

C++

Programming

BCS-031

For

BACHELOR OF COMPUTER APPLICATIONS [BCA]

By

DINESH VEERMA

MCA, BCA, 'A', 'O' LEVEL, SCJP(USA), JP, DAST



Useful For

IGNOU, KSOU (Karnataka), Bihar University (Muzaffarpur), Nalanda University, Jamia Millia Islamia, Vardhman Mahaveer Open University (Kota), Uttarakhand Open University, Kurukshetra University, Seva Sadan's College of Education (Maharashtra), Lalit Narayan Mithila University, Andhra University, Pt. Sunderlal Sharma (Open) University (Bilaspur), Annamalai University, Bangalore University, Bharathiar University, Bharathidasan University, HP University, Centre for distance and open learning, Kakatiya University (Andhra Pradesh), KOU (Rajasthan), MPBOU (MP), MDU (Haryana), Punjab University, Tamilnadu Open University, Sri Padmavati Mahila Visvavidyalayam (Andhra Pradesh), Sri Venkateswara University (Andhra Pradesh), UCSDE (Kerala), University of Jammu, YCMOU, Rajasthan University, UPRTOU, Kalyani University, Banaras Hindu University (BHU) and all other Indian Universities.

Closer to Nature



We use Recycled Paper



GULLYBABA PUBLISHING HOUSE PVT. LTD.

ISO 9001 & ISO 14001 CERTIFIED CO.

Published by:

GullyBaba Publishing House Pvt. Ltd.

Regd. Office:

2525/193, 1st Floor, Onkar Nagar-A,
Tri Nagar, Delhi-110035
(From Kanhaiya Nagar Metro Station Towards
Old Bus Stand)
011-27387998, 27384836, 27385249
+919350849407

Branch Office:

1A/2A, 20, Hari Sadan,
Ansari Road, Daryaganj,
New Delhi-110002
Ph. 011-45794768

E-mail: hello@gullybaba.com, **Website:** GullyBaba.com, GPHbook.com

New Edition

Price: ₹89/-

ISBN: 978-93-82688-08-2

Copyright© with Publisher

All rights are reserved. No part of this publication may be reproduced or stored in a retrieval system or transmitted in any form or by any means; electronic, mechanical, photocopying, recording or otherwise, without the written permission of the copyright holder.

Disclaimer: Although the author and publisher have made every effort to ensure that the information in this book is correct, the author and publisher do not assume and hereby disclaim any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause.

If you find any kind of error, please let us know and get reward and or the new book free of cost.

The book is based on IGNOU syllabus. This is only a sample. The book/author/publisher does not impose any guarantee or claim for full marks or to be passed in exam. You are advised only to understand the contents with the help of this book and answer in your words.

All disputes with respect to this publication shall be subject to the jurisdiction of the Courts, Tribunals and Forums of New Delhi, India only.

HOME DELIVERY OF GPH BOOKS

You can get GPH books by VPP/COD/Speed Post/Courier.

You can order books by Email/SMS/WhatsApp/Call.

For more details, visit gullybaba.com/faq-books.html

Our packaging department usually dispatches the books within 2 days after receiving your order and it takes nearly 5-6 days in postal/courier services to reach your destination.

Note: Selling this book on any online platform like Amazon, Flipkart, Shopclues, Rediff, etc. without prior written permission of the publisher is prohibited and hence any sales by the SELLER will be termed as ILLEGAL SALE of GPH Books which will attract strict legal action against the offender.

Preface

C++ is the most widely used programming language around and is an industry standard for programming applications of all kinds. In addition, C++ is a highly efficient programming language that can conserve resources more effectively than languages such as Visual Basic or Delphi. In fact, because of its functionality and style, in many ways, C++ is the only non-Web-based programming language that you might ever need to know.

The book ‘C++ Programming’ (BCS-031) is fulfilling the requirements of students. The whole book is organised in well-planned 9 chapters. All the chapters are followed by relevant review questions and programming exercises are also provided in addition to review questions.

In order to make you learn the subject matter in better way, we have presented this book in a Question-Answer format. Besides this, we have provided you some sample question papers with answers in the last of this book which will widen your knowledge and scope of understanding.

Hopefully, it would be accorded a warm welcome in all the concerned quarters.

– Dinesh Veerma

For The Publishers

My compliments go to the **GullyBaba Publishing House (P) Ltd.**, and its meticulous team who have been enthusiastically working towards the perfection of the book.

Their teamwork, initiative and research have been very encouraging. Had it not been for their unflagging support, this work wouldn't have been possible. The creative freedom provided by them along with their aim of presenting the best to the reader has been a major source of inspiration in this work. Hope that this book would be successful.

– Dinesh Veerma

Publisher's Note

The present book 'BCS-031' of BCS series in the BCA education programme is targeted for examination purpose as well as enrichment. With the advent of technology and the Internet, there has been no dearth of information available to all; however, finding the relevant and qualitative information, which is focused, is an uphill task.

We at **GullyBaba Publishing House (P) Ltd.**, have taken this step to provide quality material which can accentuate in-depth knowledge about the subject. GPH books are a pioneer in the effort of providing unique and quality material to its readers. With our books, you are sure to attain success by making use of this powerful study material.

Our site **www.gullybaba.com** is a vital resource for your examination. The publisher wishes to acknowledge the significant contribution of the Team Members and our experts in bringing out this publication and highly thankful to Almighty God, without His blessings, this endeavor wouldn't have been successful.

– Publisher

Topics Covered

Block-1 Basics of Object Oriented Programming & C++

- Unit-1 Object Oriented Programming
- Unit-2 Introduction to C++
- Unit-3 Objects and Classes
- Unit-4 Constructors and Destructors

Block-2 Inheritance and Polymorphism in C++

- Unit-1 Inheritance
- Unit-2 Operator Overloading
- Unit-3 Polymorphism and Virtual Function

Block-3 Advanced Features of C++

- Unit-1 Streams and Files
- Unit-2 Templates and STL
- Unit-3 Exception Handling

Contents

Chapter-1	Basic Foundation.....	1
Chapter-2	Starting C++.....	11
Chapter-3	Objects and Classes.....	21
Chapter-4	Introduction to Functions.....	63
Chapter-5	Operator Overloading.....	83
Chapter-6	Inheritance.....	89
Chapter-7	Template and Streams.....	135
Chapter-8	Exception Handling.....	161
Chapter-9	Supplementary.....	167

Sample Papers

Sample Paper-I.....	249
Sample Paper-II.....	250

Question Papers

(1) Dec: 2012.....	251
(2) June: 2013.....	253
(3) Dec: 2013.....	255
(4) June: 2014.....	257
(5) Dec: 2014.....	259
(6) June: 2015.....	261
(7) Dec: 2015.....	263
(8) June: 2016 (Solved).....	265
(9) Dec: 2016.....	281
(10) June: 2017	283
(11) Dec: 2017 (Solved).....	285
(12) June: 2018	294
(13) Dec: 2018 (Solved).....	296
(14) June: 2019.....	312
(15) Dec: 2019.....	314
(16) June: 2020.....	316
(17) Dec: 2020.....	318

1

Basic Foundation

Procedural Languages and OOP

Q1. What is Object Oriented Programming? How is it different from procedural programming? Does object oriented programming paradigm not use procedural programming at all? Justify your answer.

Or

What are the advantages of object oriented programming? How is it different from procedural programming? What are the disadvantages of object oriented systems?

Ans. Object Oriented Programming is a programming methodology where the main emphasis is given on the object or the data and not on the functions or operations that operate on data. Data is undervalued in procedural programming. Here, the design puts data upfront and organisation of data is significant area. Access rights are reserved as well to protect data from accidental changes by an intruder.

Advantages:

- Promotes organisation in programming style.
- Offers reusability, i.e. the work done in the past can be put into use as such in future.
- One function can have many forms that permits widening of area of application.

Disadvantages:

- OOP is a high level concept so takes more time to execute as many routines run behind at the time of execution.
- Offers less number of functions as compared to low level programming which interacts directly with hardware.
- Increased burden on part of OOP developer.

Difference: Procedural languages are the languages where each statement in the language tells the computer to do something. Get some input add these numbers, divided by 6, display that output. A program in a procedural language is a list of instructions. For very small programs no other organising principle (often called a paradigm) is needed. The programmer creates the list of instructions, and the computer carries them out. Pascal, C, BASIC, Fortran, and similar languages are procedural languages.

OOP does use procedural programming but only for small internal code. Actually the whole organisation of data finds its base at procedural steps. The structure that we call OOP has its foundation in procedural languages. E.g. data & function put together to form an object, but data and functions follow sequential or procedural steps of programming.

Division into Functions

When programs become larger, a single list of instructions becomes unwieldy. Few programmers can comprehend a program of more than a few hundred statements unless it is broken down into smaller units. For this reason the function was adopted as a way to make programs more comprehensible to their human creators. (The term function is used in C++ and C. In other languages the same concept may be referred to as a subroutine, a subprogram, or a procedure).

Problems with Structured Programming

The principal idea behind structured programming was as simple as the idea of “divide and conquer”. A computer program could be regarded as consisting of a set of tasks. Any task that was too complex to be simply described would be broken into a set of smaller component tasks, until the tasks were sufficiently small and self-contained to be easily understood. As programs grow ever larger and more complex, even the structured programming approach begins to show signs of strain. You may have heard about, or been involved in, horror stories of program development. The project is too complex, the schedule slips, more programmers are added, complexity increases, cost skyrocket, the schedule slips further, and finally disaster.

Analysing the reasons for these failures reveals that there are weakness in the procedural paradigm itself. No matter how well the structured programming approach is implemented, large programs become excessively complex.

What are the reasons for this failure of procedural languages? One of the most crucial is the role played by data.

Data is undervalued

In a procedural language, the emphasis is on doing things--read the keyboard, add the values, check for error, and so on. The subdivision of a program into functions continues this emphasis. Functions do thing as single program statements do. What they do may be more complex or abstract, but the emphasis is still on the action.

What Happens to the Data in this Paradigm?

Data is, after all, the reason for a program's existence. Data is given second-class status in the organisation of procedural languages.

For example, in an inventory program, the data that makes up the inventory is probably read from a disk file into memory, where it is treated as a global variable. By global we mean that the variables that constitute the data are declared outside of any function, so they are accessible to all functions. These functions perform various operations on the data. They read it, analyse it, update it, rearrange it, display it, write it back to the disk, and so on.

We should note that most languages, such as pascal and C, also support local variables, which are hidden within a single function. But local variables are not useful for important data that must be accessed by many different functions. Figure below shows the relationship between global and local variables.

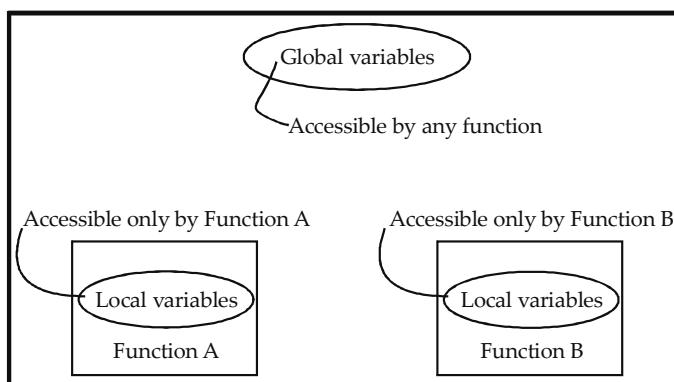


Fig. 1.1: Global and local variables

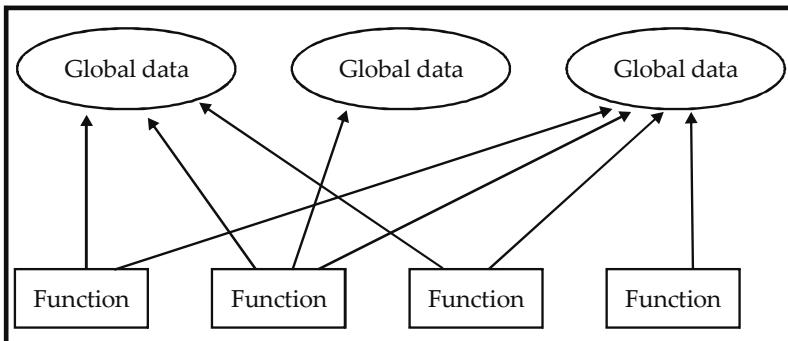


Fig. 1.2: Procedural paradigm

Now suppose a new programmer is hired to write a function to analyse this inventory data in a certain way. Unfamiliar with the subtleties of the program, the programmer creates a function which has complete access to the data. It's like leaving your personal papers in the lobby of your apartment building: Anyone can change or destroy them. In the same way, Global data can be corrupted by functions that have no authorisation to change it. What is needed is a way to restrict access to the data, to hide it from all but a few critical functions. This will Protect the data, simplify maintenance, and offer other benefits as well.

Traditional Languages does not Provide Extensibility

One is the difficulty of creating new data types. Computer languages typically have several built-in data types: your own data type? Perhaps you want to work with complex numbers, or two-dimensional coordinates or dates-quantities the built-in-data types don't handle easily. Being able to create your own types is called extensibility; you can extend the capabilities of the language. Traditional languages are not usually extensible. Without unnatural convolutions, you can't bundle together both X and Y coordinates into a single variable called point, and then add subtract values of this type. The result is that traditional programs are more complex to write and maintain.

The Object-Oriented Approach : Solution to above problems

The fundamental idea behind object-oriented languages is to combine both data and the functions that operate on that data into a single unit called an object. An object's functions, called member functions in C++ typically provide the only way to access its data. If you want to read a data item in an object, you call only way to access its data. If you want to read a data item in an object, you call a member function in the object. It will read the item and return the value to you it can't access the data directly. The data is hidden, so it is safe to accidental alterations. Data and its functions are

said to be encapsulated into a single entity, data encapsulation and data hiding are key terms in the description of object-oriented languages.

No other functions can access the data. This simplifies writing, debugging, and maintaining the program.

A C++ program typically consists of a number of objects, which communicates with each other by calling one another's member functions.

Sometime, what are called member functions in C++ are called methods in some other-oriented (OO) languages (such as Smalltalk, one of the first OO languages). Also data items are referred to as instance variables. Calling an object's member function is referred to as sending a message to the object. These terms are not usually used in C++.

An Analogy

You might want to think of objects as departments such as sales accounting personnel, and so on in company. The people in each department control and operate on that department's data. Dividing the company into departments makes it easier to comprehend and control the company's activities, and helps to maintain the integrity of the information used by the company.

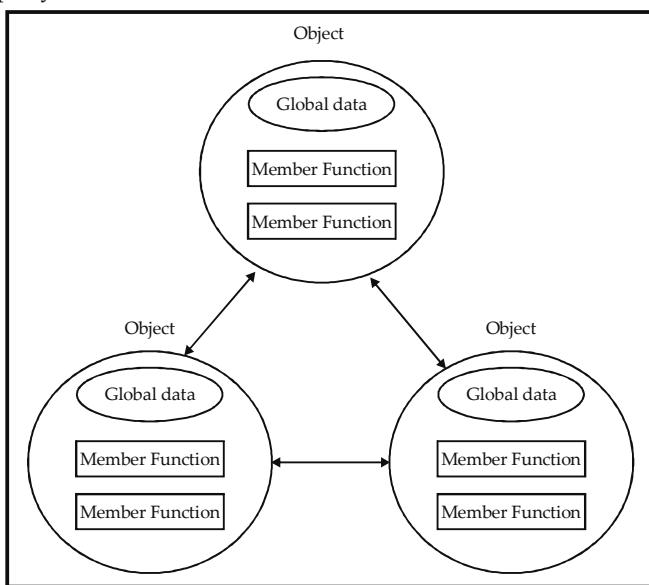


Fig. 1.3: Object-Oriented Paradigm

Know Object-Oriented Languages in detail

Let's briefly examine a few of the major elements of object-oriented languages or C++.

Objects

When you approach a programming problem in an object-oriented language you are no longer ask how the problem will be divided into functions, but how it will be divided into object. Thinking in terms of object, rather than functions has a surprisingly helpful effect on how easily programs can be designed. This results from the close match between objects in the programming sense and objects in the real world.

What kinds of things become objects in object-oriented programs? The answer to this is limited only by your imagination, but below are some examples:

Physical Objects

- Automobiles in a traffic-flow simulation
- Electrical components in a circuit-design program
- Countries in an economics model
- Aircraft in an-traffic-control system

Elements of the Computer-user Environment

- The mouse and the keyboard
- Windows
- Menus
- Graphics object (lines, rectangles, circles)

The match between programming objects and real-world object is the happy results of combining data and functions: The resulting objects offer a revolution in program design. No such close match between programming constructs and the items being modeled exists in a procedural language.

Classes

In OOP we say that objects are members of classes. What does this mean? Let's look at an analogy. Almost all computer languages have built-in data types. For instance a data type int. Meaning integer, is pre-defined in C++. You can declare as many variables of types int as you need in your program:

```
int marks;  
int no_of_assignments;  
int no_of_papers;
```

In a similar way you can make objects of the same class, as shown in the following figure.

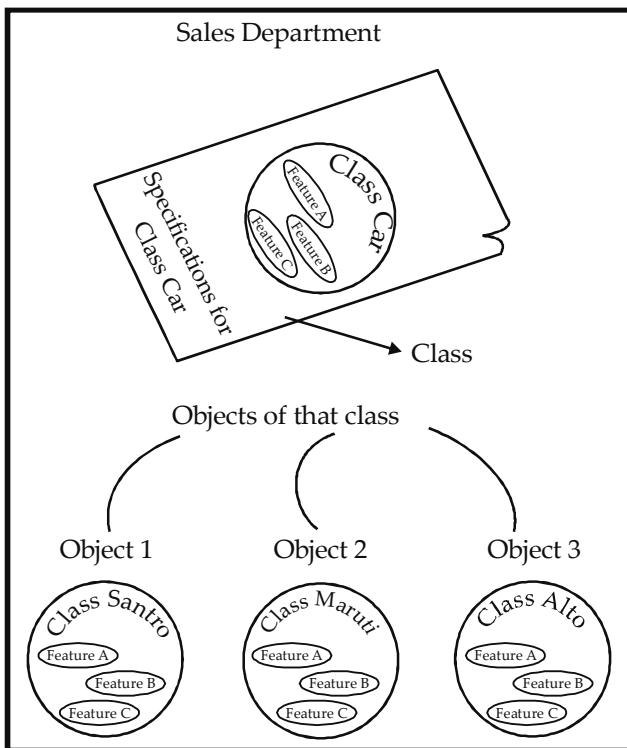


Fig. 1.4: A Class and its Objects

A class serves as a plan, or template. It specifies what data and what functions will be included in object of that class. Defining the class doesn't create any object, just as the mere existence of a type int doesn't create any variables.

A class is thus a collection of similar objects. This fits our nontechnical understanding of the class of mountaineers. There is no one person called "mountaineers" but specific people with specific names are members of this class if they possess certain characteristics.

Inheritance

The idea of classes leads to the idea of inheritance. In our daily lives, we use the concept of classes being divided into subclasses. We know that the class of animals is divided into mammals, amphibians, insects, bird, and so on. The class of vehicles is divided into cars, truck, buses, and motorcycles.

Principle in this sort of division is that each subclass shares common characteristics with the class from which it's derived. Cars, truck, buses, and motorcycles all have wheels and a motor; these are the defining

characteristics of vehicles. In addition to the characteristics: buses, for instance, have seats for many people, while trucks have space for large loads.

This ideas shown in the following figure notice in the figure that features A and B, which are part of the base class, are common to all derived classes, but that each deriver class also has features of its own.

In a similar way, an OOP class can be divided into subclass. In C++ the original class is called the base class: other classes can be defined to share characteristics, but add their own as well. These are called derived classes.

Inheritance is somewhat analogous to using functions to simplify a traditional procedural program. If we find that three different sections of a procedural program do almost exactly the same thing, we recognise an opportunity to extract the common elements of these three sections and put them into a single function. As functions do in a procedural program, inheritance shortens an object-oriented program and clarifies the relationship among program elements.

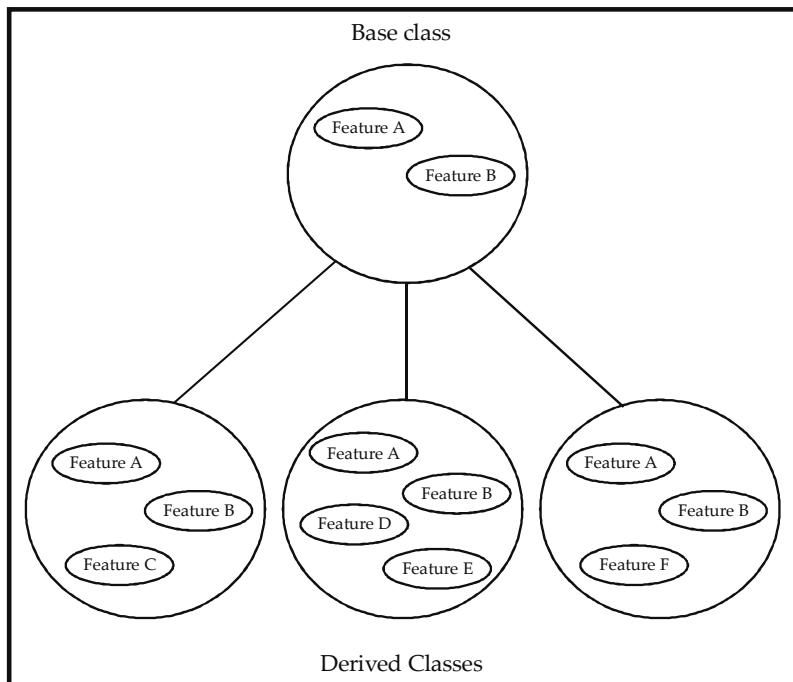


Fig. 1.5: Inheritance

Reusability

Once a class has been written, created, and debugged, it can be distributed to other programmers for use in their own programs. This is called reusability. It is similar to the way a library of functions in a procedural language can be incorporated into different programs.

In OOP the concept of inheritance provides an important extension to the idea of reusability. A programmer can take an existing class without modifying and can use it to develop any new application. This is done by deriving a new class from the existing one. The new class will inherit the capabilities of the old one but is free to add new features of its own.

For example, you might have written (or purchased from someone else) a class that creates a menu system, such as used in the Turbo C++ integrated development system (IDE). This class works fine, and you don't want to change it but you want to add the capability to make some menu entries flash on and off. To do this you simply create a new class that inherits all the capability of the existing one but adds flashing menu entries.

Creating New Data Types

One of the benefits of objects is that it give the programmer a convenient way to construct new data types. Suppose you work with two-dimensional positions (such as X and Y coordinates, or latitude and longitude) in your program. You would like to express operations on these positional values with normal arithmetic operations, such as:

```
position1 = position2 + origin;
```

Where the variables position1, position2 and origin each represent a pair of independent numerical quantities. By creating a class that incorporates these two values, and declaring position1, position2 and origin to be objects of this class, we can, in effect, create a new data type. Many features of C++ are intended to facilitate the creation of new data types in this manner.

