << "\t";
    }
}
```

In the function prototype, `matrix add(const matrix&, const matrix&);` the `matrix&` is a reference to matrix class. Please note this represent call by reference and not call by value. The advantage of call by reference here is that since the matrix is a large object, its instance will not be duplicated as is the case in call by value. Moreover, `const` ensures that original values of matrix does not get modified even by mistake in programming. Such mistakes will be caught by compiler. Thus, this method of parameter passing may be considered for large objects.

```
matrix first, second;
void main ( )
{
    cout << "enter the order of the matrix:";
    cin >> row;
    cin >> col;
```

```

    first.in_element ( );
    second.in_element ( );
    matrix result = add (first, second);
    result.print ( )
}

```

```

matrix add(const matrix& one, const matrix& two)
{
    matrix temp_result;
    for(int k = 0; k < row; ++k)
        for (int l = 0; l < col; ++l)
            temp_result.m[k][l] = one.m[k][l] + two.m[k][l];
    return temp_result;
}

```

The same can be done by sending objects by value as follows:

//demonstration of call of objects by value

```

#include <iostream.h>
#define MAXROW 50
#define MAXCOL 50
int row, col;
class matrix
{
    int m[MAXROW][ MAXCOL];
public:
    void in_element(void);
    void print(void);

friend matrix add(const matrix&, const matrix&);
};

void matrix::print( )
{
    cout << "the matrix is:";
    for (int i = 0; i < row; ++i)
    {
        cout << "\n";
        for (int j = 0; j < col; ++j)
            cout << m[i][j] << " ";
    }
}

void matrix::print( )
{
    cout << "the matrix is:";
    for (int i = 0; i < row; ++i)
    {
        cout << "\n";
        for (int j = 0; j < col; ++j)
            cout << m[i][j] << " ";
    }
}

matrix add(matrix, matrix);
matrix first, second;
void main( )
{
    cout << "enter the order of the matrix :";
    cin >> row;
    cin >> col;
    first.in_element( );
    second.in_element( );
}

```

```
matrix result = add(first, second);  
result.print();  
}  
  
matrix add(matrix one, matrix two)  
{  
    matrix temp_result;  
    for (int k = 0; k < row; ++k)  
        for (int l = 0; l < col; ++l)  
            temp_result.m[k][l] = one.m[k][l] + tow.m[k][l];  
return temp_result;  
}
```

Check Your Progress 2

- 1) Write a program for the implementation of stack using classes.

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

- 2) Write a program for the multiplication of matrix of order $m \times n$ with a vector of order $m \times 1$ using classes. Design a function, which accepts matrix and vector as arguments and returns the resultant matrix.

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

2.9 SUMMARY

In this unit, we discussed the concept of class, its declaration and definition. It also contains the ways for creating objects, accessing the data members of the class. We have seen the way to pass objects as arguments to the functions with call by value and call by reference.

2.10 MODEL ANSWERS

Check Your Progress 1

```
1) const maxlen = 20;  
class string  
{ private:  
    char * str;  
    int length;  
Public:  
    // constructors  
    string (); // defines a string of length 0 but of a default size 20.  
    string (int i_len) // create a blank string of size i_len  
    string (const char *s)  
    int strlen (const string &s)  
    //:other string functions  
    ~string (); // destructor
```

```

};

string :: string ( )
{
    str = new char [maxlen];
    length = 0;
    str [0] = '\0';
}

string :: string (int i_len)
{
    length = i_len;
    str = new char [i_len];
    int i =0;
    for (i =0; i<i_len; i++) str [i] = ' ';
    str[i] = '\0';
}

string ::string (const char *s)
//constructor with initialization using a constant string
{
    length = strlen(s);
    str = new char [length + 1];
    strcpy (str, s);
} // strlen & strcpy one library string functions

string :: string (const ' string & s)' // copy constructor
{
    length = s.length
    str = new char (s.length +1)
    strcpy (str, s.str)
} // create a new instance of a string