Polymorphism & Overloading

Note that the = (equal) and + (plus) operators, used above don't act the same way they do in operations on built-in types like int. The object position1 and so on are not pre-defined in C++, but are programmer-defined objects of class position. How do the = and + operations know how to operate on objects? The answer is that we can define new operations for these operators. These operators will be member functions of the position class.

Using operators or functions in different ways, depending on what they are operating on, is called polymorphism. One thing with several distinct forms. When an existing operator, such as + or = is given the

capability to operate on a new data type, it is said to be overloaded. Overloading is a kind of polymorphism; it is also an important features of OOP.



Opportunities for Authors

If you have good knowledge on any subject, and a keen desire and aptitude to write, GPH is the place for you. We are taking this initiative of introducing new authors and getting their work published.

Good Chance for Teachers/Faculties/Diligent Students.

Log on to www.GullyBaba.com

Apply for offline/online, Part time or full time writers.

Email us your detail on : jobs@gullybaba.com

2

Starting C++

Introduction to C++

C++ was developed by Bjarne Stroustrup at Bell Laboratories in 1983. Originally, it was called as “C with class”. C+ as an enhancement to the C language was developed primarily to facilitate managing, programming and maintaining large software projects.

The most important aspect of the C language is, probably, the flexibility to do whatever the programmer wants. The limits of the language are defined by the programmer’s imagination. Unfortunately, with very large projects in which many programmers use shared routines, this liberty can lead to what are called as ‘side effects’. This is one problem, which C++ attempts to resolve by restricting indiscriminate access. At the same time, C++, also attempts to keep the freedom and flexibility given by the language. All the keywords of C are keywords of C++ also. In addition we have some new one’s too.

However, C++ is not merely an extension of the C language, where some new symbols have been added. The basic purpose of C++ language is to add features to the C language that supports the concepts of OOP.

What are the advantages of C++?

- (1) **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.

- (2) **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.
- (3) **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.
- (4) **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 initialisation, and call to library functions/classes, you can use the libraries.
- (5) **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.
- (6) **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.
- (7) **Programming in Large:** Many programming languages have their own limitation; some have limitations of line of code, some on recursion, etc. However, C++ provides many features that support 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 program to manageable one.

C++ and Object-Oriented Programming

The most important term is 'Abstraction'. Abstraction means ignoring those aspects of a subject that are not revealed to the current purpose in

order to concentrate more fully on those that are relevant. To an analyst who is as good as a subject expert it means choosing certain things or aspects of the system over others at a time. Most of the things in the real world are intrinsically complex, far more complex than one can comprehend at a time. By using abstraction one selects only a part of the whole complex instead of working with the whole thing. It does not mean ignoring the details altogether but only temporarily, the details are embedded inside and are dealt with at a later stage. Abstraction is used by OOP as a primary method of managing complexity.

Association is the principle by which ideas are connected together or united. Association is used to tie together certain things that happen at some point in time or under similar conditions.

Communicating with messages is the principle for managing complexity, which is used when interfaces are involved to communicate between the entities. OOP uses classes and objects have attributes, which can be exclusively accessed by services of the objects. To change the attributes of the object (a data member of the object), a message has to be sent to the object to invoke the service. This mechanism helps in implementing encapsulation and data abstraction.

Encapsulation and Data Hiding

The property of being a self-contained unit is called encapsulation. The idea that the encapsulated unit can be used without knowing how it works is called data hiding.

Encapsulation is the principle by which related contents of a system are kept together. It minimises traffic between different parts of the work and it separates certain specific requirements from other parts of specification, which use those requirements.

The important advantage of using encapsulation is that it helps to minimise rework when developing a new system. The part of the work, which is prone to change, can be encapsulated together. Thus any changes can be made without affecting the overall system and hence changes can be easily incorporated.

When an engineer needs to add resistor to the device she is creating, she doesn't typically build a new one from the scratch. She walks over to a bin of resistors, examines the bin of resistors, examines the colored bands that indicate the properties, and picks the one she needs. The resistor is a "black box" as far as the engineer is concerned - that is, she doesn't care how it does its work as long as it conforms to her specifications.

All the resistor's properties are encapsulated in the resistor object - they are not spread out through the circuitry. It is not necessary to understand how the resistor works to use it effectively, because its data is hidden inside the resistor covering.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called classes. Once created, a well-defined class acts as a fully encapsulated entity and can be used as a whole unit. The actual inner workings of the class should be hidden; users of well-defined class do not need to know how the class works, only how to use it.

Inheritance and Reuse

Consider a car manufacturing company. When they want to build a new car, they have two choices. They can start from the scratch, or they can modify an existing model. Perhaps their former model is nearly perfect (say CAR1), but they would like to add a turbocharger and a six-speed transmission. The chief engineers would prefer to use the earlier model and make some changes to it than creating a new one (CAR2).

C++ supports the idea of reuse through inheritance. Through this concept, a new type can be declared that is an extension of the existing type. This new subclass is said to derive from the existing type and is sometimes called a derived type.

Polymorphism

The CAR2 in the above example may respond differently than the CAR1 when the car accelerates. The CAR2 might engage fuel injection and a turbocharger, for example, whereas the CAR1 simply get petrol into its carburetor. A user, however, does not have to know about these differences; the user simply presses the accelerator and the car responds with the correct function.

C++ supports this idea - that different objects do "the right thing" - through function polymorphism and class polymorphism. *Poly* means many, whereas *morph* means form. Thus, *polymorphism* refers to the same name taking many forms.

Some Differences between C and C++

There are several differences between the C language and C++, which have nothing to do with OOP. Some of them are highlighted below.

Additional keywords in C++

class	friend	virtual	inline
private	public	protected	const
this	new	delete	operator

The actual use and description of these additional keywords will be covered in their specific contexts.

Comments

Comments are integral part of any program. Comments help in coding, debugging and maintaining a program. The compiler ignores them. They should be used liberally in the program.

In C++, a comment starts with two forward slashes (//) and ends with the end of that line. A comment can start at the beginning of the line or on a line following the program statement. This form of giving comments is particularly useful for the short line comments. If a comment continues on more than a line, the two forward slashes should be given on every line.

The C style of giving a comment is also available in C++. This style /*....*/ is particularly useful when the comment spans more than a line.

E.g.

```
void main()
{
    /* this is a good old style of giving
       a comment. It can be continued to
       next line and has to be ended with */
```

```
clrscr();           //Clears the Screen
init();            //Initialise variables
:
:
}
```

Variable Declaration

This declaration of variables in the C language is allowed only in the beginning of their block, prior to executable program statements. In C++ declaration of variables can be interspread with executable program statements. The scope of variables, however, remains the same - the block in which they are declared.

E.g.

```
void main()
{
    int x = 10;
    printf ("the value of x= % d \n",x);
    int y = 0;
    for(int z= 0; z < 10; z++)           //variable declared here
    {
        y++;
        x++;
    }
}
```

Although, a deviation from the old style of declaring all variables in the beginning of the block, this does save some amount of memory, i.e. a variable is not given a memory until the declaration statement. Also, since a variable can be declared just before using it is suppose to give a better control over variables.

The Scope Resolution Operator (::)

Global variables are defined outside any functions and thus can be used by all the functions defined thereafter. However, if a global variable is declared with the same name as that of a local variable of a function, the local variable is the one in the scope when the program executes that function. The C++ language provides the scope resolution operator (::) to access the global variable thus overriding a local variable with the same name. This operator is prefixed to the name of the global variable. The following example shows its usage.

```
int global = 10;  
void main()  
{  
    int global = 20;  
  
    printf("Just writing global prints : %d\n", global);  
    printf("Writing ::global prints : %d\n", ::global);  
}
```

The output of this program will be:

Just writing global prints : 20

Writing ::global prints : 10

Default Arguments

A default argument is a value that is automatically assigned to a formal variable, if the actual argument from the function call is omitted.

E.g.

```
void drawbox(int x1=1, int y1=1, int x2=25, int y2=80, int color=7);  
                                // Prototype  
void main (void)  
{  
    drawbox(10, 1, 25, 80, 14);      // Parameters passed  
    drawbox();                      // Uses default arguments  
}
```

```
void drawbox(int x1, int y1, int x2, int y2, int color)
{
    // Body of the function
}
```

Function `drawbox()` draws a box around the edges of the coordinates passed as parameters. If these parameters are omitted, then the default values, as given in the declaration are passed.

When a function is declared, default values must be added from right to left. In other words, a default value for a particular argument cannot be given unless all default values for the arguments to its right are given. This is quite logical, since, if there are some arguments missing in the middle then the compiler would not know as to which arguments have been specified and which should be taken as default.

E.g.

```
void drawbox(int x1=1, int y1=1, int x2=25, int y2=80, int color = 7);
                                // Valid
```

```
void drawbox(int x1, int y1, int x2, int y2, int color = 7);  
                                // Valid
```

```
void drawbox(int x1=1, int y1=1, int x2=25, int y2=80, int color);  
                                // Not Valid
```

Default arguments are useful when the arguments almost have the same value. They are also useful to increase the number of arguments to a function in an already working program. In this case, using default arguments mean that existing function calls need not be modified, whereas, new function calls pass more number of arguments.

The New and Delete operators

The C language has defined library functions: `malloc()` and `free()` for dynamic allocation and de-allocation of memory. C++ provides yet another approach to allocate blocks of memory - the *new operator*. This operator allocates memory for a given size and returns a pointer to its starting point. C++ also provides `delete`, to release the memory allocated by `new`. The pointer returned by the `new` operator need not be typecasted.

E.g.

In the above example, new returns a pointer to a block of size bytes. It is synonymous with declaring a character array. However, declaring an array is an example of static binding - the array is built at the compile time. This array remains in existence right from the beginning of the program to its end, even if not in use. Whereas, the array declared by new operator can be allocated memory only when required and can be released when over with, using the delete operator. This is an example of dynamic binding.

It is possible that the system may not have enough memory available to satisfy a request by the new operator. In this case, new returns a null pointer. Thus, the returned value from the new should always be checked before performing any operations on it, which otherwise may lead to a sudden system crashes.

Once the new operator allocates memory, it can be accessed using the indirection operator (*) or the array subscripts.

E.g.

```
int *some = new int[10];           //A block of 10 integers
double *d5 = new double[5];        //A block of 5 doubles
char *cp;
cp = new char;                   //One character only
delete some;                     //Release memory
delete d5;

char *ptr , *str = "the sky is blue";
ptr = new char [strlen(str)+1];    //Character array

if(ptr != 0)
{
    strcpy(ptr,str);
}
else
{
    printf("not enough memory");
}

struct date
{
    int dd,mm,yy;
};
```

```
date *dp = new date;      // This allocated a new structure to
                         // the pointer variable dp

printf("enter day");
scanf("%d",&dp->dd);
:
}
```

The Cin and Cout objects

In C++ the keyword **cin** with the extraction operator “**>>**” is used to accept the input from the keyword. **Cout** with the insertion operator “**<<**” is used to display string literals and values, in place of scanf() and printf() functions of C. However, here we do not need any type specifiers like %d, %f, etc. to denote the variables, cin and cout can be used in context with any basic data types.

E.g.

```
int ivar;
float fvar;
char cvar;
char name[10];

cin >> ivar ;
cin >> fvar >> cvar >> name;

cout<< " ivar = " << ivar << endl;
cout<< " fvar = " << fvar << " cvar= " << cvar << name;
```

Q1. What is the difference between Object Base and Object Oriented?

Ans. The main difference between the object oriented and object base is that in object oriented you use built in class, template class and also define your own class according to need. While in case of object base we can only use built in class. User cannot create his own class.

Q2. What is the meaning of data abstraction? What is encapsulation? How does C++ achieve these two concepts? Describe with the help of an example.

Ans. Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and cost, and functions to operate on these attributes. They

encapsulate all the essential properties of the objects that are to be created. Since the classes use the concept of data abstraction, they are known as Abstract Data Types (ADT).

E.g.

```
float a;  
int b;
```

Here, background details of how is number stored in float or int data type is not a matter of programmer's concern.

The wrapping up of data and functions into a single union (called class) is known as encapsulation. Data encapsulation is the most striking feature of a class. The data is not accessible to the outside world and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called data hiding.

E.g.

```
class A {  
    int a;  
public :  
    void getData() {}  
};
```

```
void main()  
{  
    A a1;  
    a1.getData();  
}
```

The data 'a' & function getData() are encapsulated in object a1.



3

Objects and Classes

Overview

Object-oriented programming (OOP) is a conceptual approach to design programs. This session will continue our discussion on OOP and C++. Some of the features of OOP are Data encapsulation and data hiding, and possible because of data abstraction. All these features have one thing in common – the vehicle to implement them. This vehicle is class. It is a new data type similar to structures. Since, the structure data type is a stepping-stone to this singular important concept of C++, now here, we will take another look at it.

Structures

A structure is a user-defined data type, which may contain different data types as its members. Creating a structure is a two-part process. First, a structure template is defined. This template gives a lot of information to the compiler. For instance, How it is stored in the memory? How many member variables are there in this composite data element? What are the types of its member variables? For data types like int, float, double, char this information is built into the compiler. But for the user-defined data types like structures, it has to be provided to the compiler. This is given in the definition of the structure template. This template creates a new data type. Variables of this new data type can then be declared and used like basic data types.

E.g.

```
struct Student
{
    int Rollno;
    char Name[15];
    int Marks[6];
    float percent;
};
```

The above example depicts structure called Student with enclosed data variables. The keyword struct is used to define a structure template. Student is a name or tag. The variables of this type can be declared as follows:

```
struct Student s1,s2;
```

In C++ you can even omit the struct tag and declare the variables as,

```
Student s1 = {100, "Gullybaba", 20, 10, 30, 40, 50, 60, 35.0};
```

```
Student s2,jack;
```

```
Student *sptr = &s1;
```

```
Student s[100];
```

In C++, functions also can be declared in a structure template. These functions are called as member functions.

E.g.

```
struct Student
{
    int Rollno;
    char Name[15];
    int Marks[6];
    float percent;

    void Getdetails();
    void Putdetails();
    int ClacPercent();
};
```

There are three functions in the template given above. These functions have been incorporated in the structure definition to make it a fundamental unit. The binding of data and the functions that operate on that data is

called as data encapsulation. But the problem is that the member variables of this structure can be accessed by functions other than the ones declared in the structure template. This direct access of member variables of a structure by functions outside the structure is not what OOP has to offer. The actual OOP methodology in this regard is implemented in C++ not by the structure data type, but by the data type class. The data type class is just like structures, but with a difference. It also has some access specifier, which controls the access to member variables of a class.

Introduction to Classes

Object-oriented programming (OOP) is a conceptual approach to design programs. It can be implemented in many languages, whether they directly support OOP concepts or not. The C language also can be used to implement many of the object-oriented principles. However, C++ supports the object-oriented features directly. All these features like Data abstraction, Data encapsulation, Information hiding etc have one thing in common the vehicle that is used to implement them. The vehicle is “class”.

Class is a user defined data type just like structures, but with a difference. It also has three sections namely private, public and protected. Using these, access to member variables of a class can be strictly controlled.

Class Definition

The following is the general format of defining a class template:

```
class tag_name
{
public :                                         // Must
    type member_variable_name;
    :
    type member_function_name();
    :

private :                                       // Optional
    type member_variable_name;
    :
    type member_function_name();
    :
};
```

The keyword `class` is used to define a class template. The private and public sections of a class are given by the keywords ‘`private`’ and ‘`public`’

respectively. They determine the accessibility of the members. All the variables declared in the class, whether in the private or the public section, are the members of the class. Since the class scope is private by default, you can also omit the keyword private. In such cases you must declare the variables before public, as writing public overrides the private scope of the class.

E.g.

```
class tag_name
{
    type member_variable_name;           //private
    :
    type member_function_name();        //private
    :

public:                                // Must
    type member_variable_name;
    :
    type member_function_name();
    :
};
```

The variables and functions from the public section are accessible to any function of the program. However, a program can access the private members of a class only by using the public member functions of the class. This insulation of data members from direct access in a program is called information hiding.

E.g.

```
class player
{
public :
    void getstats(void);
    void showstats(void);
    int no_player;
private :
    char name[40];
    int age;
    int runs;
    int tests;
```

```
        float average;  
        float calcaverage(void);  
    };  
}
```

The above example models a cricket player. The variables in the private section - name, age, runs, highest, tests, and average - can be accessed only by member functions of the class calcaverage(), getstats() and showstats(). The functions in the public section - getstats() and showstats() can be called from the program directly, but function calcaverage() can be called only from the member functions of the class - getstats() and showstats().

With information hiding one need not know how actually the data is represented or functions implemented. The program needs not to know about the changes in the private data and functions. The interface(public) functions take care of this. The OOP methodology is to hide the implementation specific details, thus reducing the complexities involved.

Classes and Objects

A class is a vehicle to implement the OOP features in the C++ language. Once a class is declared, an object of that type can be defined. An object is said to be a specific instance of a class just like Maruti car is an instance of a vehicle or Pigeon is the instance of a bird. Once a class has been defined several objects of that type can be declared. For instance, an object of the class defined above can be declared with the following statement:

player Sachin, Dravid, Mohan;

Or

class player Sachin, Dravid, Mohan;

where, Sachin and Dravid are two objects of the class player. Both the objects have their own set of member variables. Once the object is declared, its public members can be accessed using the dot operator with the name of the object. We can also use the variable no_player in the public section with a dot operator in functions other than the functions declared in the public section of the class.

E.g.

```
Sachin.getstats();  
Dravid.showstats();  
Mohan.no_player = 10;
```

Public, Private and Protected members

Class members can either be declared in ‘public’,‘protected’ or in the ‘private’ sections of the class. But as one of the features of OOP is to prevent data from unrestricted access, the data members of the class are normally declared in the private section. The member functions that form the interface between the object and the program are declared in *public* section (otherwise these functions can not be called from the program). The member functions which may have been broken down further or those, which do not form a part of the interface, are declared in the *private* section of the class. By default all the members of the class are private. The third access specifier *protected* that is not used in the above example, pertains to the member functions of some new class that will be inherited from the base class. As far as non-member functions are concerned, private and protected are one and the same.

Constructors

Example:

```
#include <iostream.h>
class integer
{
private :
int i;
public :
void getdata( )
{
cout<<endl<<"Enter any integer";
cin>>i;
}
void setdata(intj)
{
i=j;
}
integer()           //zero argument constructor
{
}
integer(intj)      //one argument constructor
{
i=j;
```

```
}
```

```
void displaydata()
```

```
{
```

```
cout<<endl<<"value of i="<<i;
```

```
}
```

```
};
```

```
void main()
```

```
{
```

```
integer i1(100), i2, i3;
```

```
i1.displaydata();
```

```
i2.setdata(200);
```

```
i2.displaydata();
```

```
i3.getdata();
```

```
i3.displaydata();
```

```
}
```

The constructor is a special member function that allows us to set up values while defining the object, without the need to make a separate call to a member function. They have exactly the same name as the class of which they are members. No return type is used for constructors. Why not? Since the constructor is called automatically when an object is created, returning a value would not make sense.

In our program, the statement like integer i1(100), i2, i3; creates three objects of the type integer.

If you notice carefully you would find that there are two constructors with the same name integer(). Hence we call these constructors as overloaded constructors.

integer i2;	//calls zero-argument constructor
integer i1(100);	//calls one-argument constructor

Constructors

We've seen that a special member function-the constructor-is called automatically when an object is first created. Similarly, when an object is destroyed a function called destructor automatically gets called. A destructor has the same name as the constructor (which is same as the class name) but is preceded by a tilde. The following program shows destructor at work.

```
#include <iostream.h>
class example
{
private :
int data ;
public:
example()           // constructor (same name as class)
{
cout << endl << "inside the constructor" ;
}
~example()          // destructor (same name with tilde)
{
cout << endl << "Inside the destructor";
}
void main()
{
example e;
}
```

When the object e gets created the constructor gets called when control goes outside main() the object e gets destroyed.

Q1. Design and implement a class student_list having the following requirements:

- It stores details of two types of students “residential” and “non-residential”.
- Different fee structures are stored for different types of students.
- Fee of the student is calculated on the basis of types and are vastly different.
- You need to define the class hierarchy of the student supporting above.
- The classes should include necessary constructors, destructors and fee calculation functions (make suitable assumptions.)
- The student name should be represented using pointers.

Write the appropriate main() function. Make suitable assumptions, if any.

Ans. #include<iostream.h>

```
#include<conio.h>
```

```
class std_list
```

```
{
```

```
public :
```

```
char *name;
```

```
int length;
```

```
char res_type;
```

```
int total;
```

```
//default constructor
```

```
std_list()
```

```
{
```

```
length=20;
```

```
name=new char[length+1];
```

```
total=0;
```

```
}
```

```
//destructor
```

```
~std_list()
```

```
{
```

```
delete name;
```

```
}
```

```
void accept(); //function for accepting the detail of student
```

```
void display(); // function to display the student details
```

```
void fee(); // function to accept fee;
```

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

```
void std_list::fee()
```

```
{
```

```
if((res_type == 'y') || (res_type == 'Y'))
```

```
{
```

```
total=5000 + 5000 + 2000;
```

```
}
```

```
else
```

```
{
```

```
total=5000;
}
cout<<"total ==> "<<total;
}
void std_list::display()
{
cout<<"\n\n\nThe name of the student is \t";
cout<<name;
cout<<"\nThe residential type\t";
cout<<res_type;
cout<<"nyour total fees \t";
cout<<total;
getch();
}

void std_list::accept()
{
cout<<"\nEnter your name =>\t";
cin>>name;
cout.flush();
cout<< "Choose The residential type"<<endl;
cout<<"1 Residential include Room rent + Food Cost + tutionfees cost
==>12000"<<endl;
cout<<"2 Non Residential include Tutionfees cost ==>5000"<<endl;
cout<<"press 1 for Residential \n";
cout<<"press 2 for non Residential\t";
cin>>res_type;

if((res_type == 'y')||(res_type == 'Y'))
{
    cout<<"\n\nyou choose Residential option so your charges are as";
    cout<<"nyour Room Rent is 5000 \t";
    cout<<"nyour tuition Fees is 5000 \t";
    cout<<"nyour Food cost is 2000 \t";
}
```

```
    else
{
cout<<"Your Tuition fee is 5000 \t";
}
}

void main()
{
clrscr();
std_list ob;
ob.accept();
ob.fee();
ob.display();
}
```

Q2. Design and implement a class point stack using pointers. (The data representation must not be an array). Implement constructor(s), destructor and other suitable functions.

Ans. #include<iostream.h>

```
#include<conio.h>
#define max 100
class point_stack
{
int *arr;
int tos;
public:
public_stack()
{
tos=0;
cout<<"stack Initialised\n";
}
~point_stack()
{
cout<<"stack destroyed\n";
getch();
}
void push(int i)
```

```
{  
if(tos == max)  
{  
cout<<"Stack is full\n";  
return;  
}  
*arr=i;  
tos++;  
arr++;  
}  
int pop()  
{  
if(tos == 0)  
{  
cout<<"stack overflow\n";  
return 1;  
}  
arr--;  
return *arr;  
}  
};  
void main()  
{  
clrscr();  
point_stack a,b;  
a.push(1);  
b.push(2);  
a.push(3);  
b.push(4);  
cout<<a.pop()<<"\n";  
cout<<a.pop()<<"\n";  
cout<<b.pop()<<"\n";  
cout<<b.pop()<<"\n";  
getch();  
}
```

Q3. Design and implement the class stack implement constructor, destructor and other suitable function.

Ans. #include<iostream.h>

#include<stdio.h>

#include<conio.h>

#include<process.h>

```
class st
{
public :
int stack[30], top;
st()
{
    top=-1;
}
int pop(int stack[], int &top);
void push(int stack[], int &top, int val, int n);
void display(int stack[], int top);
};
//Insert in stack
void st::push(int stack[], int &top, int val, int n)
{
if(top < n)
{
    top++;
    stack[top]=val;
}
else
    cout<<"Stack is full";
}
//deletion in stack
int st::pop(int stack[], int &top)
{
int delvalue;
if(top>=0)
{
```

```
delvalue=stack[top];
top--;
return delvalue;
}
else
{
cout<<"\n Stack is empty";
return(-9999);
}
}

//Display in stack
void st::display(int stack[], int top)
{
int i;
if(top>=0)
{
cout<<"\n";
for(i=top; i>=0; i--)
{
cout<<stack[i]<<" ";
}
getch();
fflush(stdin);
}
}

void main()
{
st obj;
int val, choice;
do
{
//clrscr();
cout<<"\n      Menu    ";
cout<<"\n 1 ==> Push  2 ==> Pop";
```

```
cout<<"\n 3 ==> Display  4 ==>Quit ";
cout<<"Enter your choice";
cin>>choice;
switch(choice)
{
    case 1: cout<<"Enter the value to pushed";
    cin>>val;
    obj.push(obj.stack,obj.top,val,30);
    break;
    case 2: val=obj.pop(obj.stack, obj.top);
    if(val!= -9999)
        cout<<"\n the popped last value of stack is"<<val;
    break;
    case 3: obj.display(obj.stack, obj.top);
    break;
    case 4: exit(0);
    default:
        cout<<"wrong value";
}
}while(choice!=4);
```

Q4. Design and implement a class matrix having the following requirements:

- It represents matrix in two dimensional arrays. The maximum size of array can be 100×100 .
- There is a constructor that allocates the array of maximum size to the matrix object.
- You can store any matrix of size $m \times n$ ($m \leq 100$ and $n \leq 100$) in this class. There is an additional initialisation function to do so.
- It prints the stored matrix in row/column format.
- It adds and subtracts two matrices of same size.
- It multiplies two given matrices after checking the necessary time constraints.
- The class has provision for copy constructor and overloaded assignment operation.

Write the appropriate main() function. Make suitable assumptions, if any.

Ans. #include<conio.h>

#include<stdio.h>

#include<iostream.h>

#include<iomanip.h>

class matrix

{

private:

int maxrow, maxcol;

int *ptr;

public:

matrix(int r, int c)

{

maxrow=r;

maxcol=c;

ptr=new int[r *c];

}

void getmat()

{

int i, j, mat_off, temp;

cout<<endl<<"Enter Elements matrix:"<<endl;

for(i=0; i<maxrow; i++)

{

for(j=0; j<maxcol; j++)

{

mat_off=i*maxcol+j;

cin>>ptr[mat_off];

}

}

}

void printmat()

{

```
int i, j, mat_off;
for(i=0; i<maxrow; i++)
{
    cout<<endl;
    for(j=0; j<maxcol;j++)
    {
        mat_off=i*maxcol+j;
        cout<<setw(3)<<ptr[mat_off];
    }
}
}

int detmat()
{
    matrix q(maxrow-1, maxcol-1);
    int sign=1, sum=0, i, j, k, count;
    int newsize, newpos, pos, order;

    order=maxrow;
    if(order==1)
        return(ptr[0]);
    for(i=0; i<order; i++, sign*=-1)
    {
        for(j=1; j<order; j++)
        {
            for(k=0, count=0; k<order; k++)
            {
                if(k==i)
                    continue;

                pos=j*order+k;
                newpos=(j-1)*(order-1)+count;
                q.ptr[newpos]=ptr[pos];
                count++;
            }
        }
    }
}
```

```
}

sum=sum+ptr[i]*sign*q.detmat();

}

return (sum);

}
```

```
matrix operator + (matrix b)
{
matrix c(maxrow, maxcol);
int i, j, mat_off;

for(i=0; i<maxrow; i++)
{
for(j=0; j<maxcol; j++)
{
mat_off=i*maxcol+j;
c.ptr[mat_off]=ptr[mat_off]+b.ptr[mat_off];
}
}
return(c);
}
```

```
matrix operator * (matrix b)
{
matrix c(b.maxcol, maxrow);
int i, j, k, mat_off1, mat_off2, mat_off3;

for(i=0; i<c.maxrow; i++)
{
for(j=0; j<c.maxcol; j++)
{
mat_off3=i*c.maxcol+j;
c.ptr[mat_off3]=0;
for(k=0; k<b.maxrow; k++)
{
mat_off2=k*b.maxcol+j;
```

```
mat_off1=i*maxcol+k;
c.ptr[mat_off3]+=ptr[mat_off1]*b.ptr[mat_off2];
}
}
}
return(c);
}

int operator==(matrix b)
{
int i, j, mat_off;
if(maxrow!=b.maxrow || maxcol!=b.maxcol)
return(0);

for(i=0; i<maxrow; i++)
{
for(j=0; j<maxcol; j++)
{
mat_off=i*maxcol+j;
if(ptr[mat_off]!=b.ptr[mat_off])
return 0;
}
return(0);
}
return(1);
};

void main()
{
int rowa, cola, rowb, colb;

cout<<endl<<"Enter dimensions of A";
cin>>rowa>>cola;
```

```
matrix a(rowa, cola);
a.getmat();

cout<<endl<<"Enter dimensions of matrix B";
cin>>rowb>>colb;
matrix b(rowb,colb);
b.getmat();

matrix c(rowa,cola);
c=a+b;
cout<<endl<<"The sum of two matrix = ";
c.printmat();

matrix d(rowa, colb);
d=a*b;
cout<<endl<<"The product of two matrices = ";
d.printmat();

cout<<endl<<"Determinant of matrix a = "<<a.detmat();
if(a==b)
cout<<endl<<"a & b are equal";
else
cout<<endl<<"a & b are equal";
}
```

- Q5.** Design and implement a class linked-queue which is implemented using pointers. The class includes AddQ, DeleteQ and Empty operations (standard operations of Data Structure Queue). Make suitable assumptions, if any.

Ans. #include<iostream.h>

```
#include <stdio.h>
#include<ctype.h>
#define FALSE 0
#define NULL 0
struct listelement
{
    int    dataitem;
    struct listelement *link;
```

```
};

class queue :public listelement
{
public:
listelement * lp,* tempp;
void Menu (int *choice);
listelement * AddQ (listelement * listpointer, int data);
listelement * DeleteQ (listelement * listpointer);
void PrintQueue (listelement * listpointer);
void ClearQueue (listelement * listpointer);
};

void main ()
{
listelement *listpointer;
queue x;
int data, choice;
listpointer = NULL;
do {
x.Menu (&choice);
switch (choice) {
case 1:
cout<< ("Enter data item value to add");
cin>>data;
listpointer = x.AddQ (listpointer, data);
break;
case 2:
if (listpointer == NULL)
cout<<"Queue empty!\n";
else
listpointer = x.DeleteQ (listpointer);
break;
case 3:
x.PrintQueue (listpointer);
break;
default:
```

```
cout<<"Invalid menu choice - try again\n";
break;
}
} while (choice != 4);
x.ClearQueue (listpointer);
}
/* main */
void queue::Menu (int *choice)
{
char local;
cout<<"\nEnter\t1 to add item,\n\t2 to remove item\n\t
\t3 to print queue\n\t4 to quit\n";
do {
local = getchar ();
if ((isdigit (local) == FALSE) && (local != '\n'))
{
cout<< "\nyou must enter an integer.\n";
cout<< "Enter 1 to add, 2 to remove, 3 to print, 4 to quit\n";
}
} while (isdigit ((unsigned char) local) == FALSE);
*choice = (int) local - '0';
}
listelement * queue:: AddQ (listelement * listpointer, int data)
{
lp = listpointer;
if (listpointer != NULL)
{
while (listpointer -> link != NULL)
listpointer = listpointer -> link;
listpointer -> link = new listelement;
listpointer = listpointer -> link;
listpointer -> link = NULL;
listpointer -> dataitem = data;
return lp;
```

```
}

else {
listpointer = new listelement;
listpointer -> link = NULL;
listpointer -> dataitem = data;
return listpointer;
}
}

istelement * queue ::DeleteQ (listelement * listpointer)
{
cout<<"Element removed is"<<listpointer -> dataitem;
tempp = listpointer -> link;
free (listpointer);
return tempp;
}

void queue ::PrintQueue (listelement * listpointer)
{
if (listpointer == NULL)
cout<<"queue is empty!\n";
else
while (listpointer != NULL) {
cout<<listpointer -> dataitem;
listpointer = listpointer -> link;
}
cout<<"\n";
}

void queue::ClearQueue (listelement * listpointer)
{
while (listpointer != NULL) {
listpointer = DeleteQ (listpointer);
}
}
```

Q6. Design and implement a class linked having constructors/destructors and operations of adding an item for searching a value and inserting a value after a searched item. (Please assume singly link list). Make suitable assumptions, if any.

Ans. #include<iostream.h>

```
#include<conio.h>
```

```
class Linked
```

```
{
```

```
public:
```

```
int info;
```

```
Linked *next;
```

```
Linked *first;
```

```
Linked()
```

```
{
```

```
first=0;
```

```
next='\0';
```

```
}
```

```
~Linked()
```

```
{
```

```
delete next;
```

```
delete first;
```

```
}
```

```
void addnode(void);
```

```
void insnode(void);
```

```
void display(void);
```

```
};
```

```
void Linked::addnode()
```

```
{
```

```
Linked *curr;
```

```
static Linked *last=0;
```

```
curr=new Linked();
```

```
cout<<"Enter number:";
```

```
cin>>curr->info;
```

```
curr->next='\0';
```

```
if (last)
```

```
last->next=curr;
else
first=curr;
last=curr;
}
void Linked::display()
{
Linked *l;
l=first;
do
{
cout<<l->info<<"";
l=l->next;
}
while(l!='\0');
}
void Linked::insnode()
{
Linked *curr,*last,*l,*n;
int no,val,flag=0;
l=first;
cout<<"Enter value which you want to search :";
cin>>no;
cout<<"Enter value which you want to insert :";
cin>>val;
}
while(l!="\0")
if(l->info==no)
{
curr=l;
flag=1;
break;
}
else
{
l=l->next;
```

```
}

}

if(flag==0)
{
    cout<<"not found";
    getch();
    return ;
}

else
{
    n=new Linked();
    n->info=val;
    n->next='\0';
    last=l->next;
    curr->next=n;
    n->next=last;
}

void main()
{
    clrscr();
    Linked ob;
    int ch,f;
    do
    {
        clrscr();
        cout<<"1.Add Node\n";
        cout<<"2.Insert Node\n";
        cout<<"3.Display and Exit\n";
        cout<<"Enter choice:";

        cin>>ch;
        switch(ch)
        {
            case 1:ob.addnode();
            break;
```

```
case 2:ob.insnode();
break;
case 3:ob.display();
getch();
break;
}
}while(ch!=3);
}
```

Q7. Design and implement a table class in C++ having the following requirements.

The table may have only floating point values.

- The sum of a row or a column can be calculated with the help of overloaded sum function.**
- There is a copy constructor and other constructor for the table.**
- There are functions to find largest and the smallest values from the table and its positions in the table.**
- It allows addition and subtraction of two tables having similar size.**

Ans. #include<iostream.h>

```
#include<conio.h>
class table
{
public:
float a[10];
int size, i;
float sum();
void accept();
float min();
float max();
void display();
friend table operator +(table x, table y);
table()
{
size=0;
i=0;
```

```
}

table(table &x)           //copy constructor
{
for(i=0; i<x.size; i++)
{
a[i]=x.a[i];
}
}

//function to overload the operator
table operator +(table x, table y)
{
table temp;
if(x.size==y.size)
{
for(int i=0; i<x.size; i++)
temp.a[i]=x.a[i] + y.a[i];
return temp;
}
else
cout<<"size doesn't match";
}

//function to accept the array
void table::accept()
{
cout<<"\nEnter the number of element you want to enter";
cin>>size;
for(int i=0; i<size; i++)
cin>>a[i];
}

//function for sum of row
float table::sum()
{
```

```
float sum=0.0;
for(int i=0; i<size; i++)
{
    sum+=a[i];
}
return sum;
}

//function to find minimum in row
float table::min()
{
    float m=a[i-1],temp;
    for(int j=0; j<size; j++)
    {
        if(m>a[j])
        {
            temp=m;
            m=a[j];
            a[j]=temp;
        }
    }
    return m;
}

//function to find maximum in row
float table::max()
{
    float m=a[i-1],temp;
    for(int j=0; j<size; j++)
    {
        if(m<a[j])
        {
            temp=m;
            m=a[j];
            a[j]=temp;
        }
    }
}
```

```
}

return m;
}

void table::display()
{
for(int i=0; i<size; i++)
cout<<a[i];
}

void main()
{
float w;
char ch;
table x,y,z;
clrscr();

x.accept();
w=x.min();
cout<<"\nMinimum value i"<<w;

w=x.max();
cout<<"\nMaximum value i"<<w;
w=x.sum();
cout<<"\nAfter the row addition value is"<<w;

cout<<"\nDo you want to add another vector to this vector";
cin>>ch;
if(ch=='y' | ch=='Y')
{
y.accept();
z=x+y;
}
cout<<"\nAfter the table row addition value is";
//after the table addition value is stored in z';
z.display();
}
```

Q8. Write a program in C++ for merging two strings into one. You must design the program using pointers. Strings should be represented using pointers.

Or

Write a C++ program having the following functionality:

- The program have a class string in it.
- The string class have the data member as pointer to character.
- The class has overloaded constructor and copy constructor.
- The class have overloaded + operator which concatenates two strings.
- The class have a destructor.
- The class have functions for string comparison.

Also write the appropriate main function that demonstrate the complete functionality of the class string.

Ans. # include<string.h>

```
# include<iostream.h>
class string
{
char *p;
int len;
public :
string( ) {len = 0; p = 0;} //create null string
string ( const char * s) ; // create string from array s
string (const string & s) ; //copy constructor
~string () {delete p;} // destructor
friend string operator + (const string & s, const string & t);
friend int operator <= (const string & s, const string & t);
friend void show (const string s) ;
string &operator=(string &c)
{strcpy(p, c.p);
return *this; }
};

string :: string (const char * s)
{
```

```
len = strlen (s);
p = new char [len + 1];
strcpy (p, s);
}
string :: string (const string & s)
{
len = s.len;
p = new char[len + 1];
strcpy (p, s.p);
{
//.....OVERLOADING + operator .....
string operator + (const string & s, const string & t)
{
string temp;
temp.len = s.len + t.len;
temp.p = new char [temp.len + 1];
strcpy (temp.p, s.p);
strcat (temp.p, t.p);
return (temp);
}
//.....OVERLOADING <= operator .....
Int operator <= (const string & s, const string & t)
{
int m = strlen (s.p);
int n = strlen (t.p);
if (m<= n) return (1);
else return (0);
}
void show (const string s)
{
cout << s.p;
}
main ( )
{
string s1 = "New";
```

```
string s2 = "York";
string s3 = "Delhi"
string t1, t2, t3, t4;

t1 = s1;
t2 = s2;
t3 = s1+s2;
t4 = s1 + s3 ;

cout << "\n" t1 = "; show (t1);
cout << "\n" t2 = "; show (t2);
cout << "\n";
cout << "\n" t3 = "; show (t3);
cout << "\n" t4 = "; show (t4);
cout << "\n\n";

if (t3 <= t4)
{
    show (t3);
    cout << "smaller than";
    show (t4);
    cout << "\n";
}
else
{
    show (t4);
    cout << "smaller than";
    show (t3) ;
    cout << "\n";
}
```

||||||||||||||||||||||||||||<Program>||||||||||||||||||||||||

The following is the output of Program

t1 = New

t2 = York

t3 = New York

t4 = New Delhi

New York smaller than New Delhi.

Q9. Design and implement a class 'Itemlist' having following requirements:

- It is a linear array of size N
- Each element has Itemcode, Item Price and Quantity
- Item to be appended to the list
- Delete item from the list with the help of item code
- Print total value of stock

Ans. #include<iostream.h>

```
#include<conio.h>
class Itemlist
{
private:
struct node
{
int itemcode;
float itemprice;
int quantity;
node*link;
}*p;
public:
Itemlist();
void append(int, float, int);
void del(int);
void total();
//~Itemlist();
};
```

```
Itemlist::Itemlist()
{
p=NULL;
}
```

```
void Itemlist::append(int code, float price, int qua)
{
node*q,*t;
if(p==NULL)
{
//create first node
p=new node;
p->itemcode=code;
p->itemprice=price;
p->quantity=qua;
p->link=NULL;
}
else
{
//go to last node
q=p;
while(q->link!=NULL)
q=q->link;
//add node at the end
t=new node;
t->itemcode=code;
t->itemprice=price;
t->quantity=qua;
q->link=t;
t->link=NULL;
}
}
void Itemlist::del(int code)
{
node*q,*r;
q=p;
//if node to be deleted is first node
if(q->itemcode==code)
{
```

```
p=q->link;
delete q;
return;
}
//traverse
r=q;
while(q!=NULL)
{
if(q->itemcode==code)
{r->link=q->link;
delete q;
return;
}
r=q;
q=q->link;
}
cout<<"\n\nElement "<<code<< "not found\n";
}
void Itemlist::total()
{
node*q;
float t=0;
for(q=p;q!=NULL;q=q->link)
t=t+q->itemprice*q->quantity;
cout<<"\n total value of stock is:"<<t;}
void main()
{
clrscr();
Itemlist l;
cout<<"\n Value of stock:\n";
l.total();
l.append(001, 1200.5, 50);
l.append(002, 1400.2, 20);
l.append(003, 1500.5, 100);
```

```
cout<<"\n Value of stock :\n";
l.total();
l.del(002);
l.del(004);                                //Element 004 not present message
cout<<"\n Value of stock now";
l.total();
getch();
}
```

Q10. Design and implement a class 'binsearch' for a binary search tree. It includes search, remove and add options. Make suitable assumption.

Ans. #include<iostream.h>

```
#include<conio.h>
#define TRUE 1
#define FALSE 0
class binsearch
{
private:
struct node
{
node*left;
int item;
node*right;
}*start;
public:
binsearch();
~binsearch();
void traverse();
void search(int, int&, node*&, node*&);
void add(int);void delend(node*);
void del(int);void inorder(node*);
};
binsearch::binsearch()
{
start=NULL;
```

```
}

binsearch::~binsearch()
{
    delend(start);
}

void binsearch::search(int n, int&found, node*&parent, node*&x)
{
    node*q;
    found=FALSE;
    parent=NULL;
    if(start==NULL)
        return;
    q=start;
    while(q!=NULL)
    {
        if(q->item==n)
        {
            found=TRUE;
            x=q;
            return;
        }
        if(q->item>n)
        {
            parent=q;
            q=q->left;
        }
        else
        {
            parent=q;
            q=q->right;
        }
    }
}

void binsearch::add(int n)
```

```
{  
int found;  
node*t,*parent,*x;  
search(n, found, parent, x);  
if(found==TRUE)  
cout<<"\n"<<"such a node already exists";  
else  
{  
t=new node;  
t->item=n;  
t->left=NULL;  
t->right=NULL;  
if(parent==NULL)  
start=t;  
else  
parent->item>n?parent->left=t:parent->right=t;  
}  
}  
void binsearch::delend(node*q)  
{  
if(q!=NULL)  
{  
delend(q->left);  
del(q->item);  
delend(q->right);  
}  
}  
void binsearch::del(int num)  
{  
int found;  
node*parent,*x,*xsucc;  
/* if tree is empty */  
if(start==NULL)  
{
```

```
cout<<endl<<"Tree is empty";
return;
}
parent=x=NULL;
//call search function to find node to be deleted
search(num, found, parent, x);
//if not found
if(found==FALSE)
{
cout<<endl<<"Node not found";
return;
}
//if node to be deleted has two children
if(x->left!=NULL&&x->right!=NULL)
{
parent=x;
xsucc=x->right;
while(xsucc->left!=NULL)
{
parent=xsucc;
xsucc=xsucc->left;
}
x->item=xsucc->item;
x=xsucc;
}
//if node to be deleted has no child
if(x->left==NULL&&x->right==NULL)
{
if(parent->right==x)
parent->right=NULL;
else
parent->left=NULL;
delete x;
return;
```

```
}

//if it has left child only
if(x->left!=NULL&&x->right==NULL)
{
    if(parent->left==x)
        parent->left=x->left;
    else
        parent->right=x->left;
    delete x;
    return;
}

//if it has right child only
if(x->left==NULL && x->right!=NULL)
{
    if(parent->left==x)
        parent->left=x->right;
    else
        parent->right=x->right;
    delete x;
    return;
}

void binsearch::traverse()
{
    inorder(start);
}

void binsearch::inorder(node*q)
{
    if(q!=NULL)
    {
        inorder(q->left);
        cout<<endl<<q->item;
        inorder(q->right);
    }
}
```

```
}
```

```
void main()
```

```
{
```

```
binsearch t;
```

```
int i, num;
```

```
for(i=0; i<=6; i++)
```

```
{
```

```
cout<<"\n"<<"Enter data to be inserted";
```

```
cin>>num;
```

```
t.add(num);
```

```
}
```

```
t.traverse();
```

```
cout<<endl<<"Enter value to be deleted";
```

```
cin>>num;
```

```
t.del(num);
```

```
cout<<endl;
```

```
t.traverse();
```

```
getch();
```

```
}
```

OOO

GET PUBLISHED, BECOME FAMOUS

GULLYBABA PUBLISHING HOUSE (P) LTD.

**Complete Publishing Assistance:- Content Writing,
Proof Reading, Editing, Designing, Printing,
Translation and Marketing.**

9312235086, 9350849407, 27387998, 27384836

4

Introduction to Functions

Features of OOP make C++ such a strong language. In this session, we will continue exploring some other features, which make this language more powerful.

Function Prototypes

A function prototype is a declaration that defines both: the argument passed to the function and the type of value returned by the function. If we were to call a function `fool()` which receives a float and an int as arguments and returns a double value its prototype would look like this:

```
double fool(float, int);
```

The C++ compiler uses this prototype to ensure that the types of the arguments you pass in a function call are same as those mentioned in the prototype. If there is a mismatch the compiler points it out immediately. This is known as strong type checking; something which C lacks.