string :: length (void) const
{
    return length;
}

string :: ~string ( )
{
    delete str;
}

```

Check Your Progress 2

- 1) **Hint:** Use array as data structure for implementing the stack and for pushing and popping elements on the stack.
- 2) **Hint:** The following should be provided in your program:
 - A Class for Matrix
 - A Class for Vector (One dimensional array)
 - A Function that accepts, Matrix and Vector as arguments and returns Matrix or Vector as per the type of expected result.

UNIT 3 OPERATOR OVERLOADING

Structure

- 3.0 Introduction
- 3.1 Objectives
- 3.2 Operator Functions
- 3.3 Large Objects
- 3.4 Assignment and Initialisation
- 3.5 Subscripting
- 3.6 Function Call
- 3.7 Increment
- 3.8 Decrement Operator
- 3.9 Friends
- 3.10 Summary
- 3.11 Model Answers

3.0 INTRODUCTION

In the previous unit we discussed concept of a Class. In this unit, we shall discuss the concept of overloading. For example, to add two matrices, you need to define a Matrix class. We will create two objects of this class. Now, when we have to add these two matrices, we have to access the data of their objects through member functions and add them. But, this is cumbersome. It will be convenient if I can add the two objects directly and the other details can be taken care of compiler. But, the '+' operator is meant for addition of data types like integers etc. So, to add two objects with '+' operator, in C++, there is a facility to define a function with name '+' which accepts two objects as arguments. Then, we write the code in that procedure, which adds the data of these two objects and returns the object. This concept is known as operator overloading since single operator can be used for more than one purpose.

3.1 OBJECTIVES

After going through this unit, you should be able to:

- write functions with operator overloading features and
- use friends clause.

3.2 OPERATOR FUNCTIONS

The meanings of the following operator can be redefined using functions:

+	-	*	/	%	^	&		~	!
=	>	<	+=	-=	*=	/=	%=	=	&=
!=	<<	>>	>>=	<<=	==	!=	<=	>=	&&
	++	--	?*	,	?	[]	()	New	Delete

Though the meanings are redefined, their precedence cannot be changed. At the same time, a Unary operator cannot be redefined as a Binary Operator.

The Syntax of declaration of an Operator function is as follows:

Operator Operator_name

For example, suppose that we want to declare an Operator function for '=='. We can do it as follows:

```
operator =
```

A Binary Operator can be defined by either a member function taking one argument or a global function taking two arguments. For a Binary Operator X, a X b can be interpreted as either a.operator X (b) or operator X (a, b).

For a Prefix unary operator Y, Ya can be interpreted as either a.operator Y () or Operator Y(a). For a Postfix unary operator Z, aZ can be interpreted as either a.operator Z(int) or Operator (Z(a), int).

The operator functions namely operator=, operator[], operator () and operator? must be non-static member functions. Due to this their first operands will be lvalues.

An operator function should be either a member or take atleast one class object argument. The operators new and delete need not follow the rule. Also, an operator function, which needs to accept a basic type as its first argument, cannot be a member function. Some examples of declarations of operator functions are given below:

```
class C
{
    C operator ++ (int); // Postfix increment
    C operator ++( ); // Prefix increment
    C operator || (C); // Binary OR
}
```

Some examples of Global Operator functions are given below:

```
C operator - (C); // Prefix unary minus
C operator - (C, C); // Binary "minus"
C operator - - (C&, int); // Postfix Decrement
```

We can declare these Global Operator Functions as being friends of any other class.

Examples of operator overloading:

Operator overloading using friend.

```
Class complex
{
    int real;
    int imag;
    public;
    friend complex operator + (const complex & x, const complex & y);
                                //operator overloading using friend
    complex ( ) {real = imag = 0;}
    complex (int x, int y) { real = x; imag = y;}
}

complex operator + (const complex & x, const complex & y)
{
    complex z;

    z.real = x.real + y.real;
    z.imag = x.imag + y.imag;
    return z;
}
```

```

    }
    main () {
        Complex x, y, z;
        x = complex (5,6);
        y = complex (7,8);
        z = complex (9,10);
        z = x + y;    // addition using friend function +
    }

```

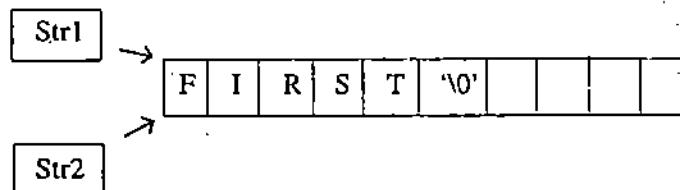
Operator overloading using member function:

```

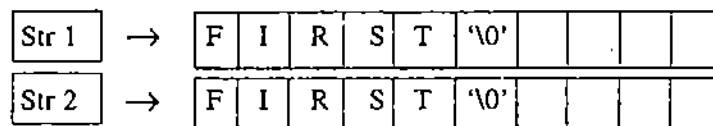
Class string
{
    char * str
    int    length; // Present length of the string.
    int    max_len;    //(maximum space allocated to string)
public:
    string ();        // blank string of length 0 of maximum allowed length
                    // of size 10.
    String (const  string & s)    // copy constructor
    ~ string () {delete str;}
    int operator == (const string & s) const; // check for equality
    string & operator = (const string & s)
    // overloaded assignment operator
    friend string operator + (const string & s1, const string & s2)
    } // string concatenation
    string :: string ()
    {
        max_len = 10;
        str = new char [max_len];
        length = 0;
        str [0] = '\0';
    }
    string :: string (const string & s)
    {
        length = s.length;
        max_len = s.max_len
        str = new char [max_len];
        strcpy (str, s.str)    //physical copying in the new location
    }

```

[Comment: Please note the need of explicit copy constructor as we are using pointers. For example, if a string object containing string "first" is to be used to initialize a new string and if we do not use copy constructor then will cause:



That is two pointers pointing to one instance of allocated memory, this will create problem if we just want to modify the current value of one of the string only. Even destruction of one string will create problem. That is why we need to create separate space for the pointed strings as:



Thus, we have explicitly written the copy constructor. We have also written

the explicit destructor for the class. This will not be a problem if we do not use pointers.