Without strong type checking, it is easier to pass illegal values to functions. For example, a non-prototype function would allow you to pass an int to a pointer variable, or a float to a long int. The C++ compiler would immediately report such types of errors. In C++ prototypes do more than making sure that actual arguments (those used in calling function) and formal arguments (those used in called function) match in number, order and type. C++ internally generates names for functions, including the argument type information. This information is used when several functions have same names.

Remember that all function in C++ must be prototyped. This means every function must have its argument list declared, and the actual definition of a function must exactly match its prototypes in the number, order and types of parameters.

Call by reference, Call by value

Call by reference: A reference provides an alias, i.e. alternate name for the variable. It is useful when the values of the original variables are to be changed using a function. Reference arguments are indicated by an ampersand preceding the argument, e.g. int & a;

E.g.

```
#include<iostream.h>
#include<conio.h>
void Swap (int &, int &)
void main ()
{
int a,b;
Cout<<"Enter any two values for a and b" <<endl;
Cin>>a;
Cin>>b;
Swap(a, b);
Cout<<"The two entered numbers are"<<"a"<<a<<" " <<"-"
and"<<"b"<<b<<endl. ;
}
void Swap (int & a, int & b)
{
int c:
c = a;
a = b;
b = c;
}
```

The above program uses *call by reference* method for invoking a function.

Call by value: In this method the called function creates a new set of variables and copies the value of arguments into them. The function does not have access to the original variable(actual parameters) and can only work on the copies of values it created. It is useful when the original values are not to be modified. In fact, call by value method offers insurance that the function cannot harm the original value.

The following example explain this:

```
Calling()
{ int passit = 90;
Called (Passit);
Cout<<passit; //there will be no change in passit value, remains 90.
}
void Called (int receiveit)
{
receiveit = 0;
}
```

In call by references arguments that are objects, are passed by references to the called method. Any change made to the object by the called method is reflected in the calling method:

```
Calling ()
{
int passit=90;
Cout<<passit //passit changes here to 0.
}
Called (int &receiveit)
{
receiveit=0;
}
```

In a call by reference, the memory address, i.e. reference of the argument's variable is passed to the called method.

When the called ()method changes the value of the argument, the change is reflected in the calling() method.

The output-generated from the above program is "changed".

Static Class Members

All the objects of the class have different data members but invoke the same member functions. However, there is an exception to this rule. If the data member is declared with the keyword *static*, then only one such data item is created for the entire class, no matter how many objects it has. Static data members are useful, if all objects of a class must share a common data item. Whereas, the visibility of this data item remains same, the duration of this variable is for entire lifetime of the program.

For example, such a variable can be particularly useful if an object requires to know how many objects of its kind exist.

```
class counter
{
public :
counter();
int getcount();
private:
static int count;
};

counter::counter()
{
count++;
}

int counter::getcount()
{
return (count);
}

int counter :: count = 0;           //Initialisation of static member

void main()
{
counter c1,c2;
cout << " Count = " << c1.getcount() << endl;
cout << " Count = " << c2.getcount() << endl;

counter c3;
cout << " Count = " << c3.getcount() << endl;

counter c4,c5;
cout << " Count = " << c4.getcount() << endl;
cout << " Count = " << c5.getcount() << endl;
}
```

Output:

Count = 2 //not 1 because 2 objects are already created.
Count = 2

```
Count = 3
```

```
Count = 5
```

```
Count = 5
```

In the above example, the class counter demonstrates the use of static data members. It contains just one data member count. *Notice the initialisation of this static class member.* For some compilers it is mandatory. Even though the data member is in the private section it can be accessed in this case as a global variable directly, but has to be preceded by the name of the class and the scope resolution operator. This example contains a constructor to increment this variable. Similarly, there can be a destructor to decrement it.

You can use these to generate register numbers for student objects from a student class. Whenever an object is created, it will be automatically assigned a register number if the register number is a static variable and a constructor is used to write an equation to generate separate register numbers in some order. You could initialise the variable to give the first register number and then use this in the constructor for further operations.

Static Member Functions

All the objects of the class share static data members of a class. The example above demonstrates how to keep track of all the objects of a class which are in existence. However, this function uses existing objects to invoke a member function getcount(), which returns the value of the static data member. *What if the programme does not want to use objects to invoke this function and still the programme would like to know how many objects have been created? If there is no object how the member function is invoked?* Further, as can be seen from the previous output, the number of objects (count) remains same at a given instance no matter which object is used to invoke the member function. In fact, the use of existing objects, like in the above example, is not an effective way to access the value of the static data member. A specific object should not be used to refer to this member, since it does not belong to that object; it belongs to the entire class. C++ gives a facility to define static function members, for the same. That is, to invoke such a function, an object is not required. It can be invoked with the name of the class. The programme given below illustrates its use.

```
class counter
{
public :
    counter ();
    static int getcount();

private:
```

```
static int count;  
};  
  
counter::counter ()  
{  
    count++;  
}  
  
int counter::getcount()  
{  
    return ( count );  
}  
  
int counter :: count = 0; //Initialisation of static member  
  
void main()  
{  
    counter c1,c2;  
  
    cout << " Count = " << counter :: getcount() << endl;  
    cout << " Count = " << counter :: getcount() << endl;  
  
    counter c3;  
  
    cout << " Count = " << counter :: getcount() << endl;  
  
    counter c4,c5;  
  
    cout << " Count = " << counter :: getcount() << endl;  
    cout << " Count = " << counter :: getcount() << endl;  
}
```

Output:

Count = 2
Count = 2
Count = 3
Count = 5
Count = 5

Q1. What is the need of friend functions? Describe the usage of friend functions with the help of an example.

Or

What is the purpose of using friend functions? Describe with an example. Can a class be made friend of other class? Justify your answer.

Ans. One of the main features of OOP is information hiding. A class encapsulates data and methods to operate on that data in a single unit. The data from the class can be accessed only through member functions of the class. This restricted access not only hides the implementation details of the methods and the data structure, it also saves the data from any possible misuse, accidental or otherwise. However, the concept of data encapsulation sometimes takes information hiding too far. There are situations where a rigid and controlled access leads to inconvenience and hardships.

For instance, consider a case where a function is required to operate on object of two different classes. This function cannot be made a member of both the classes. What can be done is that a function can be defined outside both the classes and made to operate on both. That is, a function not belonging to either, but able to access both. Such a function is called as a friend function. In other words, a friend function is a nonmember function, which has access to a class's private members. It is like allowing access to one's personal belongings to a friend.

Using a Friend

Using a friend function is quite simple. The following example defines a friend function to access members of two classes.

```
class Bclass; // Forward Declaration
```

```
class Aclass
```

```
{
```

```
public :
```

```
    Aclass(int v)
```

```
{
```

```
    Avar = v;
```

```
}
```

```
friend int addup(Aclass &ac, Bclass &bc);
```

```
private :
```

```
int Avar;
```

```
};

class Bclass
{
public :
Bclass(int v)
{
Bvar = v;
}

friend int addup(Aclass &ac, Bclass &bc);
private :
int Bvar;
};

int addup(Aclass &ac, Bclass &bc)
{
return( ac.Avar + bc.Bvar);
}

void main()
{
Aclass aobj;
Bclass bobj;

int total;
total = addup(aobj,bobj);
}
```

The program defines two classes - Aclass and Bclass. It also has constructors for these classes. A friend function, addup(), is declared in the definition of both the classes, although it does not belong to either. This friend function returns the sum of the two private members of the classes. Notice, that this function can directly access the private members of the classes. To access the private members, the name of the member has to be prefixed with the name of the object, along with the dot operator.

The first line in the program is called forward declaration. This is required to inform the compiler that the definition for class Bclass comes after class Aclass and therefore it will not show any error on encountering the object of Bclass in the declaration of the friend function. Since it does not belong to both the classes, while defining it outside we do not give any scope resolution and we do not invoke it with the help of any object. Also, the keyword friend is just written before the declaration and not used while defining.

Sometimes friend functions can be avoided using inheritance and they are preferred. Excessive use of friend over many classes suggests a poorly designed program structure. But sometimes they are unavoidable.

Granting friendship to another class

Yes, a class can be made friend of other class. To grant friendship to another class, write the keyword followed by the class name. The keyword class is optional. Note that this declaration also implies a forward declaration of the class to which the friendship is being granted. The implication of this declaration is that all of the member functions of the friend class are friend functions of the class that bestows the friendship.

```
class Aclass
{
public :
    :
    :
    :

friend class Bclass; // Friend declaration.

private :
int Avar;
};

class Bclass
{
public :
    :
    :

void fn1(Aclass ac)
{
```

```
Bvar = ac. Avar;           //Avar can be accessed.  
}  
private :  
int Bvar;  
};  
  
void main()  
{  
Aclass aobj;  
Bclass bobj;  
bobj.fn1(aobj);  
}
```

The program declares class Bclass to be a friend of Aclass. It means that all member functions of Bclass have been granted direct access to all member functions of Aclass.

Inline Functions

Imagine a C language program, which reads disk records containing employee information. If this is a payroll application each employee record data is probably processed via a series of function calls. These function calls could be to print the entity data, to compute the salary, to compute the taxes to be withheld etc.. Each one of these calls inherently contains overhead that must be part of your overall program. In other words it takes code, space and time to push actual arguments onto the stack, call the function, push some return value onto the stack and finally pop all of those values .

C++ provides the *inline functions*, a mechanism by which these explicit function calls can be avoided.

“An inline function by definition is a function whose code gets substituted in place of the actual call to that function”.

Whenever the compiler encounters a call to that function it merely replaces it with the code itself thereby saving you all of the overhead. Such a function can be either a member of a class or a global function. Inline functions work best when they are small, straightforward bodies of code that are not called from too many different places within your program. The time saved will increase with the increase in number of calls.

Even if you request that the compiler make a function into an inline function, the compiler may or may not honor that request. It depends on the code of the function. For example, the compiler will not inline any

function that contains a loop, static data member, or aggregate initialiser list. In such cases, a warning message will be issued.

The disadvantage with inline functions is that if the code itself ever needs to be modified, then all programs using these functions would then have to be recompiled. Furthermore, an inline function is a violation of implementation hiding.

How to write a global inline function

First, let's get away from member functions for a moment and consider a global function. To make a request that this function be inline :

- Precede the return type of the function with the keyword `inline`.
- Write the function body (the definition, not just the declaration) before any calls to that function.

Here is a program that uses an inline function to compute and return the absolute value of its input argument.

```
# include <stdio.h>
inline int abs(int x)
{
    return x <0? -x : x ;
}

void main( )
{
    for (int i=-2 ; i<2 ; ++i)
    {
        int value = abs(i) ;
        printf ("Absolute value of %d = %d\n",i, value);
    }
}
```

The output is:

```
Absolute value of -2 =+2
Absolute value of -1 =+1
Absolute value of +0 =+0
Absolute value of +1 =+1
```

When the call to the `abs()` function is encountered, the compiler, instead of making a function call, generates this assembly code.

Q2. What are the usages of “const” keyword in functions and arguments? Describe with the help of example.

Or

What is the meaning of having “const and classname” as a parameter in a function? Describe.

Ans. We have met the const keyword already as a way of defining a constant value which can be used in place of a raw number within your program. The const keyword has several other uses in C++, particularly in relation to function definitions. Consider the function definition:

```
int ConstArgs(const int &Arg1, const float &Arg2) {...}
```

The const keyword before each argument tells the compiler that although these arguments are being passed by reference, they are not allowed to be modified within the function body. They become, in essence, ‘read-only objects: their values may be accessed but not altered. Thus, the following function definition would give rise to a compile-time error:

```
int ConstArgs(const int &Arg1, const float &Arg2)
{
    Arg1 *= 2;
}
```

A general rule is: if you agree with those people who tell you to pass arguments by reference whenever possible, you should carefully consider whether each variable should be changed within the function to which it is being passed. If not, you should ensure that it is declared as a const parameter in the function definition. Thus is purely a safeguard against subtle errors; if your program is written correctly, it won’t make any difference whether the const is there or not, but if you make a mistake and attempt to alter a variable within a function where it shouldn’t be altered, the prefix const just might save you hours of debugging. Another use of the const keyword is *after* a function declaration in a class.

Constant member functions are used in classes to protect changes to class’s data.

For example, in the class definition:

```
Class ConstClass
{
    private:
        declarations
    public:
        int ConstFunc(args) const;
```

The const after the declaration of ConstFunc() states that ConstFunc() is not allowed to change any fields of the object that calls it. For example, with the code fragment:

```
ConstClass Object;
```

```
Object.ConstFunc( );
```

The call to ConstFunc() must not alter any of the values of any of the data fields of Object. This is another safety feature in C++ designed to help you catch programming errors early. If you declare a function as const in this fashion and then inadvertently insert some statements into the definition of that function that alter some of the fields of the object calling the function, the compiler will flag an error. It is therefore a good idea to think carefully about whether or not a function should be allowed to alter data fields and, if not, declare it as const.

Q3. What is a “Reference” in C++?

Ans. A reference is an alternate name of object, and you can use it just as you would use the object itself. You can think of a reference as the address of an item except that unlike a pointer, a reference is not a real variable- a reference is initialised when it is defined, and you cannot modify its value later on.

Q4. Where is a “Reference” in C++ used? Explain with the help of examples.

Ans. The syntax of a reference mimics that of a pointer except that a reference needs an ampersand (&) where that pointer declaration has an asterisk (*). Thus,

```
int *ptr; //An uninitialised pointer to integer
```

Defines an int pointer, whereas the following is a reference to an int variable named I:

```
int &r=I; //Reference to I (an int variable)
```

As a practical example of the use of reference, here is the swap_int function with arguments passed by reference:

```
void swap_int(int &a, int &b)
{
    int temp;
    temp=a;
    a=b;
    b=temp;
}
```

Compare this version of the function with the one that used pointer. You can see that version that uses references looks cleaner. You no longer have to dereference pointer in each expression. It is also simple to use the new version of the function, because you do not have to provide as arguments, the address of the integers being swapped. Instead, you call the function as if the arguments were being passed by value:

```
void swap_int(int&, int&);           // prototype of function
int x=2; y=3;
swap_int(x, y);                      // Now x=3 and y=2
```

The compiler knows from the function's prototype that it has to pass references to the variables x and y.

Q5. Why do you need references in C++? How can a reference be used in parameter passing? Describe with the help of an example.

Or

What is a reference variable? What is the usage of references in parameter passing? Why are references needed when call by value exists? What is the purpose of "const" keyword with respect to references?

Ans. Need/Usage of Passing By Reference: This sounds like a good reason to always pass by reference, but if you are awake you should spot a danger here. Remember that one difference between that two methods of passing a variable is that in one case, any changes made on the variable within the function have no effect on the calling routine, while in the other case, the changes *do* affect the variable in the calling routine. If you pass a variable by reference and inadvertently make changes to it within the function that you don't. Fortunately, C++ has a mechanism that allows you to protect against such an error, even if you pass the variable by reference.

Use of Reference in parameter passing

C++ provides a second way of passing a variable to a function called *passing by reference*. The syntax for this is:

```
int RefFunction (int &Arg1, float &Arg2);
```

Note that each argument is prefixed by an ampersand. Such a function is called in exactly the same way as one where the arguments are passed by value:

```
ReturnedInt = RefFunction (Int, Float2);
```

The difference between IntFunction and RefFunction is that the *addresses* of the arguments are passed in RefFunction, and not the actual values. As such, no extra memory is reserved within the function, since no copy of the arguments need to be made. This variable Arg1 within the

function and the variable Int1 passed to it are identical: they share the same address in memory. In this case, any change made to Arg1 within the function will also affect Int1 the calling routine.

Example:

```
#include<iostream.h>
void main()
{
    void change(int&);
    int a=10;
    cout<<"Starting value of a is "<<a;
    change(a);
    cout<<"The changed value of a is "<<a;
    getch();
}
void change(int &x)
{
    x=x+2;
}
```

Need of references when call by value exists

(1) Passing By Value: Dynamic memory allocation using the new operator, as in the last section, is only one instance in which dynamic allocation occurs. The other instance is more suitable, and happens more often than you might expect. This occurs when data are passed to functions as arguments. Now review what happens when a function is called.

A function such as: *int IntFunction (int Arg1, float Arg2) {.....}* expects two arguments, one int and one float, and returns int. Such a function is called from another function with a statement like: *eturnedInt = IntFunction (Int1, Float2);* where Int1 and ReturnedInt have been declared previously and ints, and Float2 as a float. The arguments Arg1 and Arg2 in the function's argument list are variable local to the function. The space for them is allocated when the function is called and released when the function returns. The space for the argument Arg1, for example, is allocated *dynamically* when the function is called, the value passed to that argument (Int1) is copied to the space just allocated, the variable is used within the function in whatever manner the function's statements dictate, and that the memory is deallocated (freed up for later use by the same program or any other program running on the same machine) when the function terminates, and control passes back to the routine that called the function in the first place. This method of passing arguments to a function is known

as *passing by value*, because only the *value* of the variable is passed to the function, not the address of the variable. Any changes made to the local copy of the value within the function do not affect the original variable in the calling routine.

(2) Passing by reference: When we pass a variable to a function as such, i.e. by value we do not see the changes in the variable's value. That is why we must use & symbol, i.e. pass by reference.

Purpose of “const” keyword with respect to references:

We use const keyword with reference whenever we want to avoid passing variable to a function by value, i.e. avoid making copy and we want that the called function should not change the value of variable & only use it.

Example:

```
#include<iostream.h>
void main()
{
    int change(const int&);

    int a=10;
    cout<<"Starting value of a is "<<a;
    int x=change(a);
    cout<<"The changed value of a is "<<x;
    getch();
}

int change(const int &x)
{
    int r=x+2;                                //cannot use x=x+2
    return r;
}
```

Q6. “The access rules violated by friend functions and pointers”.
Comment on the above statement. Give reasons in support or against the above statement.

Ans. Friend function has ability that can access the private data of the class. Pointer can access the address of any part of data. Again where is the security. So by both pointer and function **access rules violation** is done in C++. For example,

```
class x
{
```

```
int data;
void add();
public:
friend void add(x obj; y obj2);
}

void add(x obj; obj2)
{
int sum;
sum = obj.data + obj.2data;
}

{
class y
int 2data;
public:
friend void add( x obj; y obj2);
}

main()
{
x obj2;
y obj3;
obj2.data = 10;
obj3.2data = 20;
data (obj2, obj3)
}
```

- Q7. Explain how pointer to functions can be declared in C++. Under what conditions can two pointer variables be added, subtracted and compared?**

Ans. Pointer to a function is made (like pointer to an array is) by using name of function.

If we have int a[10];

pointer to it is

```
int*p=a;
```

For a function as int display()

we will have pointer fn_as

int(*fn)();

fn->display;

now we can call function display by using pointer fn.

as

(*fn());

Note: pointer to function is used in manipulators like

endl.

(1) Pointers can never be added.

(2) They can be subtracted if both pointers point to same array

e.g. int arr[]={10, 20, 30, 45, 67, 56, 44};

int*i, *j;

i=&arr[1];

j=&arr[5];

cout<<j-i

(3) They can be compared if both pointers point to objects of same type

e.g.: int arr[]={10, 20, 36, 72, 45, 36};

int*j, *k;

j=&arr[4];

k=(arr+4);

if(j==k)

cout<<"Pointer point to same location.";

else

cout<<"Pointer point to different location";

Output

Pointers point to same location.

Q8. Describe Macro.

Ans. Macro: We can define Macro by using #define directive. Each time the macro is encountered, the arguments used in its definition are replaced by the actual arguments found in the program. This form of a macro is called a function-like macro.

Example:

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

```
#define ABS( a )           ( a ) < 0 ? - ( a ) : ( a )
int main( )
{
    cout << "abs of 1 and 1 ; " << ABS ( -1 ) << ABS ( 1 );
    return 0;
}
```

When this program is compiled, a in the macro definition will be substituted with the values 1 and 1.

The use of a function-like macro in place of real functions has one major benefit: it increases the execution speed of the code because there is no function call overhead. However, if the size of the function-like macro is very large, this increased speed may be paid for with an increase in the size of the program because of duplicate code.

Although parameterised macros are a valuable features, C++ has a way of creating inline code which uses the `inline` keyword.

Q9. What is bridge function?

Ans. Bridge function is other name of friend function.

The concept of data encapsulation sometimes takes information hiding too far. There are situations where a rigid and controlled access leads to inconvenience and hardships.

For instance, consider a case where a function is required to operate on object of two different classes. This function cannot be made a member of both the classes. What can be done is that a function can be defined outside both the classes and made to operate on both. That is, a function not belonging to either, but able to access both. Such a function is called as a friend function. In other words, a friend function is a nonmember function, which has access to a class's private members.





Special Care to Our Students!

Get

**'How to pass
IGNOU Exams' Book**

Worth Rs.199/-

FREE

**Register now on Gullybaba and
download book from download section.**



<https://bit.ly/HowtoPassIGNOUExam>

Operator Overloading

Operator overloading is nothing but defining a member function, for a particular operator, which will be invoked when the operator is used with objects of that class. This is also called operational polymorphism. The second form of polymorphism in C++ is functional polymorphism also called function overloading.

Function Overloading

Function overloading is a form of polymorphism. Function overloading facilitates defining one function having many forms. In other words it facilitates defining several functions with the same name, thus overloading the function names. Like in operator overloading, even here, the compiler uses context to determine which definition of an overloaded function is to be invoked.

Function overloading is used to define a set of functions that essentially, do the same thing, but use different argument lists. The argument list of the function is also called as the function's signature. You can overload the function only if their signatures are different.

The differences can be:

- In number of arguments,
- Data types of the arguments, and
- Order of arguments, if the number and data types of the arguments are same.

E.g.

```
int add(int, int);
int add(int, int, int);
float add(float, float);
float add(int, float, int);
float add(int,int,float);
```

The compiler cannot distinguish if the signature is same and only return type is different. Hence, it is a must, that their signature is different. The following functions therefore raise a compilation error.

E.g.

```
float add(int, float, int);
int add(int, float, int);
```

Consider the following example, which raises a given number to a given power and returns the result.

```
long long_raise_to(int num, int power)
{
    long lp = 1;
```

```
for( int i = 1; i <= power; i++ )
{
    lp = lp * num ;
}
```

```
return lp;
}
```

```
double double_raise_to(double num, int power)
{
    double dp = 1;
```

```
for( int i = 1; i <= power ; i++ )
{
    dp = dp * num ;
}
```

```
return dp;
}
```

```
void main(void)
{
    cout<<"2 raise to 8 is "<<long_raise_to(2,8);
    cout<<"2.5 raise to 4 is "<<double_raise_to(2.5,4);
}
```

Precautions with function overloading

Function overloading is a boon to designers, since different names for similar functions need not be thought of, which often is a cumbersome process, many times people run out of names. But, this facility should not be overused, it becomes an overhead in terms of readability and maintenance. Only those functions, which basically do the same task, on different sets of data, should be overloaded.

Operator Overloading

Operator overloading is one of the most fascinating features of C++. It can transform complex, obscure program listings into intuitively obvious ones. By overloading operators we can give additional meaning to operators like +, *, -, <=, >=, etc. which by default are supposed to work only on standard data types like ints, floats, etc. For example, if str1 and str2 are two character arrays holding strings "Bombay" and "Nagpur" in them then to store "BombayNagpur" in a third string str3, in C the following operations need to be performed:

```
char str1[20] = "Nagpur";
char str2[] = "Bombay";
char str3[20];
strcpy (str3, str1);
strcat (str3, str2);
```

No doubt this does the desired task but don't you think that the following form would have made more sense:

```
str3 = str1 + str2 ;
```

Such a form obviously would not work with C, since we are attempting to apply the +operator on non-standard data types (strings) for which addition operation is not defined. That's the place where C++ scores over C, because it permits the + operator to be overloaded such that it knows how to add two strings.

Now take another example where operator overloading would make the programming logic more convenient to handle. Suppose we want to perform complex number arithmetic.

```
# include <iostream.h>
struct complex
{
double real, imag;
};
complex complex_set (double r, double i);
void complex_print (complex c);
complex operator + (complex c1, complex c2);
complex operator - (complex c1, complex c2);

void main ( )
{
complex a, b, c, d;
a = complex_set (1.0, 1.0);
b = complex_set (2.0, 2.0);
c = a + b;
d = b + c - a;
cout << endl << "c = ";
complex_print (c);
cout << endl << "d";
complex_print (d);
}
complex complex_set (double r, double i)
{
complex temp;
temp.real = r;
temp.imag = i;
return temp;
}
void complex_print (complex t)
{
cout << "(" << t.real << ',' << t.image << ")";
}
complex operator + (complex c1, complex c2)
{
```

```
complex temp;
temp.real = c1.real + c2.real;
temp.image = c1.image + c2.image;
return temp;
}
complex operator - (complex c1, complex c2)
{
complex temp;
temp.real = c1.real - c2.real;
temp.imag = c1.image - c2.imag;
return temp;
}
```

Operator overloading really speaking teaches a normal C++ operator to act on a user-defined operand. In our case the operator + and – are taught to operate on a user-defined data type “complex”. This is achieved by declaring a function that defines the actions to be performed on the arguments passed to it. This function is declared using the keyword operator to be overloaded.

Q1. How can you overload the input operator in C++? Describe with the help of example.

Or

How can you overload the output operator in C++? Describe with the help of an example.

Ans. We can overload input/output operator using references of istream and ostream objects along with friend function to permit normal representation even for user defined objects.

```
#include<iostream.h>
class de
{
public:
int a, b, c;
friend istream & operator >>(istream & os, de &obj);
friend ostream & operator <<(ostream & os, de &obj);
};
//overloading output operator
ostream& operator <<(ostream & os, de &obj)
{
```

```
os<<obj.a<<obj.b<<obj.c;
return os;
}
//overloading input operator
istream& operator >>(istream & os, de & obj)
{
os>>obj.a>>obj.b>>obj.c;
return os;
}
void main()
{
de w, e;
cin>>w;
cout<<w;
cin.get();
}
```

○○○

6

Inheritance

Overview

Object-oriented programming as seen in the preceding sessions emphasises the data, rather than emphasising algorithms. The previous sessions covered OOP features like extensibility, data encapsulation, information hiding, functional polymorphism and operational polymorphism. OOP, however, has more jargon associated to it, like reusability, inheritance. This session covers reusability and inheritance.

Reusability

Reusability means reusing code written earlier, may be from some previous project or from the library. Reusing old code not only saves development time, but also saves the testing and debugging time. It is better to use existing code, which has been time-tested rather than reinvent it. Moreover, writing new code may introduce bugs in the program. Code, written and tested earlier, may relieve a programmer of the nitty-gritty. Details about the hardware, user-interface, files and so on. It leaves the programmer more time to concentrate on the overall logistics of the program.

Inheritance

The new dimension of OOP uses a method called inheritance to modify a class to suit one's needs. Inheritance means deriving new classes from the old ones. The old class is called the *base class* or *parent class* or *super class* and the class which is derived from this base class is called as *derived class* or *child class* or *sub class*. Deriving a new class from an existing one, allows

redefining a member function of the base class and also adding new members to the derived class and this is possible without having the source also. In other words, a derived class not only inherits all properties of the base class but also has some refinements of its own. The base class remains unchanged in the process. In other words, the derived class "is a" type of base class, but with more details added.

Derived class

A singly inherited derived class is defined by writing:

- The keyword *class*.
- The name of the derived class.
- A single colon (:).
- The type of derivation (private, protected, or public).
- The name of the base, or parent class.
- The remainder of the class definition.

E.g.

```
class A
{
public :
    int public_A;
    void public_function_A();
```

```
private :
    int pri_A;
    void private_function_A();
```

```
protected :
    int protected_A;
    void protected_function_A();
};
```

```
class B : private A
{
public :
    int public_B;
    void public_function_B();
```

```
private :  
int pri_B;  
void private_function_B();  
};  
class C : public A  
{  
public :  
int public_C;  
void public_function_C();
```

```
private :  
int pri_C;  
void private_function_C();  
};  
class D : protected A  
{  
public :  
int public_D;  
void public_function_D();
```

```
private :  
int pri_D;  
void private_function_D();  
};
```

A derived class always contains all of the members from its base class. You cannot “subtract” anything from a base class. However, accessing the inherited variables is a different matter. Just because you happen to derive a class does not mean that you are automatically granted complete and unlimited access privileges to the members of the base class. To understand this you must look at the different types of derivation and the effect of each one.

Private Derivation

If no specific derivation is listed, then a private derivation is assumed. If a new class is derived privately from its parent class, then:

- The private members inherited from its base class are inaccessible to new member functions in the derived class. This

means that the creator of the base class has absolute control over the accessibility of these members, and there is no way that you can override this.

- The public members inherited from the base class have private access privilege. In other words, they are treated as though they were declared as new private members of the derived class, so that new member functions can access them. However, if another private derivation occurs from this derived class, then these members are inaccessible to new member functions.

E.g.

```
class base
{
private :
int number;
};

class derived : private base
{
public :
void f()
{
    ++number;           // Private base member not accessible
}
};
```

The compiler *error* message is

'base :: number' is not accessible in the function derived :: f();

E.g.

```
class base
{
public :
int number;
};

class derived : private base
{
public :
```

```
void f()
{
    ++number; // Access to number O.K.
}

class derived2 : private derived
{
public :
void g()
{
    ++number; // Access to number is
               // Prohibited.
}
};

The compiler error message is
'base :: number ' is not accessible in the function derived2 :: g();
```

Since public members of a base class are inherited as private in the derived class, the function derived :: f() has no problem accessing it. However, when another class is derived from the class derived, this new class inherits number but cannot access it. Of course, if derived1::g() were to call upon derived::f(), there is no problem since derived::f() is public and inherited into derived2 as private.

I.e. In derived2 we can write,

```
void g()
{
    f();
}
```

Public Derivation

Public derivations are much more common than private derivations. In this situation:

- The private members inherited from the base class are inaccessible to new member functions in the derived class.
- The public members inherited from the base class may be accessed by new member functions in the derived class and by instances of the derived class.

Eg.

```
class base
{
private :
int number;
};
```

```
class derived : public base
{
public :
void f()
{
    ++number;           //Private base member not accessible
}
};
```

The compiler *error* message is

'base :: number ' is not accessible in the function derived::f();

Here, only if the number is public then you can access it.

Note : However example 2 and 3 in the above section works here if you derive them as "public".

Protected Derivation

In addition to doing private and public derivations, you may also do a protected derivation. In this situation:

- The private members inherited from the base class are inaccessible to new member functions in the derived class.(this is exactly same as if a private or public derivation has occurred.)
- The protected members inherited from the base class have protected access privilege.
- The public members inherited from the base class have protected access.

Thus, the only difference between a public and a protected derivation is how the public members of the parent class are inherited. It is unlikely that you will ever have occasion to do this type of derivation.

Summary of Access Privileges

- If the designer of the base class wants no one, not even a derived class to access a member, then that member should be made private.
- If the designer wants any derived class function to have access to it, then that member must be protected.
- If the designer wants to let everyone, including the instances, have access to that member, then that member should be made public.

Summary of Derivations

- Regardless of the type of derivation, private members are inherited by the derived class, but cannot be accessed by the new member function of the derived class, and certainly not by the instances of the derived class.
- In a private derivation, the derived class inherits public and protected members as private. A new member function can access these members, but instances of the derived class may not. Also any member of subsequently derived classes may not gain access to these members because of the first rule.
- In public derivation, the derived class inherits public members as public, and protected as protected. A new member function of the derived class may access the public and protected members of the base class, but instances of the derived class may access only the public members.
- In a protected derivation, the derived class inherits public and protected members as protected. A new member function of the derived class may access the public and protected members of the base class, both instances of the derived class may access only the public members.

Multilevel Inheritance

In multilevel inheritance there is a parent class, from whom we derive another class. Now from this derived class we can derive another class and so on.

E.g.

```
class Aclass
{
    :
    :
}
```

```
class Bclass : public Aclass
{
    :
    :
}
```

```
class Cclass : public Bclass
{
    class equation      :
    :
}
```

Multiple Inheritance

Multiple inheritance, as the name suggests, is deriving a class from more than one class. The derived class inherits all the properties of all its base classes. Consider the following example:

Class A Class B
 Class C

E.g.

```
class Aclass
{
    :
    :
};
```

```
class Bclass
{
    :
    :
};
```

```
class Cclass : public Aclass, public Bclass
{
    private :
    :
}
```

```
public :
```

```
Cclass(...): Aclass (...), Bclass(...)
```

```
{  
};  
};
```

The class Cclass in the above example is derived from two classes - Aclass and Bclass - therefore the first line of its class definition contains the name of two classes, both publicly inherited. Like with normal inheritance, constructors have to be defined for initialising the data members of all classes. The constructor in Cclass calls constructors for base classes. The constructor calls are separated by commas.

Problems with Multiple Inheritance

The following example presents a problem with multiple inheritance.

```
class Aclass  
{  
public :  
void put()  
{  
:  
}  
};
```

```
class Bclass  
{  
public :  
void put()  
{  
:  
}  
};
```

```
class Cclass : public Aclass, public Bclass  
{
```

```
public :  
:  
:  
};  
  
void main()  
{  
A objA;  
B objB;  
C objC;  
  
objA.put();           //From class A  
objB.put();           //From class B  
objC.put();           //Ambiguous-results in Error  
}
```

The above example has a class C derived from two classes A and B. Both these classes have a function with the same name - put(), which assume, the derived class does not have. The class C inherits the put() function from both the classes. When a call to this function is made using the object of the derived class, the compiler does not know which class it is referring to. In this case, the scope resolution operator has to be used to specify the correct object. Its usage is very simple. The statement giving an error from the above example has to be replaced with the following:

```
objC.A::put();           //for class A  
objC.B::put();           //for class B
```

Multiple Inheritance with a common base (Hybrid Inheritance)

Inheritance is an important and powerful feature of OOP. Only the imagination of the person concerned is the limit. There are many combinations in which inheritance can be put to use. For instance, inheriting a class from two different classes, which in turn have been derived from the same base class.

```
E.g.  
class base  
{
```

```
:  
:  
};  
  
class Aclass : public base  
{  
:  
:  
};  
  
class Bclass : public base  
{  
:  
:  
};  
  
class derived : public Aclass, public Bclass  
{  
:  
:  
};
```

Aclass and Bclass are two classes derived from the same base class. The class derived has a common ancestor - class base. This is multiple inheritance with a common base class. However, this type of inheritance is not all that simple. One potential problem here is that both, Aclass and Bclass, are derived from base and therefore both of them, contain a copy of the data members of base class. The class derived is derived from these two classes. That means it contains two copies of base class members - one from Aclass and the other from Bclass. This gives rise to ambiguity between the base data members. Another problem is that declaring an object of class derived will invoke the base class constructor twice. The solution to this problem is provided by virtual base classes.

Virtual Base Classes

This ambiguity can be resolved if the class derived contains only one copy of the class base. This can be done by making the base class a virtual class.

This keyword makes the two classes share a single copy of their base class. It can be done as follows :

```
class base
{
    :
    :
};

class Aclass : virtual public base
{
    :
    :
};

class Bclass : virtual public base
{
    :
    :
};

class derived : public Aclass, public Bclass
{
    :
    :
};
```

This will resolve the ambiguity involved.

Abstract Classes

Abstract classes are the classes, which are written just to act as base class. Consider the following classes.

```
class base
{
    :
    :
};
```

```
class Aclass : public base
{
    :
    :
};
```

```
class Bclass : public base
{
    :
    :
};
```

```
class Cclass : public base
{
    :
    :
};
```

```
void main()
{
    Aclass objA;
    Bclass objB;
    Cclass objC;

    :
    :
}
```

There are three classes - Aclass, Bclass, Cclass - each of which is derived from the class base. The main () function declares three objects of each of these three classes. However, it does not declare any object of the class base. This class is a general class whose sole purpose is to serve as a base class for the other three. Classes used only for the purpose of deriving other classes from them are called as *abstract classes*. They simply serve as base class, and no objects for such classes are created.

Virtual Functions

The keyword *virtual* was earlier used to resolve ambiguity for a class derived from two classes, both having a common ancestor. These classes

are called virtual base classes. This time it helps in implementing the idea of polymorphism with class inheritance. The function of the base class can be declared with the keyword *virtual*. The program with this change and its output is given below.

```
class Shape
{
public :
    virtual void print()
    {
        cout << " I am a Shape " << endl;
    }
};

class Triangle : public Shape
{
public :
    void print()
    {
        cout << " I am a Triangle " << endl;
    }
};

class Circle : public Shape
{
public :
    void print()
    {
        cout << " I am a Circle " << endl;
    }
};

void main()
{
    Shape S;
```

```
Triangle T;
```

```
Circle C;
```

```
S.print();
```

```
T.print();
```

```
C.print();
```

```
Shape *ptr;
```

```
ptr = &S;
```

```
ptr -> print();
```

```
ptr = &T;
```

```
ptr -> print();
```

```
ptr = &C;
```

```
ptr -> print();
```

```
}
```

The **output** of the program is given below:

```
I am a Shape
```

```
I am a Triangle
```

```
I am a Circle
```

```
I am a Shape
```

```
I am a Triangle
```

```
I am a Circle
```

Pure Virtual Functions

An abstract class is one, which is used just for deriving some other classes. No object of this class is declared and used in the program. Similarly, there are pure virtual functions which themselves won't be used. Consider the above example with some changes.

```
class Shape
```

```
{
```

```
public :
```

```
virtual void print() = 0; // Pure virtual function
```

```
};
```

```
class Triangle : public Shape
{
public :
void print()
{
cout << " I am a Triangle " << endl;
}
};
```

```
class Circle : public Shape
{
public :
void print()
{
cout << " I am a Circle " << endl;
}
};
```

```
void main()
{
Shape S;
Triangle T;
Circle C;
```

```
Shape *ptr;
```

```
ptr = &T;
ptr -> print();
```

```
ptr = &C;
ptr -> print();
}
```

The **output** of the program is given below:

I am a Triangle
I am a Circle

Q1. What are the access control violation in the following program fragment :

```
Class X {  
    private :  
        int i;  
    void pvt_X();  
    protected :  
        int j;  
    void prot_X();  
    public :  
        int k;  
    void pub_X(int, int);  
};  
  
Class Y :: public X  
{  
    Char c;  
};  
  
void main() {  
    X x;  
    Y y;  
    x.i=10;  
    x.k=15;  
    y.prot_X();  
    y.pub_X(5, 10);  
};  
  
Ans. Class X {  
    private :  
        int i;  
    void pvt_X();  
    protected :  
        int j;
```

```

void prot_X();
public :
int k;
void pub_X(int, int);
};

Class Y :: public X      → (Mistake). Here “Class Y :: public X statement should be Class”.
Y : public X.
{
Char c;
};

void main() {
X x;
Y y;
x.i=10;           → (Wrong) . Here “i' is a private variable hence cannot be accessed outside the class definition”.
x.k=15;          → (Correct). Here “k' is a public variable and can be accessed with an object of X”.
y.prot_X();       → (Wrong). Here “protected members are only accessible within the derived class definition”.
y.pub_X(5, 10);
};

```

→ (Correct). Here “*pub_X() is a public member of class X*”

Q2. What are the access control violations in the following program segment:

```

class test_protected {
private:
int pri;
protected:
int pro;
public:

```

```
int pub;
test_protected () {}
void print (); ;
class test_pro : private test_protected
private:
int sub_pri;
public:
int read ();
void print () {
cout << pri << pro << pub << sub_pri;
}
void main () {
test_protected t;
t.print ();
}
```

Ans. #include<iostream.h>
#include<conio.h>

```
class test_protected
{
private :
int pri; //private data member of the class
protected :
int pro;
public:
int pub;
test_protected();
void print();
};

class test_pro:private test_protected
{
private : int sub_pri;
```

```
public:  
int read();  
void print()  
{  
cout<<pri<<pro<<pub<<sub_pri;      // pri cannot access private member  
//declare in class test_protected  
}  
};  
void main()  
{  
test_protected t;  
t.print();  
}
```

Q3. Design and implement a class hierarchy for the following requirements:

- The class should represent complex numbers.
- It should have overloaded operator for addition and multiplication of complex numbers
- If an attempt is made to divide one complex number with another it generates an error message.
- The class should have suitable constructors, destructors, copy constructor and assignment operator.

Write appropriate main (). Make and state suitable assumptions, if any.

Ans. #include<iostream.h>

```
#include<conio.h>  
class Complex {  
double real;  
double imag;  
public:  
class divide(){};  
//constructors  
Complex();           //default
```

```
Complex(float);           //real given
Complex(float, float);   //both given
Complex(Complex &s);     //copy constructor
                        //other methods

void show()
{
cout<<"real= "<<real<<
"imaginary= "<<imag<<endl;
}

//operator methods
Complex operator+(Complex);
Complex operator-(Complex);
Complex operator*(Complex);
Complex operator/(Complex);
void operator=(Complex);
};

//default constructor
Complex::Complex()
{
real= imag=0.0;
}

//constructor--real given but not imag
Complex::Complex(float)
{
real=r;imag=0.0;
}

Complex::Complex(Complex &s)
{
real=s.real;imag=s.imag;
}

//constructor--real and imag given
Complex::Complex(float r, float)
{
real=r;imag=i;
```

```
}

//complex+as binary operator
Complex Complex::operator+(Complex c)
{
    return Complex(real+c.real, imag+c.imag);
}

//Complex-as binary operator
Complex Complex::operator-(Complex c)
{
    return Complex(real-c.real,imag-c.imag);
}

Complex Complex::operator*(Complex c)
{
    return Complex(real*c.real-imag*c.imag,imag*c.real+real*c.imag);
}

Complex Complex::opeator/(Complex c)
{
    throw divide();
}

void Complex::operator=(Complex c)
{
    Complex temp;
    this.real=c.real;
    this.imag=c.imag;
}

void main()
{
    clrscr();
    Complex c1(2.0, 5.2), c2(2.3, 5.9), c3;
    Complex c4(c3);
    try{
        c4=c1+c2;
        c4.show();
        c4=c1*c2;
        c4.show();
    }
}
```

```
c4=c1/c2;  
}  
catch(Complex::divide)  
{  
cout<<"can't divide".  
}  
getch();  
}
```

Q4. What are the access control violation in the following program segment:

```
class All_public {  
public:  
int all_pub;  
All_public (){};  
void print ()  
}  
class Inherited : Private All_public {  
private:  
int some_private_data;  
public:  
int read();  
void print();  
}
```

Ans. The following access violations are there:

- All_public::all_pub is not accessible. Because in this example our subclass inherited is privately inherited not publicly.
- All_public is not member of inherited subclass.
- Inherited::Second.some_private_data=200; is not accessible because it is defined private in subclass inherited.

Q5. What are the access control violations in the following program segment:

```
Class A {
```

```
int x;
```

```
protected:  
int y  
public:  
A( ) {x = 5, y = 10};  
print(); //prints A.  
};  
  
Class B : public A {  
int z;  
public:  
B(){  
z = 5;  
x = 1;  
y = 2;  
};  
  
main(){  
A a;  
B b;  
a.z = 5;  
b.print();  
b.x = 10;  
};  
void main()  
{  
All_public first;  
inherited second ;  
second.all_pub = 10;  
second.all_public.print();  
first.all_pub = 50;  
second. some_private _ data = 200;  
}.  
}
```

Ans.

CLASS B	
(1)	Private member int z
(2)	Protected member intY
(3)	Public member A(),B(),Print()

- a.z=5 not possible a is not accessible to object of class A.
- b.x=10 wrong because X is private member of class A. So not inheritable.

Q6. What are the access control violations in the following program segment:

```
class X{  
    int one ;  
protected :  
    int two ;  
public :  
    int three;  
    void x1(){  
        one=1;  
        two=2;  
        three=3;  
    };  
}  
  
class Y : public x{  
    int four;  
public:  
    void y2 ()  
    {  
        one=2;  
        two=3;  
        three=4;  
        four=5;
```

```

}
};

void main (){
    X a;
    Y b;
    a.x1();
    a.one=5;
    b.one=10;
    b.y2 ();
}

```

Ans.

Class y	
(1)	Private member
	int four
(2)	Protected member
	int two
(3)	Public member
	int three
	void X1 ()
	void Y2 ()

- a.one=5 is inaccessible because one is private member of class X.
- b.one is wrong because one is not a member of class Y.

Q7. What is meant by the term information violate information hiding? Describe with the help of an example. Does friend function violate information hiding? Justify your answer.

Ans. The term information hiding differs from the term information violate. The word information violate means that you are using the Public, Private, Protected access specifier. It means that you are restricting its limit or defining its area or specifying its boundary. If any function is breaking its limit or boundary that means it violates its rules. The term information hiding refers that some part of data is hiding from user. It restrict user to access unauthorised data.

Example : Oracle database user is authorised by Database Administrator to see some of the tables and not all the company details.

Yes, friend function violate information hiding because with help of friend function you are able to access the private part or data of the class .

Q8. What is single and multiple inheritance? Explain with the help of an example. What are the problems relating to multiple inheritance?

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

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

The derived class inherits some or all of the traits from the base class. A class can also inherit properties from more than one class or from more than one level. A derived class with only one base class is called single inheritance and one with several base classes is called multiple inheritance. On the other hand, the traits of one class may be inherited by more than one class. This process is known as hierarchical inheritance. The mechanism of deriving a class from another 'derived class' is known as multilevel inheritance. Following figure shows various forms of inheritance that could be used for writing extensible programs. The direction of arrow indicates the direction of inheritance.

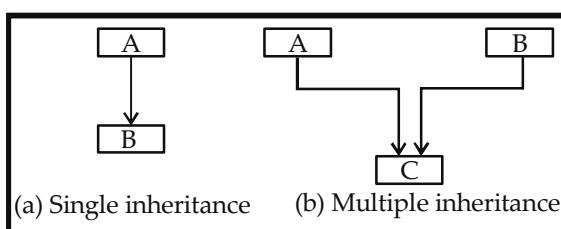


Fig. 6.1

Q9. Design a class hierarchy in C++ for employees of a company. Employees can be of two types, full time and part-time, all employees are employed for a particular department. One of the employees in the department is designated as manager. In your design you may

include the data members, constructor, destructor and one polymorphic function for printing the details.