```
String :: ~ string ( )
{
    delete str
}
string & string :: operator = (const string & s)
{
    if (this != &s) // if the left and right hand variables are different
    {
        length = s.length;
        max_len = s.max-len;
        delete str; // get rid of old memory space allocated to this string
        str = new char [max_len]; // create new locations
        strcpy (str, s.str); // copy the content using string copy function
    }
    return *this
}

```

// Please note the use of this operator which is a pointer to object that invokes the call to this assignment operator function.

```
inline int string :: operator == (const string & s) const
{
    // uses string comparison function
    return strcmp (str, s.str)
}
string string :: operator + (const string & s)
{
    string s3;
    s3.length = length + s.length
    s3.max_len = s3.length
    char * newstr = new char [len+1];
    strcpy (newstr, str);
    strcat (newstr, s.str);
    s3.str = newstr;
    return (s3)
}

```

Overloading << operator

To overload << operator, the following function may be used:

```
Ostream & operator << (ostream & s, const string & x)
{
    s << "The String is:" << x
    return s
}

```

You can write appropriate main function and use the above overloaded operators as shown in the complex number example.

3.3 LARGE OBJECTS

For all operator functions, which take classes as arguments, there is the overhead of copying entire object in the temporary storage. This can be avoided by declaring operator functions as taking reference arguments. For example, consider the following class matrix.

```
class Matrix
{ int m[10][10];
public :
Matrix ( );

```

```
friend Matrix operator + (const Matrix&, const Matrix&);  
friend Matrix operator * (const Matrix&, const Matrix&)
```

Pointer cannot be used because it is not possible to redefine the meaning of an operator when applied to a pointer. The reference type, thus, avoids copying of large objects.

3.4 ASSIGNMENT AND INITIALISATION

Consider the following class:

```
class Employee  
{  
    char name;  
    int ssno;  
public:  
    Employee( ) {name = new char[20];}  
    ~Employee( ) {delete[] name;}  
};  
  
int f( )  
{Employee E1, E2;  
    cin >> E1;  
    cin >> E2;  
    E1 = E2;  
}
```

Now, the problem is that after the execution of f(), destructors for E1 & E2 will be executed. Since both E1 & E2 point to the same storage, execution of destructor twice will lead to error as the storage being pointed by E1 & E2 were disposed off during the execution of destructor for E1 itself!

Defining assignment of strings as follows can solve this problem.

```
class Employee  
{  
public:  
    char name  
    int ssno;  
    Employee( ) {name = new char [20];}  
    ~Employee ( ) {delete [] name;}  
    Employee& operator = (const Employee&  
};  
Employee & Employee :: operator = const Employee & e  
{  
    if (this !=&e)  
    {  
        delete[] name;  
        name = new char [20];  
        strcpy(name, e.name);  
    }  
    return *this  
}
```

3.5 SUBSCRIPTING

An operator [] function can be used to give subscripts, a meaning for class objects. The second argument (the subscript) of an operator[] function may be of any type.

Consider the following example which demonstrates the use of operator[] function:

```
#include <iostream.h>
class item
{
    int i;
    public:
    item( )
        {
            cout<< "enter the number";
            cin>>i;
        }
    int operator[] (int);
};

int item :: operator[] (int j)
{
    if (i ==j)
        cout<< "it matches";
    else
        cout<< "it doesn't match"
    return 1;
}

void main( )
{
    item array[8];
    for (int i=0; i<8; ++i)
    {
        if (array[i][2])
            cout << "operator overloading";
    }
}
```

3.6 FUNCTION CALL

Function call which is written as Procedure_name (argument1, argument2,) can also be interpreted as a binary operation with procedure_name as the left operand and arguments as the right operand. The call operator() can be overloaded in the same way as other operators. An argument list for an operator() function is evaluated and checked according to the usual argument passing rules.

The following example demonstrate the operator() function:

```
#include < iostream.h>
#include <string.h>
class data
{
    char name[20];
    int index;
    public:
    data( )
    {
        cout << "Enter the name:";
        cin >> name;
    }
    void operator( ) (char);
};

void data :: operator( ) (char source)
{
    if(strcmp(source, name) ==0)
        cout << "Matching occurred";
}

void main( )
```

```
{
    data bank[10];
    char str[10];
    cout << "Enter the search string";
        cin >> str;
    for (int i=0; i<10; i++)
    {
        bank[i](str);
    }
}
```

3.7 INCREMENT

We can also overload “++” operator. Conventionally, since ++ can be used as postfix as well as prefix operator, we can have two different overloaded functions.

The following is an example program, which uses an operator function of ++ for prefix application:

```
#include<iostream.h>
class increment
{
    int i, j, k;
    public :
        increment( )
        {
            i =5;
            j = 6;
            k = 7;
        }

    void operator++( )
    {
        cout << (++i) << (++j) << (++k);
    }
};
void main( )
{
    increment in;
    ++in;
}
```

The following is an example of an operator function of ++ for postfix application.