Ans. #include<iostream.h>

```
#include<conio.h>
#include<string.h>
class employee
{
protected:
int no;
char name[15];
char deptname[25];
char desig[25];
public:
void getdata()
{
cout<<"Enter No:";
cin>>no;
cout<<"\nEnter Name:";
cin>>name;
cout<<"\nEnter Department name:";
cin>>deptname;
}
void putdata()
{
cout<<"No="<<no;
cout<<"\nName="<<name<< "\n";
cout<<"Dept Name="<<deptname<<"\n";
}
};

class Fulltimeemp:public employee
{
protected:
float salary;
public:
void getdata()
```

```
{  
employee::getdata();  
cout<<"\nEnter salary:";  
cin>>salary;  
}  
void putdata(void)  
{  
employee::putdata();  
cout<<"Salary ="<<salary<<"\n";  
}  
};  
class parttime:public employee  
{  
protected:  
float salary;  
public:  
void getdata()  
{  
employee::getdata();  
cout<<"\nEnter fees:";  
cin>>salary;  
}  
void putdata(void)  
{  
employee::putdata();  
cout<<"Salary ="<<salary<<"\n";  
}  
};  
int main()  
{  
int ch,i=0,j=0,n,k;  
do  
{  
clrscr();  
cout<<"1.FullTimeEmployee\n";  
}
```

```
cout<<"2.PartTimeEmployee\n";
cout<<"3.Exit\n";
cout<<"Enter choice(1/2/3):";
cin>>ch;
switch(ch)
{
    case 1: Fulltimeemp ob1[10];
    ob1[i++].getdata();
    break;
    case 2:parttime ob2[10];
    ob2[j++].getdata();
    break;
    case 3:if(i!=0 | | j!=0)
    {
        n= i;
        int k=0;
        for(; k<n; k++)
        ob1[k].putdata();
        n=j;
        k=0;
        for(; k<n; k++)
        ob2[k].putdata();
        getch();
    }
}
}while(ch!=3);
return 0;
}
```

Q10. Design and develop a polymorphic class to solve a general quadratic equation $ax^2 + bx + c = 0$.

Ans. #include<iostream.h>

```
#include<conio.h>
#include<math.h>
class equation
{
```

protected:

```
float a, b, c;
```

public:

```
void getdata()
```

```
{
```

```
cout<<"Enter value if a, b, c";
```

```
cin>>a>>b>>c;
```

```
}
```

```
int returna(){return a;}
```

```
int returnb(){return b;}
```

```
int returnc(){return c;}
```

```
equation(){a=0; b=0; c=0;
```

```
}
```

```
equation(int x,int y,int z)
```

```
{ a=x; b=y; c=z;}
```

```
virtual void showdata(float a, float b, float c){}
```

```
};
```

```
class imaginary:public equation
```

```
{
```

public:

```
void showdata(float a,float b,float c)
```

```
{
```

```
if(b*b-4*a*c<0)
```

```
cout<<"Roots are imaginary";
```

```
}
```

```
};
```

```
class real:public equation
```

```
{
```

```
float r1, r2;
```

public:

```
void showdata(float a, float b, float c)
```

```
{  
if(b*b-4*a*c>0)  
{  
r1=(-b+sqrt(b*b-4*a*c))/2*a;  
r2=(-b-sqrt(b*b-4*a*c))/2*a;  
  
cout<<"Roots are :"<<r1<<":"<<r2;  
}  
}  
};
```

```
class equal:public equation  
{  
float r;  
public:  
void showdata(float a,float b,float c)  
{  
if(b*b-4*a*c==0)  
{  
r=-b/(2*a);  
cout<<"Roots are :"<<r<<" each ";  
}  
}  
};
```

```
void main()
```

```
{  
clrscr();  
  
equation*r[3];  
imaginary m;  
real re;
```

```
equal eq;  
equation e;  
e.getdata();  
r[0]=&m;  
r[1]=&re;  
r[2]=&eq;  
for(int i=0;i<3;i++)  
r[i]->showdata(e.returna(),e.returnb(),e.returnc());  
//check all functions to decide roots  
getch();  
}
```

Q11. What is late binding? How is it same/different from dynamic binding?

Or

What is late binding? Is it different from dynamic binding? Justify your answer with the help of an example.

Or

What is Dynamic binding? How is it useful in implementing polymorphism in object orient systems? Explain with the help of an example.

Ans. At the design level, when a software engineer is deciding what type of action is appropriate for a given object, he or she should not be concerned about how the object interprets the action (message) and implements the methods but only what the effect of the action is on the object. Object-oriented languages like C++ and Smalltalk allow a programmer to send identical messages to dissimilar but related objects and achieve identical actions while letting the software system decide how to achieve the required action for the given object. A key issue associated with polymorphism is the timing of the software system's implementations decision. If the system decides how to implement an action at compile time, this is called early binding. If the decision is made dynamically at run-time, it is called late binding. Generally late binding offers the advantages of flexibility and a high level of problem abstraction.

Eg.

```
#include<iostream.h>  
class x //base class  
{  
public:
```

```
virtual void show()           //virtual function
{
cout<<"x";
}

};

class y: public x           //derived class
{
public:
void show()
{
cout<<"y";
}
};

class z:public y           //super class
{
public:
void show()
{
cout<<"z";
}
};

// In this we call a function of derived class using base class pointer. One
// important //point is that you must declare base function by using virtual
// keyword.

void main()
{
x *xb;
x xobj;
y yobj;
z zobj;

xb=&xobj;
xb->show();               //only x class function is called
```

```
xb=&yobj;  
xb->show(); // only y class function is called  
xb=&zobj;  
xb->show(); // only z class function is called  
}
```

Q12. Design a polymorphic class hierarchy for the Windows based software. Assume that a window consist of Menu, Tools, Buttons and Iteraction area. Your class should include necessary data members, constructor/destructor and at least one polymorphic function. Make and state suitable assumptions, if any.

Ans.

```
#include<iostream.h>  
#include<conio.h>  
#include<stdio.h>  
class windows  
{  
protected:  
int length;  
int width;  
public:  
void getwindowdimensions()  
{  
cout<<"Enter window length in pixels";  
cin>>length;  
cout<<"Enter window breadth in pixels";  
cin>>width;  
}  
virtual void getdata()=0;  
};  
  
class Menu:public windows  
{  
char name[20][20];  
int no_menu;  
public:  
Menu()
```

```
{  
no_menu=0;  
}  
~Menu(){}
void getdata()
{
cout<<"Enter length & width of menu in pixels";
cin>>length>>width;
cout<<"Enter no: of menus";
cin>>no_menu;
cout<<"Enter menu names\n";
for(int i=0;i<no_menu;i++)
{
cout<<"\nMenu:<<i+1<<":";
gets(name[i]);
}
}
};

class Tools:public windows
{
char name[20][20];
int no_tools;
public:
Tools()
{
no_tools=0;
}
~Tools(){}
void getdata()
{
cout<<"Enter no: of tools";
cin>>no_tools;
cout<<"Enter tool names\n";
```

```
for(int i=0; i<no_tools; i++)
{
cout<<"\nTool:"<<i+1<<"";
gets(name[i]);
}
}
};

class Buttons:public windows
{
char text[20][20];
int no_buttons;
public:
Buttons()
{
no_buttons=0;
}
~Buttons(){}
void getdata()
{
cout<<"Enter length & width of button in pixels";
cin>>length>>width;
cout<<"Enter no: of buttons";
cin>>no_buttons;
cout<<"Enter button names\n";
for(int i=0;i<no_buttons;i++)
{
cout<<"\nButton:"<<i+1<<"";
gets(text[i]);
}
}
};

class Interaction_Area:public windows
{
```

```
char message[50];
public:
Interaction_Area()
{
length=639;
width=479;
}
~Interaction_Area(){}
void getdata()
{
cout<<"Enter length and width of interaction area in pixels";
cin>>length>>width;
cout<<"Enter the message to appear by default";
gets(message);
}
};
void main()
{
clrscr();
windows *w[4];

Menu m1;
w[0]=&m1;
w[0]->getwindowdimensions();

Tools t1;
w[1]=&t1;

Buttons b1;
w[2]=&b1;

Interaction_Area a1;
w[3]=&a1;

cout<<"Enter data for windows and other elements\n";
```

```
for(int i=1;i<4;i++)
{
w[i]->getdata();
cout<<"\n\n";
}

getch();
```

Q13. Design a polymorphic class hierarchy for four-side objects. The four-side objects which may be included in this class may be square, rectangle or rombus. The class should have explicit constructors. The class should have polymorphic functions for area calculations and printing. Write the appropriate main() function that have 10 such objects and find the area of these objects/print those objects using polymorphic function.

Ans.

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
class four_sided_objects
{
protected:
int length;
int width;
int height;
int area;

public:
virtual void getdata()=0;
virtual void calcarea()=0;
virtual void printarea()=0;
};

class square:public four_sided_objects
{
public:
square()
```

```
{  
length=10;  
}  
~square(){}
void getdata()
{
cout<<"Enter side of square in pixel";
cin>>length;
}
void calcarea()
{
area=length*length;
}
void printarea()
{
cout<<"Area of square is"<<area;
}
};

class rectangle:public four_sided_objects
{
public:
rectangle()
{
length=10;
width=5;
}
~rectangle(){}
void getdata()
{
cout<<"Enter length and breadth of rectangle in pixel";
cin>>length>>width;
}
void calcarea()
```

```
{  
area=length*width;  
}  
void printarea()  
{  
cout<<"Area of rectangle is"<<area;  
}  
};  
  
class rhombus:public four_sided_objects  
{  
public:  
rhombus()  
{  
length=10;  
height=5;  
}  
~rhombus(){}
void getdata()  
{  
cout<<"Enter longer side and distance between longer sides in pixel";  
cin>>length>>height;
}  
void calcarea()  
{  
area=length*height;
}  
void printarea()  
{  
cout<<"Area of rhombus is"<<area;  
}  
};  
  
void main()
```

```
{  
clrscr();  
four_sided_objects *fs[3];  
square s1;  
rectangle r1;  
rhombus h1;  
fs[0]=&s1;  
fs[1]=&r1;  
fs[2]=&h1;  
  
cout<<"Enter the data:\n";  
  
for(int i=0; i<10; i++)  
for(int j=0; j<3; j++)  
if(j==0)  
{  
cout<<"\n\nEnter data for square:<<i+1<<"\n";  
fs[j]->getdata();  
cout<<"\n";  
fs[j]->calcarea();  
fs[j]->printarea();  
}  
else  
if(j==1)  
{  
cout<<"\n\nEnter data for rectangle:<<i+1<<"\n";  
fs[j]->getdata();  
cout<<"\n";  
fs[j]->calcarea();  
fs[j]->printarea();  
}  
else  
{
```

```
cout<<"\n\nEnter data for rhombus:<<i+1<<"\n";
fs[j]->getdata();
cout<<"\n";
fs[j]->calcarea();
fs[j]->printarea();
}
getch();
}
```

Q14. Design a polymorphic class hierarchy for the students of a university.

Each student is attached to a department. However, there are two types of students: day scholars and hostellers. Day scholars get a 30% concession in fee as they are not availing hostel fee. Design the suitable constructors/destructors and polymorphic function for fee calculation. Implement the classes. Write appropriate main() function which calculates the fee of the students of a batch of 100 belonging to two departments. Make and state suitable assumptions.

Ans.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class student
{
protected:
int no;
char name[15];
char deptname[25];
public:
void getdata()
{
cout<<"Enter No:";
cin>>no;
cout<<"\nEnter Name:";
cin>>name;
cout<< "\nEnter Department name:";
cin>>deptname;
}
void putdata()
```

```
{  
cout<<"\nNo ="<<no;  
cout<<"\nName ="<<name;  
cout<< "\nDepartment name :"<<deptname;  
  
}  
};  
class daysscholar:public student  
{  
protected:  
float fees;  
public:  
void getdata()  
{  
student::getdata();  
cout<<"\nEnter fees:";  
cin>>fees;  
}  
void putdata(void)  
{  
student::putdata();  
cout<<"\nFees ="<<(fees-(fees*.30))<<"\n";  
}  
};  
class hostlar:public student  
{  
protected:  
float fees;  
public:  
void getdata(void)  
{  
student::getdata();  
cout<<"\nEnter fees:";  
cin>>fees;  
}
```

```
void putdata(void)
{
student::putdata();
cout<<"\nFees ="<<fees<<"\n";
}
};

void main()
{
int ch,i=0,j=0,n,k;
do
{
clrscr();
cout<<"1.days scholar\n";
cout<<"2.Hostler\n";
cout<<"3.Exit\n";
cout<<"Enter choice(1/2/3):";
cin>>ch;
switch(ch)
{
case 1:daysscholar ob1[100];
ob1[i++].getdata();
break;
case 2:hostlar ob2[100];
ob2[j++].getdata();
break;
case 3:if(i!=0 | | j!=0)
{
n=i;
int k=0;
for(;k<n;k++)
ob1[k].putdata();
n=j;
k=0;
for(;k<n;k++)
}
```

```
ob2[k].putdata();
getch();
}
}
}
while(ch!=3);
}
```

○○○

Template and Streams

Function Templates

Function templates provide you with the capability to write a single function that is a skeleton, or template, for a family of similar functions.

Function overloading technique it relieves someone who is using your functions from having to know about different names for various functions that essentially do the same task. Unfortunately, overloaded functions are not the ultimate solution to the problem.

Consider the following example:

```
int max(int x, int y)
{
    return (x > y) ? x : y ;
}

float max(float x, float y)
{
    return (x > y) ? x : y ;
}

long max(long x, long y)
{
    return ( x > y) ? x : y ;
```

```
}
```

```
char max(char x, char y)
{
    return (x > y) ? x : y ;
}

void main()
{
    cout << max( 1, 2) << endl;
    cout << max( 4L, 3L) << endl;
    cout << max( 5.62, 3.48) << endl;
    cout << max('A', 'a') << endl;
}
```

The output is:

2
4
5.62
a

Even though function overloading is used, the problem with this example is that there is still too much repetitious coding. In other words, each function is essentially doing the same task. Now, instead of writing many functions, it would be nice if we were to write only one function and which can accommodate almost any type of input argument.

How a function template solves this problem?

A function template solves the problem by allowing you to write just one function that serves as a skeleton, or, template, for a family of functions whose tasks are all similar.

This function template does not specify the actual types of the arguments that the function accepts; instead, it uses a generic, or parameterised type, as a “place holder”, until you determine the specific types. The process of invocation of these functions with actual parameter types is called the template instantiation.

How to write a function template?

A function template should be written at the start of the program in the global area, or you may place it into a header file. All function templates start with a template declaration.

The syntax is:

- (1) The C++ keyword template.
- (2) A left angle bracket (<).
- (3) A comma separates a list of generic types, each one. A generic type consists of two parts:
 - (a) The keyword class (this usage of class has nothing to do with the keyword class used to create user-defined type.)
 - (b) A variable that represents some generic type, and will be used whenever this type needs to be written in the function definition. Typically the name T is used, but any valid C++ name will do.
- (4) A right angle bracket (>).

E.g.

```
template <class T>
T max(char x, char y)
{
    return (x > y) ? x : y ;
}
void main()
{
    cout << max(1, 2) << endl;
    cout << max(5.62, 3.48) << endl;
    cout << max('A', 'a') << endl;
    cout << max(4, 3) << endl;
}
```

The output is:

```
2
5.62
a
6
```

Each of first three max () function class causes a template function to be instantiated, first with an int, then with a double and then with a char. The last call using two integers does not cause another instantiation to occur because this call can use the instantiated function that was generated as a result of the first class that also uses two integers.

Class Templates

In addition to function templates, C++ also supports the concept of class templates. By definition, a class template is a class definition that describes a family of related classes. The philosophy is essentially the same as that of function templates, i.e. the ability to create a class (or structure) that contains one or more types that are generic, or parameterised.

A class template is defined in a manner similar to that of a function template. You start by writing a template declaration followed by the class definition:

E.g.

```
Template <class T>
class test
{
private : T item;

// data and functions
};
```

The generation of a class from class definition is called template instantiation. We can declare variables of this class as follows:

```
void main()
{
test<int> intobj;
test<double> doubleobj;
}
```

In the first instantiation the entire template types, i.e. variables of type T will be converted as integers. In the second case they are converted as double variables.

Of course, you may have more than one parameterised type between the angle brackets.

```
Template <class T1, class T2, class T3>
```

```
class classname;
```

Stream Class Hierarchy

The I/O system of C++ contains a set of classes that define the file handling methods. These include ifstream, ofstream and fstream. These classes are derived from fstreambase and from the corresponding iostream.h class as shown in Figure. These classes, designed to manage the disk files, are declared in fstream.h and therefore we must include this file in any

program that uses files. Table 7.1 shows the details of file operation classes. Note that these classes contain many more features. For more details, refer to the manual.

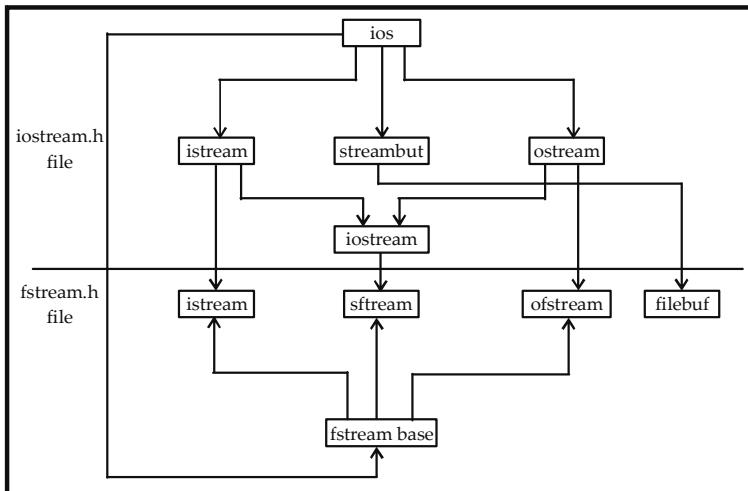


Fig. 7.1

Table 7.1

Class	Contents
filebuf	Its purpose is to set the file buffers to read and write. Contains Openprot constant in the open() of file stream. Also contains close() and open() as members.
fstreambase	Provides operations common to the file streams. Serves as a base for fstream , ifstream and ofstream classes. Contains open() and close() functions.
ifstream	Provides input operations. Contains open() with default input mode. Inherits the functions get() , getline() , read() , seekg() and tellg() functions from istream .
ofstream	Provides output operations. Contains open() with default output mode. Inherits put() , seekp() , tellp() , and write() functions from ostream .
fstream	Provides support for simultaneous input and output operations. Contains open() with default input mode. Inherits all the functions from istream and ostream classes through iostream .

Example: To input objects from an input file and outputting objects to an output file.

```
#include<fstream.h>
#include<conio.h>
#include<string.h>
class me
{
int rollno;
char name[25];
public:
void getdata(int rno, char*n)
{
rollno=rno;
strcpy(name, n);
}
void display()
{
cout<<"Roll No="<<rollno<<"\n";
cout<<"Name="<<name<<"\n";
}
};

int main()
{
clrscr();
ofstream ofile("me.dat");
me obj;
obj.getdata(1111,"Rithik");
ofile.write((char*)&obj, sizeof obj);
ofile.close();
ifstream ifile;
ifile.open("me.dat", ios::in); //open file in input mode
ifile.read((char*)&obj, sizeof obj);
obj.display();
getch();
return 0;
}
```

Q2. How can files be used for input/output? Discuss with the help of an example.

Ans. Input/Output from Text Files

Reading a text file is very easy using an ifstream (input file stream).

- Include the necessary headers.
- #include <fstream>
using namespace std;
- Declare an input file stream (ifstream) variable. For example,
ifstream myInput;
- Open the file stream. For example,
myInput.open("C:\\temp\\datafile.txt");
Path names in MS Windows use backslashes (\). Because the backslash is also the string escape character, it must be doubled. If the full path is not given, most systems will look in the directory that contains the object program.
- Check that the file was opened. For example, the open fails if the file doesn't exist, or if it can't be read because another program is writing to it. A failure can be detected with code like that below using the ! (logical not) operator:
- if (!myInput) {
• cerr << "Unable to open file datafile.txt";
• exit(1); // call system to stop
}- Read from the stream in the same way as cin.
• while (myInput >> x) {
• sum = sum + x;
}- Close the input stream. Closing is essential for output streams to make sure that all the information has been written to the disk, but is also good practice for input streams to release system resources and make the file available for other programs that might need to write it.
myInput.close();

Example:

The following program reads integers from a file and prints their sum.

```
#include <iostream>
```

```
#include <fstream>
```

```
using namespace std;
int main() {
    ifstream myInput;
    int x;
    int sum = 0;
    myInput.open("test.txt");
    if (!myInput) {
        cerr << "Unable to open test.txt.";
        exit(1);
    }
    while (myInput >> x) {
        sum += x;
    }
    cout << "total = " << sum << endl;
    myInput.close();
    return 0;
}
```

Q3. Design a template class that is used to create a circular linked list.

Ans. #include<iostream.h>

```
#include<conio.h>
#include<process.h>
template <class t>
class node
{
protected:
    node* count;
    node* p;
    node* q;
    node* first;
    node* last;
public:
    t data;                                //template type
    node* ptr;
    node()
    {
        count=NULL;
    }
```

```
}
```

public:

```
void create(t a);
```

```
void display();
```

```
void insert(t,t);
```

```
void delete1(t &);
```

```
};
```

```
template <class t>
```

```
void node<t> :: create(t a)
```

```
{
```

```
int i;
```

```
p=new node;
```

```
cout<<"\ninput node information :";
```

```
cin>>p->data;
```

```
p->ptr=NULL;
```

```
first=p;
```

```
for(i=1; i<a; i++)
```

```
{
```

```
q=new node;
```

```
cout<<"\ninput node information:";
```

```
cin>>q->data;
```

```
p->ptr=q;
```

```
q->ptr=NULL;
```

```
p=q;
```

```
}
```

```
last=p;
```

```
p->ptr=first;
```

```
}
```

```
template <class t>
```

```
void node<t> :: display()
```

```
{
```

```
count=first;
```

```
p=first;
```

```
do
```

```
{
```

```
cout<<p->data<<" ";
p=p->ptr;
}while(p!=count);
}

template <class t>
void node<t>::insert(t a, t b)
{
if(last->data==a)
{
q=new node;
p=first;
do
{
if(p->data==a)
{
q->data=b;
q->ptr=p->ptr;
p->ptr=q;
}
p=p->ptr;
}while(p!=count);
p->ptr=first;
}
else
q=new node;
p=first;
do
{
if(p->data==a)
{
q->data=b;
q->ptr=p->ptr;
p->ptr=q;
}
```

```
p=p->ptr;
}while(p!=count);
}

template <class t>
void node<t>::delete1(t &item)
{
p=first;
if(p->data==item)
{
first=p->ptr;
delete p;
}
else
{
q=first;
p=p->ptr;
while(p!=NULL)
{
if(p->data==item)
{
q->ptr=p->ptr;
delete p;
break;
}
q=p;
}
}
}

void main()
{
clrscr();
int ch, i, n, po;
node<char> k;
```

```
do {
    cin.clear();
    cout<<"\n\t MAIN MENU"      ;
    cout<<"\n  -----"   ;
    cout<<"\n  Single circular linked list";
    cout<<"\n      1. create";
    cout<<"\n      2. display";
    cout<<"\n      3. insert";
    cout<<"\n      4. delete";
    cout<<"\n      5. exit";
    cout<<"\n\n ENTER YOUR CHOICE... \t";
    cin>>ch;
    switch(ch)
    {
        case 1:
            cout<<"\nhow many no. of nodes :";
            cin>>n;
            k.create(n);
            k.display();
            break;
        case 2:
            k.display();
            getch();
            break;
        case 3:
            cout<<"\ninsert after";
            cin>>po;
            cout<<"\nenter data to be inserted";
            cin>>i;
            k.insert(po,i);
            break;
        case 4:
            cout<<"\nnode to be deleted:";
            cin>>po;
            k.delete1(po);
```

```
k.display();
break;
case 5:    exit(0);
}
}while(ch!=5);
}
```

Q4. Design a template class for a binary search tree having only the search and insert function. Make suitable assumptions, if any.

Ans. #include<iostream.h>

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
template <class t>
class tree
{
public:
t info;
t val;
tree *left;
tree *right;
tree* create(tree*);
tree* inserttree(tree*,tree*);
bool search( tree *node,t val);
void traverse(tree<t> *node);
};

void main()
{
int ch;
bool sh;
tree<int> *root;
tree<int> obj;

//clrscr();
```

```
do
{
    cout<<"\n M E N U ";
    cout<<"\n 1 Create a Tree";
    cout<<"\n 2 Search ";
    cout<<"\n 3 Traverse ";
    cout<<"\n 4 Exit";
    cout<<"\n Enter your Choice: ";
    cin>>ch;
    switch(ch)
    {
        case 1:
            root=obj.create(root);
            break;
        case 2:
            cout<<"Enter the search value";
            cin>>obj.val;
            sh=obj.search(root,obj.val);
            if(sh==true)
            {
                cout<<"value found ";
            }
            else
            {
                cout<<"value not found";
            }
            getch();
            break;
        case 3:
            obj.traverse(root);
            getch();
            break;
        case 4:
            cout<<"\nExit";
```

```
getch();
break;
default:
cout<<"\n Wrong choice";
getch();
}
}
while(ch!=4);
}

template <class t>
void tree<t>::traverse(tree<t> *node)
{
if(node)
{
traverse(node->left);
printf(" %d -> ",node->info);
traverse(node->right);
}
}

template <class t>
bool tree<t>::search(tree<t> *node, t val)
{
if(node)
{
if(node->info==val)
{
return true;
}
else if(val<node->info)
{
return search(node->left,val);
}
}
```

```
else
{
    return search(node->right,val);
}
}

return false;
}
```

```
template <class t>
tree<t>* tree<t>::create(tree<t> *root)
{
    t num;
    tree *p;
    root=NULL;
    cout<<"\n Enter tree info: ";
    cin>>num;
    do
    {
        p=new tree;
        p->info=num;
        p->left=NULL;
        p->right=NULL;
        root=inserttree(root,p);
        cout<<"\n Enter tree info(0 to stop):";
        cin>>num;
    }while(num!=0);
    return root;
}

template <class t>
tree<t>* tree<t>::inserttree(tree<t> *r,tree<t> *p)
{
    if(r==NULL)
        r=p;
    else
```

```
{  
if(p->info>=r->info)  
r->right=inserttree(r->right,p);  
else  
r->left=inserttree(r->left,p);  
}  
return r;  
}
```

Q5. Design a class for a circular queue using pointer.

Ans. #include<iostream.h>

```
#include<stdio.h>  
#include<ctype.h>  
#include<alloc.h>  
#define FALSE 0  
#define NULL 0  
struct listelement  
{  
int dataitem;  
struct listelement *link;  
};  
class queue :public listelement  
{  
public:  
listelement * lp,* tempp;  
  
void Menu (int *choice);  
listelement * AddQ (listelement * listpointer, int data);  
listelement * DeleteQ (listelement * listpointer);  
void PrintQueue (listelement * listpointer);  
void ClearQueue (listelement * listpointer);  
};  
void main ()  
{  
listelement *listpointer;
```

```
queue x;
int data, choice;
listpointer = NULL;
do {
    x.Menu (&choice);
    switch (choice) {
        case 1:
            cout<< ("Enter data item value to add");
            cin>>data;
            listpointer = x.AddQ (listpointer, data);
            break;
        case 2:
            if (listpointer == NULL)
                cout<<"Queue empty!\n";
            else
                listpointer = x.DeleteQ (listpointer);
            break;
        case 3:
            x.PrintQueue (listpointer);
            break;
        default:
            cout<<"Invalid menu choice - try again\n";
            break;
    }
} while (choice!= 4);
x.ClearQueue (listpointer);
}
/* main */
void queue::Menu (int *choice)
{
    char local;

    cout<<"\nEnter\t1 to add item,\n\t2 to remove item\n\
\t3 to print queue\n\t4 to quit\n";
}
```

```
do {
    local = getchar ();
    if ((isdigit (local) == FALSE) && (local != '\n'))
    {
        cout<< "\nyou must enter an integer.\n";
        cout<< "Enter 1 to add, 2 to remove, 3 to print, 4 to quit\n";
    }
} while (isdigit ((unsigned char) local) == FALSE);
*choice = (int) local - '0';
}

listelement * queue:: AddQ (listelement * listpointer, int data)
{
    lp = listpointer;

    if (listpointer != NULL)
    {
        while (listpointer -> link != NULL)
            listpointer = listpointer -> link;
        listpointer -> link = new listelement;
        listpointer = listpointer -> link;
        listpointer -> link = NULL;
        listpointer -> dataitem = data;
        return lp;
    }
    else {
        listpointer = new listelement;
        listpointer -> link = NULL;
        listpointer -> dataitem = data;
        return listpointer;
    }
}

listelement * queue ::DeleteQ (listelement * listpointer)
{
    cout<<"Element removed is"<<listpointer -> dataitem;
```

```
tempp = listpointer -> link;
free (listpointer);
return tempp;
}
void queue ::PrintQueue (listelement * listpointer)
{
if (listpointer == NULL)
cout<<"queue is empty!\n";
else
while (listpointer != NULL) {
cout<<listpointer -> dataitem<<" ";
listpointer = listpointer -> link;
}
cout<<"\n";
}
void queue::ClearQueue (listelement * listpointer)
{
while (listpointer != NULL) {
listpointer = DeleteQ (listpointer);
}
}
```

Q6. Design a template class for a circular queue. Make suitable assumptions, if any.

Ans. //circular queue implementation

```
#include<iostream.h>
#define MAX 10
#include<conio.h>
template<class T>
class queue
{
private:
T arr[MAX];
T front,rear;
public:
queue()
```

```
{  
front=-1;  
rear=-1;  
}  
void addq(T item)  
{  
if((rear==MAX-1&&front==0) || (rear+1==front))  
{  
cout<<endl<<"Queue is full";  
return;  
}  
if(rear==MAX-1)  
rear=0;  
else  
rear=rear+1;  
arr[rear]=item;  
if(front==-1)  
front=0;  
}  
T delq()  
{  
T data;  
  
if(front==-1)  
{  
cout<<endl<<"Queue is Empty";  
return NULL;  
}  
else  
{  
data=arr[front];  
  
if(front==rear)  
{
```

```
front=-1;
rear=-1;
}
else
{
if(front==MAX-1)
front=0;
else
front=front+1;
}

return data;
}
}
};

void main()
{
queue <int>a;
clrscr();
a.addq(11);
a.addq(12);
a.addq(13);
a.addq(14);
a.addq(15);
a.addq(16);
a.addq(17);
a.addq(18);
a.addq(19);
a.addq(20);
a.addq(21);
T i=a.delq();
cout<<endl<<"item deleted="<<i;
i=a.delq();
cout<<endl<<"item deleted="<<i;
```

```
i=a.delq();
cout<<endl<<"item deleted="<<i;
getch();
}
```

Q7. Write template class for stack.

Ans. #include<iostream.h>

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
template <class t>
class st
{
public :
t stack[20];
int top;
t val;
int choice;
st()
{top=-1;
}
int pop(t stack[ ],int &top);
void push(t stack[ ],int &top,t val,int n);
void display(t stack[ ],int top);
};
//Insert in stack
template <class t>
void st<t>::push(t stack[ ],int &top,t val,int n)
{
if(top < n)
{
top++;
stack[top]=val;
}
else
```

```
cout<<"Stack is full";
}
//deletion in stack
template <class t>
int st<t>::pop(t stack[ ], int &top)
{
int delvalue;
if(top>=0)
{
delvalue=stack[top];
top--;
return delvalue;
}
else
{
cout<<"\n Stack is empty";
return(-9999);
}
}

//Display in stack
template <class t>
void st<t>::display(t stack[ ], int top)
{
int i;
if(top>=0)
{
cout<<"\n";
for(i=top; i>=0; i--)
{
cout<<stack[i];
}
getch();
fflush(stdin);
}
```

```
}
```

```
void main()
```

```
{
```

```
st<char> obj;
```

```
do
```

```
{
```

```
cout<<"\n      Menu      ";
```

```
cout<<"\n 1 ==> Push   2 ==> Pop";
```

```
cout<<"\n 3 ==> Display  4 ==>Quit";
```

```
cout<<"Enter your choice";
```

```
cin>>obj.choice;
```

```
switch(obj.choice)
```

```
{
```

```
case 1: cout<<"Enter the value to pushed";
```

```
cin>>obj.val;
```

```
obj.push(obj.stack, obj.top, obj.val, 30);
```

```
break;
```

```
case 2: obj.val=obj.pop(obj.stack, obj.top);
```

```
if(obj.val!= -9999)
```

```
cout<<"\n the popped last value of stack is "<<obj.val;
```

```
break;
```

```
case 3: obj.display(obj.stack, obj.top);
```

```
break;
```

```
case 4: exit(0);
```

```
default:
```

```
cout<<"wrong value";
```

```
}
```

```
}while(obj.choice!=4);
```

```
}
```

```
}
```

Feedback is the breakfast of Champions.

Ken Blanchard

You can Help other students.

"Inform any error or mistake in this book."

**We and Universe
will reward you for Your Kind act.**

**Email at : feedback@gullybaba.com
or**

WhatsApp on 9350849407

8

Exception Handling

Exception Handling

The extensibility of C++ can increase substantially the number and kind of errors that can occur. The feature presented here enable programmers to write clearer, more robust, more fault tolerant programs.

There are many popular means of dealings with errors. Most commonly known as error handling. Code is interspersed throughout a system code. Errors are dealt with at the place in the code where the errors can occur. The advantage of this approach is that a programmer reading code can see the error processing in the immediate vicinity of the code and determine if the proper errors checking has been implemented.

The problem with this scheme is that code became polluted with the error processing. It becomes more difficult for a programmer concerned with the application itself to read the code and determine if the code is functioning correctly. This makes it more difficult to understand and to maintain the code. Some common example of exception are failure of new to obtain a requested amount of memory, an out range, arithmetic overflow, division by zero.

C++ exception handling features enables the programmer to remove the error handling code from the main line of a program execution. This improves the program reading and modifiability.

Exception handling is not designed to deal with asynchronous, situations such as disk I/O.

Use of Exception Handling

Exception handling is used in situations in which the system can recover from the error causing the exception. The recovery procedure is called an exception handler. Exception handling is typically used when the error will be dealt by different part of the program from which detected the error.

Exception handling is especially appropriate for situation in which the program will not be able to recover, but needs to provide orderly clean up, then shutdown “gracefully”.

- Use exceptions for errors that must be processed in a different scope from which they occur. Use other means of error handling for errors that will be processed in the scope in which they occur.
- Avoid using exception handling for purposes other than error handling because this can reduce program clarity.

Performance unit

- Although it is possible to use exception handling for purposes other than error handling, this can reduce program performance.
- Exception handling is generally implemented in compilers in such a manner that when an exception does not occur, little or no overhead is imposed by the presence of exception handling code.
- Another reason exception can be dangerous as an alternative to normal flow of control is that the stack is unwound and resources allocated prior to the occurrence of the exception may not be freed. This problem can be avoided by careful programming.

Example of new throwing bad_alloc when memory is not allocated.

```
# include <iostream.h>
using std :: cout;
using std :: cin;
# include <new>
using std :: bad_alloc;

int main ()
{
    double * ptr [50];
    try
```

```
{  
for (int I=0; I< 50; I++)  
{  
ptr (I) = new double [ 500000];  
cout << "Allocate 500000 double in ptr" << I;  
}  
}  
catch (bad_alloc exception)  
{  
cout << "Exception Occurred" << exception.what () << endl;  
}  
return 0;  
}
```

Programming Errors

- Aborting a program could leave a resource in a state in which other program would not be able to acquire the resource, hence the program would have a so-called “resource-leak”.
- Exception should be thrown only within try block. An exception thrown outside a try block causes a call to terminate.
- It is possible to throw a conditional expression. But be careful because promotion rules may cause the value returned by the conditional expression to be of a different type than you may expect for example when throwing an int or double form the same conditional expression, the conditional expression will convert the int into Double. Therefore the result will always be caught by a catch with a double argument rather than sometimes catching double and sometime catching int.
- Specifying a Comma separated list of catch argument is a syntax error.
- Placing catch () before other catch blocks would prevent those blocks from ever being executed, catch () must be placed last in the list of handlers following a try block.
- Assuming that after an exception is processed control return to the first statement after the throw is a logic error.
- Placing an empty throw statement outside a Catch handler; executing such a throw causes a call to terminate.
- User defined exception classes need not be derived from class exception. Thus writing catch (exception) is not guaranteed to catch all exception a program may encounter.

Process of Error Handling

The exception mechanism of C++ uses three new *keywords*: *throw*, *catch*, and *try*. Also, we need to create a new kind of entity called an *exception class*.

Suppose we have an application that works with objects of a certain class. If during the course of execution of a member function of this class an error occurs, then this member function informs the application that an error has occurred. This process of informing is called *throwing* an exception. In the application we have to create a separate section of code to tackle the error. This section of code is called an *exception handler* or a *catch block*. Any code in the application that uses objects of the class is enclosed in a *try block*. Errors generated in the try block are caught in the catch block. Code that doesn't interact with the class need not be in a *try block*. The following code shows the organisation of these blocks. It is not a working program, but it clearly shows how and where the various elements of the exception mechanism are placed.

```
class sample
{
public:
//exception class
class errorclass
{
};

void fun()
{
if(some error occurs)
throw errorclass(); //throws exception
}

//application
void main()
{
//try block
try
{
sample s;
s.fun();
}
```

```
catch(sample::errorclass)           //exception handler or catch block
{
//do something about the error
}
}
```

Here *sample* is any class in which errors might occur. An exception class called *errorclass*, is specified in the *public* part of *sample*. In *main()* we have enclosed part of the program that uses *sample* in a *try* block. If an error occurs in *sample::fun()* we throw an exception, using the keyword *throw* followed by the constructor for the *errorclass*:

```
throw errorclass();
```

When an exception is thrown control goes to the *catch* block that immediately follows the *try* block.

○○○

Must Read

अवश्य पढ़ें



GULLYBABA PUBLISHING HOUSE PVT. LTD.

New Syllabus Based

100%

Guidance for
IGNOU EXAM

IGNOU HELP BOOKS

B.A., B.COM, B.A. FOUNDATION, M.A., M.COM.,
BCA, B.ED., M.ED., AND OTHER SUBJECTS

IAS, PCS, UGC & All University Examinations

Chapter wise Researched

QUESTIONS & ANSWERS

Solved papers & very helpful for your assignments preparation के लिए रामबाण

Hindi & English Medium

GULLYBABA PUBLISHING HOUSE PVT. LTD.

2525/193, 1st Floor, Onkar Nagar-A, Tri
Nagar, Delhi-110035, (From Kanhaiya Nagar
Metro Station Towards Old Bus Stand)

Email : Info@gullybaba.com
Web : www.gullybaba.com

Join us on Facebook at
IGNOU Helpbooks

For any Guidance &
Assistance Call:

9350849407

Supplementary

Q1. Is constructor overloading different from ordinary function overloading? How?

Ans. Constructor overloading implies that we use same name of the class at two or more instances with varying number of arguments.

Whenever we have to put values in data members of a class we call different constructors by making objects and supplying arguments values wherever necessary. Depending upon number of argument constructors are called:

In function overloading same function name is used but we have:

- Varying number of arguments.
- Different types of arguments.

We can also have varying number of arguments in constructor case but different type of arguments is not supported because data is only of one type for a single variable.

E.g. in function overloading.

```
void print(int, float);
void print(int, int);
class:classeg{
int a;
public:
classeg(int);
```

```
classeg(float);           //not possible
}
```

Also there are different cases in the limitation of number of arguments.

- In case of constructors we can have only maximum number of arguments equal to number of classes data variables normally.
- In case of function overloading we can have any number of arguments.

E.g.

```
class classeg
{
int a;
float b;
char c;
public:
classeg()
{
a=0; b=0; c='a';
}
classeg(int a1, float b1)
{
a=a1;
b=b1;
}
classeg(int a1, float b1, char c) // maximum three arguments as we have
3                                // variables.
{
a=a1;
b=b1;
c=c1;
}
```

Q2. Design a function template to perform matrix addition of two given integer matrices, two floating point matrices and double precision value matrices separately.

Ans. #include<iostream.h>

```
#include<conio.h>
```

```
#define SIZE 10
```

```
template<class T>
void add(T *a,T *b,int m,int n)
{
T r[10][10];
int c=0;
for(int i=0;i<m;i++)
{
for(int j=0;j<n;j++)
if(c<=(m*n-1))
{
r[i][j]=a[c]+b[c];
c++;
}
}
cout<<"Matrix after addition :\n";
for( i=0;i<m;i++)
{
for(int j=0;j<n;j++)
cout<<r[i][j]<<" ";
cout<<"\n";
}
}
void main()
{
int m,n,x,a[10][10],b[10][10], iterate=0;
clrscr();
cout<<"Matrix Addition Program \n";
do{
cout<<"\n Enter number of Rows";
cin>>m;
if(m>SIZE)
cout<<"\n Too many rows ! Maximum 10 allowed";
}
while(m>SIZE);
do{
```

```
cout<<"\n Enter number of Columns";
cin>>n;
if(n>SIZE)
cout<<"\n Too many columns ! Maximum 10 allowed";
}
while(n>SIZE);
do{
cout<<"\nEnter type of matrix :\n";
cout<<"1 for integer \n";
cout<<"2 for floating point or decimal \n";
cin>>x;
if(x<1 || x>2)
cout<<"Wrong Type";
}
while(x<1 || x>2);
float a1[10][10],b1[10][10];
double a2[10][10],b2[10][10];
for(int k=1;k<=2;k++)
{
cout<<"\n Enter elements of matrix"<<k;
iterate++;
if(x==1)
{
for(int i=0;i<m;i++)
{
cout<<"\n Enter elements of row"<<i+1;
for(int j=0;j<n;j++)
{
cout<<"\n Column:"<<j+1;
if(iterate==1)
cin>>a[i][j];
else
cin>>b[i][j];
}
}
}
```

```
if(iterate==2)
add(*a,*b,m,n);
}

else
if(x==2)
{
for(int i=0;i<m;i++)
{
cout<<"\n Enter elements of row"<<i+1;
for(int j=0;j<n;j++)
{
cout<<"\n Column:"<<j+1;
if(iterate==1)
cin>>a1[i][j];
else
cin>>b1[i][j];
}
}

if(iterate==2)
add(*a1,*b1,m,n);
}

else
{
for(int i=0;i<m;i++)
{
cout<<"\nEnter elements of row"<<i+1;
for(int j=0;j<n;j++)
{
cout<<"\n Column:"<<j+1;
if(iterate==1)
cin>>a2[i][j];
else
cin>>b2[i][j];
}
}
```

```

if(iterate==2)
add(*a2,*b2,m,n);
}
}

//outer for ends.

getch();
}

//main ends

```

Q3. What are the pros and cons in declaring a friend class in a program?

Ans. Pros and Cons in declaring a friend function:

- You increase flexibility of access but at the same time it is against the concept of data hiding.
- This method is often desirable when there is no other option, at the first instance thereby producing an inefficient code.

Q4. Explain the concept of polymorphism. How has it been implemented in C++? Give an example of its use.

Ans. Using operators and functions in different ways depending upon what they are operated on, is called polymorphism.

It has been implemented in C++ using Functions.

- In function overloading we have different number of arguments and different types of arguments with same function name, i.e. same form not forming different functions.
- Operator overloading helps to operate on user defined data type (i.e. depending upon what they are operated on).
- Virtual Function: Allow same function name in different cases.

These functions can be manipulated using a single statement in main function.

E.g.

```

shape*arr[50];
for(i=0; i<50; i++)
arr[1]->draw();
//Different versions of draw function are
//Called depending upon arr[i] contents.

```

Q5. Describe the following with the help of an example each:

(a) “this” pointer

Ans. When you are writing code in a member function, the “variable” this contains a pointer to the current object. Any member element, _x, can be referenced in two ways:

```
_x = 10;
this ->_x = 10;           // same as above.
```

References to members are usually written without this because there is no need to use it, however, there are situations where this is needed. Returning or passing a pointer to the object.

(b) Inheritance and its need

Ans. Now consider a situation, where two classes are generally similar in nature with just couple of differences. Would you have to rewrite 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 subclass.

For 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.

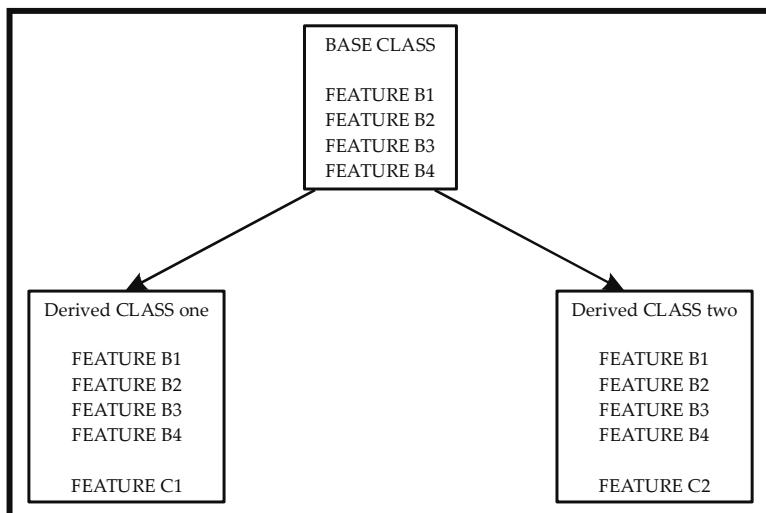


Fig. 9.1: Inheritance

In the above figure, 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.