```
#include<iostream.h>
class increment
{
    int i, j, k;
    public :
        increment( )
        {
            i =5;
            j = 6;
            k = 7;
        }

    void operator++(int )
```

```

    cout << (i++) << (j++) << (k++);
}
};
void main( )
{
increment in;
in++;
}

```

Please note the prototype of prefix and postfix ++ operator.

3.8 DECREMENT OPERATOR

We can also overload '-' operator. Conventionally, since '-' can be used as a prefix as well as postfix operator, we can have two different overloaded functions.

The following is an example of a program, which uses an operator function of -- for prefix application.

```

#include<iostream.h>
class decrement
{
    int i, j, k;
public :
    decrement( )
    {
        i = 5;
        j = 6;
        k = 7;
    }
void operator -- ( )
{
    cout << (-- i) << (-- j) << (-- k);
};
void main( )
{
decrement de;
-- de;
}

```

The following is an example of a program, which uses an operator function of --' for postfix application.

```

#include<iostream.h>
class decrement
{
    int i, j, k;
public :
    decrement( )
    {
        i = 5;
        j = 6;
        k = 7;
    }
void operator -- (int)

```



```
{
    cout << (i--) << (j--) << (k--);
}
};

void main( )
{
decrement de;
de --;
}
```

3.9 FRIENDS

As we discussed previously, any private data of a class can be accessed by only its member functions. But, any other function, which is not a part of class, can also access private data provided it was declared as a friend of the class whose private data is to be accessed.

For example,

```
Class x {
{
    int i,
    char j;
public:
        int modify( )
        {
            //
            .
            .
            .
            //
        }
}

friend void check( );
};

void check( )
{
    if (x.i < 0) // no error
    {
        //
        .
        .
        .
    }
}
```

Now, the function "check" is able to access the private data of class x since it has been declared as a friend of the class x. Similarly, we can also declare a member function of a class to be a friend of another class.

Check Your Progress

- 1) All operators of C++ can be overloaded. True False
- 2) There cannot be a member operator function, which receives other than _____ as its left argument.
- 3) We can define our own storage allocations by overloading _____ and _____.

- 4) The precedence of operators remains unchanged even if they are overloaded. True False
- 5) A Binary Operator function may be defined for a unary operator function. True False

3.10 SUMMARY

In this unit, we have seen how to overload operators. All the operators that can be overloaded were listed in 3.2. Even after writing operator overloaded functions, the precedence of operators remains unchanged. Also, an operator that is unary cannot be used as a Binary operator by overloading. A ' \rightarrow ' operator can be used as Postfix unary operator. The ' $++$ ' & ' $--$ ' operators can be used as Postfix or Prefix operators. So, separate functions overloading them for both the different applications have been shown. Finally, until the last unit, we are of a view that Private data of a class can be accessed only in member functions of that class. But, other functions, which are declared as "friend", can also access them.

3.11 MODEL ANSWERS

- 1) False
- 2) Object
- 3) New, Delete
- 4) True
- 5) False.

UNIT 4 INHERITANCE - EXTENDING CLASSES

Structure

- 4.0 Introduction
- 4.1 Objectives
- 4.2 Concept of Inheritance
- 4.3 Base Class and Derived Class
- 4.4 Visibility Modes
- 4.5 Single Inheritance
 - 4.5.1 Private Inheritance
 - 4.5.2 Public Inheritance
 - 4.5.3 Protected Inheritance
- 4.6 Multiple Inheritance
- 4.7 Nested Classes
- 4.8 Virtual Functions
- 4.9 Summary
- 4.10 Model Answers

4.0 INTRODUCTION

In this unit, you will go through the concept of inheritance using C++. Inheritance allows a class to include the members of other classes without repetition of members. There were three ways to inherit a class. They are public, private and protected inheritance. Public Inheritance means, "public parts of super class remain public and protected parts of super class remain protected". Private Inheritance means "Public and Protected Parts of Super Class remain Private in Sub-Class". Protected Inheritance means "Public and Protected Parts of Superclass remain protected in Subclass. We shall also deal with nested classes in this unit.

4.1 OBJECTIVES

After going through this unit, you should be able to:

- Describe the concepts of inheritance
- Apply inheritance concepts to real-life programs
- Define different types of inheritance.

4.2 CONCEPT OF INHERITANCE

Inheritance is a concept, which is the result of commonality between classes. Due to this mechanism, we need not repeat the declaration as well as member functions in a class if they are already present in another class.

For example, consider the classes namely "minister" and "prime-minister". Whatever information is present in minister, the same will be present in Prime Minister also. Apart from that there will be some extra information in class

Prime Minister due to the extra privileges enjoyed by him. Now, due to the mechanism of inheritance, it is enough only to indicate that information which is specific to prime minister in its class. In addition, the class prime minister will inherit the information of class minister.

4.3 BASE CLASS AND DERIVED CLASS

Let us take the classes Employee and class Manager. A Manager is an Employee with some additional information.

Now, when we are declaring the classes Employee and Manager without applying the concept of inheritance, they will look as follows:

```
class Employee
{
    public:
    char* name;
    int age;
    char* address;
    int salary;
    char* department;
    int id;
};
```

Now, the class Manager is as follows:

```
class Manager
{
    public:
    char* name;
    int age;
    char* address;
    int salary;
    char* department;
    int id;
    employee* team_members; //He heads a group of employees
    int level;              //his position in hierarchy of the organization
    .
    .
};
```

Now, without repeating the entire information of class Employee in class Manager, we can declare the Manager class as follows:

```
class Manager: public Employee
{
    public:
    Employee* Team_members;
    int level;
    .
    .
};
```

The latest declaration of class Manager is same as that of its previous one with the exception that we did not repeat the information of class Employee explicitly. This is what is meant by the Application of inheritance mechanism.

Please note that in the above example, Employee is called Base Class and Manager is called Derived Class.

4.4 VISIBILITY MODES

There were a total of three visibility modes. They are private, public, and protected. In the previous units, we have already learnt about private and public.

If a member of a class is declared as "protected", then only member functions and friends of the class in which it is declared and by member functions and friends of classes derived from this class can use its member.

4.5 SINGLE INHERITANCE

In this section, you will learn the ways of deriving a class from single class. So, there will be only one base class for the derived class.

4.5.1 Private Inheritance

Consider the following classes:

```
class A { /*.....*/ };
class C : private A
{
    /*
    .
    .
    .
    */
}
```

All the public parts of class A and all the protected parts of class A become private members/parts of the derived class C in class C. No private member of class A can be accessed by class C. To do so you need to write public or private functions in the Base class. A public function can be accessed by any object, however, private function can be used only within the class hierarchy that is class A and class C and friends of these classes in the above cases.

4.5.2 Public Inheritance

Consider the following classes:

```
class A { /*.....*/ };
class E : public A
{
    /*
    :
    :
    :
    */
};
```

Now, all the public parts of class A become public in class E and protected parts of A become protected in E.

4.5.3. Protected Inheritance

Consider the following classes:

```
class A { /*.....*/ };
class E : protected A
{
    /*
    .
    .
    */
};
```

Now, all the public and protected parts of class A become protected in class E.

No private member of class A can be accessed by class E. Let us take a single example to demonstrate the inheritance of public and private type in more details. Let us assume a class `closed_shape` as follows:

```
Class closed_shape
{
    public:
    .
    .
}
class circle: public closed_shape
    // circle is derived in public access mode from class
    // closed-shape
{
    float x, y;    // Co-ordinates of the centre of the circle
    float radius;
    public:
    .
    .
}
class semi-circle: public circle
{ private:
    public:
    .
    .
}
class rectangle: private closed_shape
{
    float x1, y1;
    float x2, y2;
```

public:

```

};
Class rounded-rectangle : public rectangle
{ private:
} public:

```

Figure 1 shows the access control for these inherited classes.

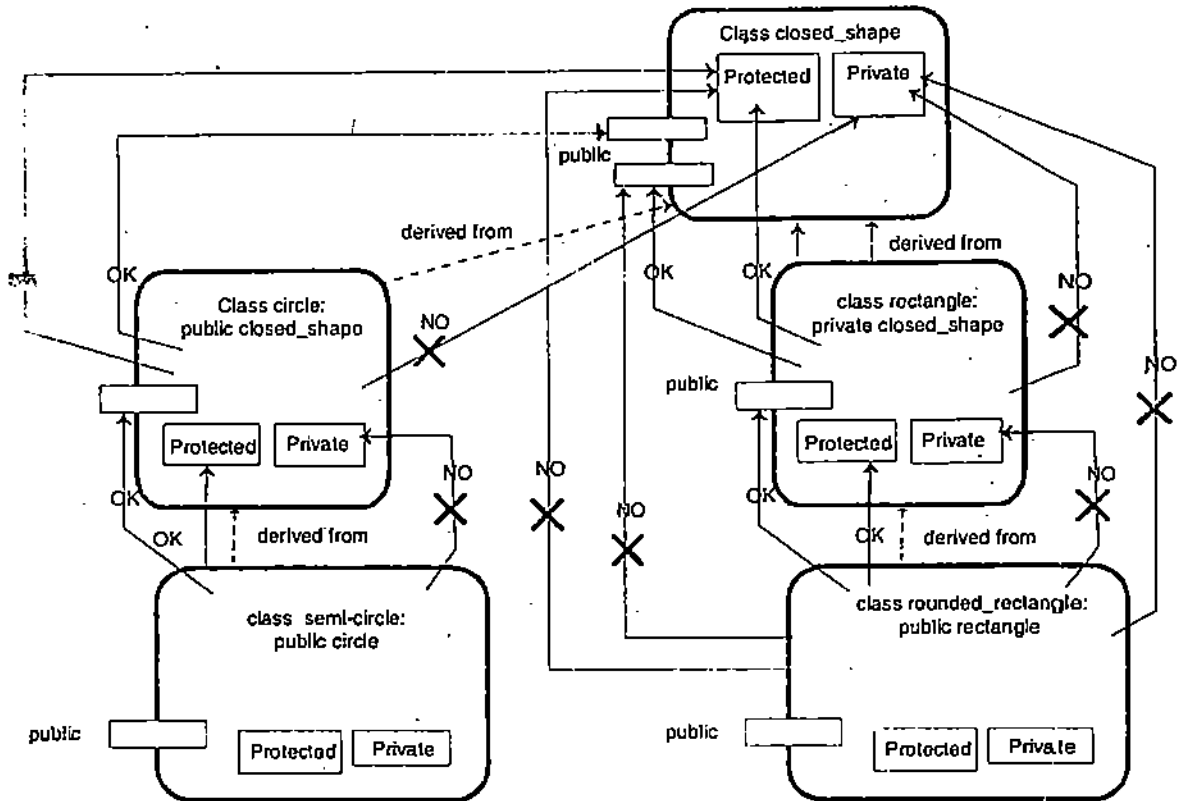


Figure 1: Access control

Please note the following in the above diagram:

- Class rectangle is a privately derived class, thus, all the public and protected members of class closed_shape will become private in rectangle class and, therefore, rounded-rectangle will not be able to access as they for this inherited class are private. On the other hand, circle & semi-circle both are derived as public classes and will allow access except for private members.

4.6 MULTIPLE INHERITANCE

A class can have more than one direct base classes.

Consider the following classes:

```

class A { /* .... */ };
class B { /* ..... */ };
class C : public A, public B
{
    /*

```

```
*/
};
```

This is called Multiple Inheritance. If a class is having only one base class, then it is known as single inheritance.

In the case of class C, other than the operations specified in it, the union of operations of classes A and B can also be applied.

4.7 NESTED CLASSES

A class may be declared as a member of another class. Consider the following:

```
class M1
{
    int n;
    public:
        int m;
```

```
};
```

```
class M2
```

```
{
    int n;
    public:
        int m;
```

```
};
```

```
class M3
```

```
{
    M1 N1;
    public:
        M2 N2;
```

```
};
```

Now, N1 & N2 are nested classes of M3. M3 can access only public members of N1 and N2. A nested class is hidden in the lexically enclosing class.

4.8 VIRTUAL FUNCTIONS

Polymorphism is a mechanism that enables same interface functions to work with the whole class hierarchy. Polymorphism mechanism is supported in C++ by the use of virtual functions. The concept of virtual function is related to the concept of dynamic binding. The term Binding refers to binding of actual code to a function call. Dynamic binding also called late binding is a binding mechanism in which the actual function call is bound at run-time and it is dependent on the contents of function pointer at run time. It means that by altering the content of function pointers we may be able to call different functions having a same name but different code, that is demonstrating poly-morphic behavior.

Let us look into an example for the above concept.

```
#include <iostream.h>
class employee
{
public:
    char* name;
    char* department;

    employee(char* n, char* d)
    {
        name = n;
        department = d;
    }

    virtual void print( );
};

void employee:: print( )
{
    cout << "name : "<< name;
    cout << "department: "<< department;
}

class manager : public employee
{
public:
    short position;
    manager (char* n, char* d, short p) : employee(n, d)
    {
        name = n;
        department = d;
        position = p;
    }

    void print( )
    {
        cout << name << "\n" << department << "\n" << position;
    }
};

void main( )
{
    employee* e ("john", "sales");
    manager* m ("james", "marketing", 3);
    e → print ( )
    m → print( );
}
```

The output will be:

John
Sales

Check Your Progress

- 1) None of the subclasses can access _____ members of a base class.
- 2) _____ members of a class can only be accessed by another class for which it is a member.
- 3) Both public and protected members of a class become _____ when this class is privately derived.
- 4) Both public and protected members of a class become _____ when this class is protectedly derived.

4.9 SUMMARY

In this unit, you have been exposed to the concepts of base class and derived classes. A derived class is a class, which includes the members of another class. This concept is also known as inheritance. When a derived class has more than one direct base class, then it is called Multiple Inheritance. There were three types of Inheritance. They are Public, Private and Protected Inheritance. We can also declare classes as members of another class. We have also touched on the concept of polymorphism.

4.10 MODEL ANSWERS

- 1) Private
- 2) Public
- 3) Private
- 4) Protected.

UNIT 5 STREAMS AND TEMPLATES

Structure

- 5.0 Introduction
- 5.1 Objectives
- 5.2 Output
- 5.3 Input
- 5.4 Files and Streams
- 5.5 Templates
- 5.6 Exception Handling
- 5.7 Summary
- 5.8 Model Answers

5.0 INTRODUCTION

In this unit, we will discuss about C++ streams library. We have already used "<<" and ">>" for standard input and output. In this unit, we will discuss the way of using the same operators for user defined types. Also, we will discuss the ways of reading data from and writing data to other files. We will discuss ways of opening files in different modes and closing them. In the context, we will be discussing about istream and ostream classes. In addition, we will also look into Exception Handling using C++.

5.1 OBJECTIVES

After studying this unit, you should be able to:

- Write programs which perform input and output from built in data types,
- Write programs which perform input and output from user defined data types, and
- Write programs which open other files and perform operations on the data in those files.

5.2 OUTPUT

As we have seen in a number of previous programs, the standard operation for output is left shift operation "<<".

We have already seen, how to apply this operator for built in types. For example, Consider

```
int x = 5;
cout << x; // will print 5
```

Now, we will consider, how to apply this operator for user-defined types:

The following program demonstrates the application of this operator for user defined types:

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

```

int i;

public:
    output(int j = 0)
    {
        i = j;
    }

    friend int show(output& a) {return a.i;}
};

ostream& operator << (ostream& s, output o)
{
    return s<<show(o);
}

void main( )
{
    output x(1);
    cout << "x =" << x;
}

```

5.3 INPUT

As previously seen, the standard operator for input is right shift operator ">>". We have already applied it for built in data types. For example,

```

int i;
cin >> i; // reads the value into i.

```

Now, we shall apply it for user defined data types. For an input operation, it is essential that the second argument is of reference type.

The following program demonstrates use of operator ">>" for user defined types:

```

#include <iostream.h>
class output {
int i;
public :
    output(int j = 0)
    {
        i = j;
    }
    friend int show(output& a)
    {
        return a.i;
    }
};

ostream& operator<<(ostream& s, output o)
{
    return s<<show(o);
}

istream& operator>>(istream& s, output& o)
{

```

```
        int i = 0;
        s >> i;
        if (s) o = output(i);
        return s;
    }
void main( )
{
    output x;
    cin >> x;
    cout << "x=" << x;
}
```

5.4 FILES AND STREAMS

In this section, we will see how to open files, close files and attaching files to streams.

The following example demonstrates those concepts. This is a program, which copies the data from one file and copies it to another. The program receives the names of the files as command line arguments.

```
#include <fstream.h>
void main(int argc, char* argv[])
{
    if(argc != 3)
        {   cout << "The no. of arguments should be 3";
            return;
        }
    ifstream read(argv[1]);
    ofstream write(argv[2]);
    char c;
    while(read.get(c)) write.put(c);
}
```

<fstream.h> is a library which declares the C++ stream classes that support file input and output. It also includes iostream.h.

ifstream is a class that provides an input stream to input from a file using a buffer.

Now, hereafter, "read" will be the handle to the file presented as second argument. So, whenever we have to refer to the second argument, we will be using "read". So, the second argument on the command line will be opened from which data is read in future operations.

ofstream is a class which provides an output streams to extract from a file using a buffer.

Now, hereafter, "write" will be the handle to the file presented as third argument. So, whenever we have to refer to the third argument, we will be using "write". So, the third argument on the command line will be opened for writing the data in future operations.

Here "read" & "write" are two names, which we have chosen on our own. The user can have any other names as handles to files.

So, finally, `read.get(c)` will read a character from file `argv[1]` and `write.put(c)` will write it to file `argv[2]`. This is automatically done until the end of file is encountered.

An "ofstream" is opened for writing by default and "ifstream" is opened for reading by default.

Also, we can open a file in other modes. In this case, the above classes will accept a second argument.

The different modes are defined in the following class `ios`.

```
class ios    {
public:
    enum open_mode
        { in = 1, out = 2, ate = 4, app = 010, trunc = 020, nocreate = 040,
          noreplace = 0100
        };
};
```

in mean "Open for Reading"

out means "Open for output"

ate means "Open and seek to end of file"

app means "Append"

trunc means "truncate file to 0-length"

nocreate means "Fail if file doesnot exist"

noreplace means "fail if file exists"

Consider,

```
ofstream ex(file, ios::out | ios::nocreate);
```

This means, "Open the file identified by variable "file" for output mode. If the file does not exist, the operation should fail.

We can also open a file for both input and output. For example, consider

```
fstream ex(name, ios::in | ios::out);
```

A file can be closed by calling the function `close()` on its stream. For example, consider

```
ex.close( );
```

5.5 TEMPLATES

Templates are also referred to as Parameterised types. It enables you to define generic classes. It defines a family of classes and functions. For example, stack of various data types such as int, float etc. Similarly function template for sort function will help create versions of sort function. It enables function of classes and function with parameters.

A Stack template:

```
template <class T> class stack
{
}
```

template <class T> prefix in the class declaration states that you are going to declare a class template and you would use T as a class name in the declaration. Thus, stack is a parametrized class with the type T as its parameter. With this definition of the stack class template you can create stacks for different data types, such as:

```
stack <int> istack
```

```
stack <float> fstack
```

You could similarly define a generic array class as follows:

```
Template <class T> class Array
```

```
{  
:  
:  
:  
}
```

You can then create instances of different Array types in the following manner:

```
Array <int> iarray(128);
```

```
Array <float> farray(32);
```

Function templates

Like class templates, function templates define a family of functions parameterized by a data type. For example, you could define a parameterized sort function for sorting any type of array like this.

```
template <class T> void sort(Array <T>)
```

```
{  
// Body of function (do the sorting)  
:  
:  
:  
}
```

You can invoke the sort function just like any ordinary function. The C++ compiler will analyse the arguments to the function and call the proper function.

Advantage of templates

1) It helps you define classes that are general in nature.

A Simple Stack Template

```
template <class T> class stack {  
T*v;  
T*p; // Stack Pointer  
int SZ;  
public:  
stack(int s) {v=p=new T[SZ = s]  
}  
~stack() {delete [ ] v;}
```

```
void Push (T a) { *p++ = a;}
T Pop( ) {return *-- p;}
}
```

The template <class T> prefix specifies that a template is being declared and that an argument T of type <type> will be used in the declaration.

Template <class T> says that T is a type name, it need not actually be the name of a class. The name of a class template follow by a type bracketed by <T is the name of a class (as defined by template) and can be used exactly like other class names. For example,

```
Stack <char> SC(100); // stack of characters defines an object SC of a class
                    stack<char>
```

Except for the special syntax of its name, stack <char> works exactly as if it had been defined.

```
class Stack-char {
char *v;
char *p;
int SZ;

public
Stack-char (int s) { v=p=new char[sz=s];
:
:
};
```

One can think of a template as a clever kind of macro that obeys the scope, naming and types rules of C++.

It is important to write templates so that they have a few dependencies on global information as possible. The reason is that a template will be used to generate function and classes based on unknown types and in unknown contexts. Any context dependency will surface as a debugging tool.

5.6 EXCEPTION HANDLING

Exception means unusual condition while execution of a program. They may cause programs to fail or may lead to errors. Some exceptions can be "array out of bound", null pointer assignment", "file does not exist", etc. The exception handling provides a uniform way of handling errors in C++ class libraries and programs. Let us discuss exception handling with the help of an example.

Let us assume a class

ReportIOException as:

```
Class ReportIOException {
Public:
ReportIOException (Char* filename):
_filename (filename) { }
Private:
Char*_filename;
}
```


In case in a Report class any output related problem such as there is no space on the storage device can be thrown as ReportIOException as:

```
void Report::write (const char* filename)
ofstream fs1 (filename) // open a file for output
if (! fs1) // If cannot open a new file as no space
{ // throw exception
    throw ReportIOException (filename);
}
// continue normal processing
.
.
}
```

The code for catching the exception is provided by the programmer using the Report class. It may be as:

```
// prepare report & try to output it but be ready for any error.
try
{
    mis.write ("Report1.rep");
    // In case of any error the exception will
    // be transferred to catch block.
}
catch (ReportIOException fileio)
{
    // Display an error message:
    cout << "ERROR! Cannot open file! Disk is full."
}.
```

Check Your Progress

- 1) "<<" and ">>" can be used for input and output on built-in data types as well as user defined types. True False
- 2) "<<" and ">>" should be _____ if they have to be applied for user defined types.
- 3) For an input operation, it is essential that the second argument is of _____ type.
- 4) A close() function should be applied if we want to close the file before reaching the end of scope in which the stream was declared. True False
- 5) It is possible to open a file in more than one mode simultaneously. True False

5.7 SUMMARY

So, we can overload operators "<<" and ">>" for input and output on userdefined types. For an input operation, it is essential that the second argument is of reference type. A file is opened for output for creating an object of class ofstream and a file is opened for reading by creating an Object of class ifstream. Other than for reading and writing, we can also open a file in more than one mode simultaneously. Though, you can close a file with close() function, it is not necessary to close a file, because the conceived object will contain a destructor which closes a file after the execution of that particular program. Any way, the close() function can be applied if we want to close the file before reaching the end of the scope in which the stream was declared. We have also discussed about the exception handling in C++.

In a C++ program, we can freely use any I/O functions, which are defined in C.

5.8 MODEL ANSWERS

- 1) True
- 2) Overloaded
- 3) Reference
- 4) True
- 5) True.