Now 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 subclass ‘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.

(c) Functions with varying number of arguments in C++

Ans. A C++ function prototype can declare that one or more of the function’s parameters have default values. What a call to the function omits the corresponding arguments, the compiler inserts the default values where it expects to see the arguments.

We can declare default values for arguments in a C++ function prototype in the following way:

```
Void myfunc(int=5, double=1.23);
```

The expression declare default values for the arguments. The C++ compiler substitutes the default values if you omit the arguments when you call the function. We can call the function by using any of the following ways:

myfunc(12,3.45);	// overrides both defaults
myfunc(3);	// effectively func(3, 1.23);
myfunc();	// effectively func(5,1.23);

To omit first parameter in these examples, you must omit the second one; however, you can omit a parameter unless you omit the parameters to its right.

Here is the example:

```
#include<iostream.h>
void show(int=1,float=2.3,long=4);
main()
{
show();           //all three parameters default
show(5);         //provide 1st parameter
show(6,7.8);    //provide all three parameters
}
```

```
void show(int first, float second, long third)
{
    cout<<"\nfirst="<<first;
    cout<<"second="<<second;
    cout<<"third="<<third;
}
```

(d) Static/Early Binding

Ans. It refers to events that occur at compile time. In essence, early binding occurs when all information needed to call a function is known at compile time. (early binding means that an object and function call are bound during compilation.) Examples of early binding include normal function calls (including standard library functions), overloaded function calls, and overloaded operators. The main advantage early binding is efficiency. Because all information necessary to call a function is determined at compile time, these types of function calls are very fast.

Q6. How is data abstraction related to/different encapsulation and polymorphism? Discuss with the help of an example.

Ans. The property of being a self-contained unit is called encapsulation. The idea that the encapsulated unit can be used without knowing how it works is called data hiding.

Encapsulation is the principle by which related contents of a system are kept together. It minimises traffic between different parts of the work and it separates certain specific requirements from other parts of specification, which use those requirements.

E.g.

```
class A {
    int a;
    public:
        void getdata();
    };
    void main()
    {
        A a1;
    }
```

Here, a1(object) is an encapsulated unit, containing attributes and functionality together.

The important advantage of using encapsulation is that it helps minimise rework when developing a new system. The part of the work,

which is prone to change, can be encapsulated together. Thus any changes can be made without affecting the overall system and hence changes can be easily incorporated.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called classes.

Q7. What are the access control violations in the following program segment:

```
class All_public {  
public:  
    int all_pub;  
    All_public () {}  
    void print ();  
};  
  
class Inherited : Private All_public {  
private:  
    int some_private_data;  
public:  
    int read();  
    void print();  
}  
  
void main ()  
{  
    All_public first;  
    Inherited second ;  
    second.all_pub = 10;  
    second.all_public::print();  
    first.all_pub = 50;  
    second. some_ private _ data = 200;  
}
```

Ans. The following access violations are there:

- All_public::all_pub is not accessible because in this example our subclass Inherited is privately inherited not publicly.

- All_public::print() is not accessible in void main because of private inheritance.
- Inherited::Second.some_private_data=200; is not accessible because it is defined private in subclass Inherited.

Q8. Why do you need references in C++? How can a reference be used in parameter passing? Describe with the help of an example.

Ans. C++ contains a feature that is related to pointer called reference. A reference is essentially an implicit pointer. There are three ways that a reference can be used:

- As a function parameter.
- As a function return value, or
- As a stand-alone reference.

The most important use of a reference is to allow us to create functions that automatically use call-by-reference parameter passing.

The syntax for this is:

```
int RefFunction (int &Arg1, float &Arg2);
```

Note that each argument is prefixed by an ampersand. Such a function is called in exactly the same way as one where the arguments are passed by value:

```
ReturnedInt = RefFunction (Int, Float2);
```

The difference between IntFunction and RefFunction is that the *addresses* of the arguments are passed in RefFunction, and not the actual values. As such, no extra memory is reserved within the function, since no copy of the arguments need to be made. This variable Arg1 within the function and the variable Int1 passed to it are identical: they share the same address in memory. In this case, any change made to Arg1 within the function will also affect Int1 in the calling routine.

Q9. Write a program that write an object to a disk file.

Ans. #include <fstream.h>

```
void main ()  
{  
    struct employee  
    {  
        char name[20];  
        int age;  
        float basic;  
        float gross  
    };
```

```
};

employee e;
char ch = 'y';
\\ create file for output
ofstream outfile;
outstream outfile;
outfile.open ("EMPLOYEE.DAT", ios::out | ios::binary);
while (ch == 'y')
{
    cout << endl << "enter a record";
    cin >> e.name >> e.age >> e.basic >> e.gross;
    outfile.write ((char *) &e, sizeof (e));
    cin >> ch ;
}
outfile.close()
\\ create file for input
ifstream in file ;
infile.open (EMPLOYEE.DAT", ios ::out | ios::binary) ;
while (infile.read ((char *) &e, sizeof (e)))
{
    cout << endl << e.name << '\t' << e.age << '\t'
    << e.basic << '\t' << e.gross ;
}
```

Q10. What is the meaning of having “const and classname” as a parameter in a function? Describe.

Ans. If we want that an argument passed to an ordinary function by reference should not get modified then the argument should be made const in the function declaration (and definition). This holds good for member functions as well.

```
#include <iostream.h>
class sample
{
private:
int data;
public:
sample ()
{
```

```
data = 0
}
void changedata()
{
data = 10;
}
void showdata() const
{
cout << endl << "data = " << data;
}
void add (sample cont &s, sample const &t)
{
data = s. data + t.data;
}
void getdata( )
{
cin >> data;
}
};

void main( )
{
sample s1
s1. Changedata( )
sample s3;
s3.add (s1, s2);
s3.showdata( );
}
```

In this program the arguments to the function add() are passed by reference, and we want to make sure that add() won't modify these arguments. To ensure this we have declared the arguments as const.

Q11. Give an example of a problem that is suitable to be programmed using an OOP language.

Ans. Problem best suited to be programming using an OOP language is that of a game having players acting as objects as in cricket.

The different players have common functions but associated data such as fielding points may vary. Common functions include catching, throwing and batting. A team may represent a class having players as objects.

The idea behind object-oriented programming is that a computer program is composed of a collection of individual units, or *objects*, as opposed to a traditional view in which a program is a list of instructions to the computer. Each object is capable of receiving messages, processing data, and sending messages to other objects.

Object-oriented programming is claimed to give more flexibility, easing changes to programs, and is highly suitable for large scale software engineering. Furthermore, proponents of OOP claim that OOP is easier to learn for those new to computer programming than previous approaches, and that the OOP approach is often simpler to develop and to maintain, lending itself to more direct analysis, coding, and understanding of complex situations and procedures than other programming methods.

Q12. Write at least 5 operators of C++ that can be overloaded. Show operator overloading of any one of these operators with the help of an example.

Ans. By overloading operators we can give additional meaning to operators like +, *, -, <=, >=, etc. which by default are supposed to work only on standard data types like *ints*, *floats*, etc.

Here is the program which implements complex number addition and multiplication using overloaded operators rather than through member functions like *add_complex()* and *mul_complex()*.

```
#include <iostream.h>
class complex
{
private
    float real, imag ;
public:
```

```
complex()
{
}
```

```
complex (float r, float i)
{
}
```

```
real = r ;
imag = i ;
}
```

```
void getdata( )
}
```

```
float r, i;
cout << endl << "Enter real and imaginary that" ;
cin >> r >> i ;
real = r ;
imag = i ;
}

void setdata (float r, float)
{
real = r ;
imag = i ;
}
void displaydata( )
{
cout << endl << "real=" << real;
cout << endl << "imaginary=" << imag;
}
complex operator + (complex c)
{
complex t;
t.real = detail on this real
t.imag = imag + c.imag;
}

complex operator * (complex c)
{
complex t ;

t.real = real * c.real - image * c.image ;
t.imag = real * c.imag + c.real * image ;
}
};

void main( )
{
```

```
complex c1, c2 (1.5, -2.5), c3, c4 ;  
c1.setdata (2.0, 2.0) ;  
c3.displaydata( ) ;  
  
c4.getdata( ) ;  
complex c5 (2.5, 3.0), c6 ;  
c6. c4 * c5 ;  
c6.displaydata( ) ;  
}
```

Now examine the call and the definition of the overloaded '+' operator more closely.

```
\ \ definition  
complex operator + (complex c)  
{  
complex t ;  
t.real = real + c.real ;  
t.imag = imag + c.imag ;  
return t ;  
}  
// call  
c3 = c1 + c2
```

When the *operator +()* function is called, the object *c2* is passed to it and is collected in the object *c*. As against this, the object *c1* gets passed to it automatically. This becomes possible because the statement *c3 = c1 + c2* is internally treated by the compiler as:

```
c3 = c1. operator + (c2) ;
```

That should give you an idea why the first operand in case of a overloaded binary operator function becomes available automatically, whereas, the second operand needs to be passed explicitly.

It the definition of the overloaded + operator function when we use the statement

```
st.real = real + c.real ;
```

real refers to the one that belongs to the object using which the operator function has been called.

Q13. Define the terms Loading and Linking. Give an example of each.

Ans. Loading: Is the process of placing a program from hard disk to RAM for a process to begin. This is done at start of the process.

Linking: Is the process of accessing a program or chunk of it from disk in between a process. This is done while a process is in continuation.

Taking example of Java Language: During the loading process, the virtual machine must parse the stream of binary data that represents the type and build internal data structures. At this point, certain checks will have to be done just to ensure the initial act of parsing the binary data won't crash the virtual machine. During this parsing, implementations will likely check the binary data to make sure it has the expected overall format. Parsers of the Java class file format might check the magic number, make sure each component is in the right place and of the proper length, verify that the file isn't too short or too long, and so on. Although these checks take place during loading, before the official verification phase of linking, they are still logically part of the verification phase. The entire process of detecting any kind of problem with loaded types is placed under the category of verification.

Another check that likely occurs during loading is making sure that every class except Object has a superclass. This may be done during loading because when the virtual machine loads a class, it must also make sure all of the class's superclasses are loaded also. The only way a virtual machine can know the name of a given class's superclass is by peering into the binary data for the class. Since the virtual machine is looking at every class's superclass data during loading anyway, it may as well make this check during the loading phase.

Another check—one that likely occurs after the official verification phase in most implementations—is the verification of symbolic references. The process of dynamic linking involves locating classes, interfaces, fields, and methods referred to by symbolic references stored in the constant pool, and replacing the symbolic references with direct references. When the virtual machine searches for a symbolically referenced entity (type, field, or method), it must first make sure the entity exists. If the virtual machine finds that the entity exists, it must further check that the referencing type has permission to access the entity, given the entity's access permissions. These checks for existence and access permission are logically a part of verification, the first phase of linking, but most likely happen during resolution, the third phase of linking. Resolution itself can be delayed until each symbolic reference is first used by the program, so these checks may even take place after initialisation.

Q14. Draw a comparative chart with respect to any 5 features of an Object Oriented Programming language for C++ and Java.

Ans.

Table 9.1

C++	Java
Pointers are used extensively.	Pointers are not used to avoid security breach.
Package concept is not present.	Packages are used intensively.
Virtual functions are present.	No Virtual functions as pointer concept is not present
Its not necessary that program name be same as classname in it.	It is necessary that program name be same as class name in it.
It is not platform independent.	It is platform independent.

Q15. What is a friend class? How is it declared? Explain its use with an example.

Ans. A class can be made friend of other class. To grant friendship to another class, write the keyword friend followed by the class name. The keyword class is optional. Note that this declaration also implies a forward declaration of the class to which the friendship is being granted. The implication of this declaration is that all of the member functions of the friend class are friend functions of the class that bestows the friendship.

class Aclass

```
{
public :
    :
    :
    :
```

friend class Bclass; // Friend declaration

private :

int Avar;

};

class Bclass

```
{
public :
```

:
:

```
void fn1(Aclass ac)
{
Bvar = ac. Avar; //Avar can be accessed
}
private :
int Bvar;
};

void main()
{
Aclass aobj;
Bclass bobj;
bobj.fn1(aobj);
}
```

The program declares class Bclass to be a friend of Aclass. It means that all member functions of Bclass have been granted direct access to all member functions of Aclass.

Q16. Write short notes on the following:

(i) Containership

Ans. When an object class contains another object class is called as containership. E.g.

class A

1

• • • •

class B {....} b;

• • • •

1

(ii) SIZEOF Operator

Ans. The sizeof() operator is a compile time operator and when used with an operand returns the number of bytes the operand occupies. The operand may be a variable constant or a data type quantifier. Example: sizeof(int) or sizeof(num), etc.

If a is an integer type variable then

```
cout<<sizeof(a); //It will give 2 as output as integer  
//occupies 2 bytes in memory.
```

(iii) Bitwise Operator

Ans. They are used for manipulating the data at bit level. These operators are used for testing the bits or shifting them right or left. Bitwise operators are applicable to integer datatypes only and Bitwise operators are not applicable to float or Double datatypes.

(iv) Nested Classes

Ans. If the class contain objects of other classes as its members, then this type of class is known as nested class.

E.g.

```
class alpha {-----};  
class beta {-----};  
class gamma {  
alpha a;           //object of alpha class  
beta b;          //object of beta class  
-----};
```

Here gamma class is a nested class as it contains the objects a and b of alpha and beta classes.

(v) Copy constructor

Ans. A copy constructor is used to declare and initialise an object from another object of the same class.

Example: If I1 is the first object of integer class and I2 is the second object of integer then the statement.

```
integer I2 = I1      //It is the way a copy constructor is implemented  
in C++.
```

(vi) Polymorphism

Ans. Using Operators and Functions in different ways by same name is called polymorphism. The word polymorphism means existing in different forms. Polymorphism can be implemented in C++ through: (a) Function Overloading and (b) Operator Overloading

(vii) Activity Diagram

Ans. Activity diagram represents the business and operational step by step work flows of components in a system. We make activity diagram to emphasise the behaviour of a system.

(viii) Python

Ans. Python is an object-oriented scripting language used for writing system utilities and Internet scripts. It is also used as a glue language for integrating components in C and C++. Python is an interpreted language that compiles to byte code and requires a "Virtual Machine" for runtime execution.

(ix) Comma Operator in C++

Ans. Comma operator is simple operator in C++ which comes at the end position of operators precedence table in C++. We can use comma operator in the following manners:

Example 1:

```
int num1, num2, num3;
```

Here comma operator is used to declare three variables num1, num2 and num3 of same data type, i.e. integer.

Example 2:

```
ans = (a = 10, b = 20, a + b);
```

Here ans = 30 using comma operator.

(x) Protected inheritance

Ans. This allows derived classes of derived classes to know about the inheritance relationship. Thus your grand kids are effectively exposed to your implementation details. This has both benefits (it allows derived classes of the protected derived class to exploit the relationship to the protected base class) and costs (the protected derived class can't change the relationship without potentially breaking further derived classes). Protected inheritance uses the : protected syntax.

(xi) Private and public member functions

Ans. Member functions can be private or public, just as member variables are. Public member functions can be accessed anywhere in the program, even outside the class. Private member functions are for the sole use of the class itself. Usually member data are made private while functions (or methods) are made public. There might be instances where you might want to make certain functions private (i.e. you may not want the user to directly access these functions). Private functions can only be called from within public member functions. These functions are also called 'helper functions'.

(xii) && and || Operator

Ans. The logical operators **&&** and **||** are used when evaluating two expressions to obtain a single relational result. The operator **&&** corresponds with Boolean logical operation AND. This operation results true if both its two operands are true, and false otherwise. The following panel shows the result of operator **&&** evaluating the expression **a && b**:

&& operator

a	b	a && b
true	true	true
true	false	false

false	true	false
false	false	false

The operator `||` corresponds with Boolean logical operation OR. This operation results true if either one of its two operands is true, thus being false only when both operands are false themselves. Here are the possible results of `a || b`:

`||` operator

a	b	<code>a b</code>
true	true	true
true	false	true
false	true	true
false	false	false

E.g. `((5 == 5) && (3 > 6))` // evaluates to false (true `&&` false).

`((5 == 5) || (3 > 6))` // evaluates to true (true `||` false).

(xiii) `++i` and `i++`

Ans. Both `i++` and `++i` exist since both can only be used to add one and both are the same length. The reason for the two alternatives is that these are not really intended to be used as stand alone statements but are really designed to be able to be incorporated into more complex statements where you actually update more than one variable in the one statement.

Probably the simplest such statement is as follows:

`j = i++;`

This statement updates the values of both of the variables `i` and `j` in the one statement. The thing is that while `++i` and `i++` do the same thing as far as updating `i` is concerned they do different things with regard to updating other variables. The above statement can be written as two separate statements like this:

`j = i;`

`i += 1;`

Note that combining those together means we have eight characters instead of 13. Of course the longer version is much clearer where it comes to working out what value `j` will have. Now if we look at the alternative:

`j = ++i;`

This statement is the equivalent of the following:

`i += 1;`

`j = i;`

This of course means that `j` now has a different value to what it had in the first example. The position of the `++` either before or after the variable

name controls whether the variable gets incremented before or after it gets used in the statement that it is used in.

(xiv) Preprocessor directives

Ans. When the preprocessor encounters this directive, it replaces any occurrence of “identifier” in the rest of the code by “replacement”. This replacement can be an expression, a statement, a block of statement, etc. The preprocessor does use # defines. Its format is # define identifier replacement.

E.g.

```
#include <iostream.h>
#define getmax(a, b) [(a)>(b)?(a):(b)] int main( )
{
int x=5, y ;
y=getmax(x, 2);
cout<<y<<endl;
cout<<getmax(7, x)<< endl;
return 0;
}
```

(xv) Base and derived classes

Ans. Base Class: A class can be built as another class that is already defined and it is existing. This existing class is called the Base Class or parent class or Super Class.

- A base class is a class that is created with the intention of deriving other classes from it.

Derived Class: The derived class inherits all the properties of Base Class. That is to say that the child class inherits all the member variables and methods of the Base Class.

- The child class can have its own member variables and methods.

Syntax for Deriving Derived Class:

Class child Class: Public Base Class

```
{
--- }
--- } // Body of Child Class
--- }
```

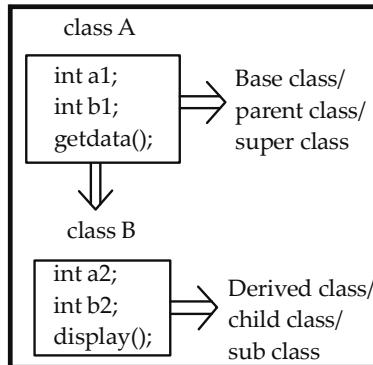
Example:

Fig. 9.2

(xvi) Dynamic Binding

Ans. Polymorphism can be implemented using operator and function overloading, where the same operator and function works differently on different arguments producing different results. The overloaded member functions are selected for invoking by matching arguments, both type and number. This information is known to the compiler at the compile time and, therefore, compiler is able to select the appropriate function for a particular call at the compile time itself. This is called early binding, static binding, static linking or compile time polymorphism.

Q17. Explain any one way of using the keyword “const” in C++? Explain with the help of one example each.

Ans. The keyword const (for constant), if present, precedes the data type of a variable. It specifies that the value of the variable will not change throughout the program. Any attempt to alter the value of the variable defined with this qualifier will result into an error message from the compiler. *const* is usually used to replace *#defined* constants.

The following program shows the usage of const.

```

#include <iostream.h>
void main()
{
float r, a ;
const float PI = 3.14 ;

cin >> r ;
a = PI * r * r ;
  
```

```
cout << endl << "Area of circle =" << a ;
}
```

If a *const* is placed inside a function its effect would be localised to that function, whereas, if it is placed outside all functions then its effect would be global. We cannot exercise such finer control while using a *#define*.

Q18. What is the purpose of class-diagrams? Explain with the help of an example.

Ans.

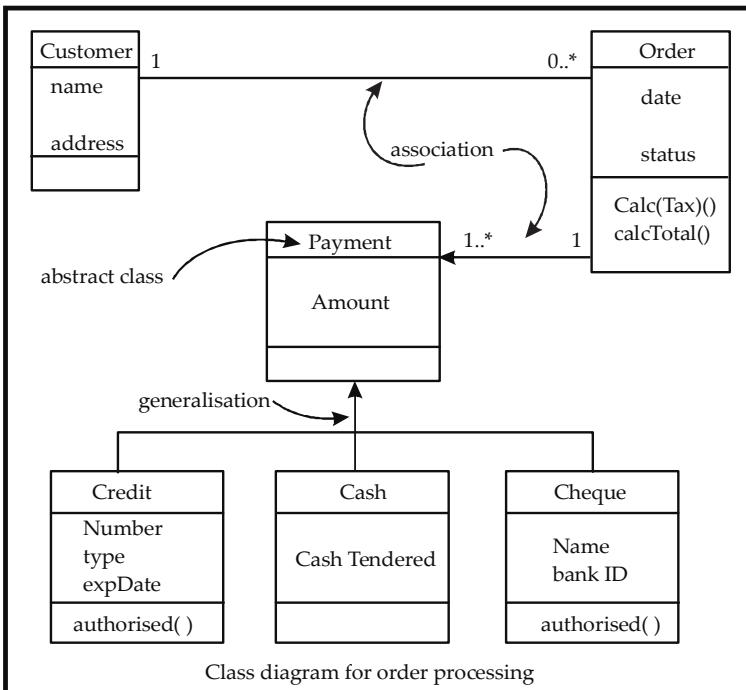


Fig. 9.3

Class diagrams serve the purpose of pictorial representation of data members and function in the class and relationships among the class is, e.g. inherited classes or classes at base level that have logical relationships.

Inheritance: also called generalisation

Relationships: also called associations

e.g. The above classes diagram

For order processing classes credit,

Cash and cheque are inherited from class payment.

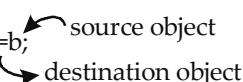
Class customer has 1 to many relations with class order and class order has 1 to many (start at 1.) relation with class payment.

Q19. "The assignment operator can not be overloaded through a friend function". Is this true? Justify your answer.

Ans. It is true that assignment operator cannot be overloaded through a friend function. The reason is derived from the fact that: (i) Writing an expression involving assignment operator in a program will tend to invoke copy constructor or overloaded assignment operator if operation involves objects of same class. (ii) If operation involves objects of different class where use of friend function could be meaningful, then also compiler only searches for a copy constructor or a conversion routine. That conversion can be as a conversion function or in the destination data class, i.e. as a constructor.

E.g.

```
class A {  
};  
class B {  
};  
void main()  
{  
A a;  
B b;
```

a=b;


Q20. What is dynamism in the context of the object-oriented programming paradigm? What are the different types of dynamism existing in the object-oriented programming paradigm? Explain each type with the help of an example.

Ans. 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 constraints on their actions by the demand of the computer language and the requirements of the compiler and linker. In other words, freeing the world of users from the programming environments.

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

(1) Dynamic Typing: Dynamic typing is the basis of dynamic binding. It allows associations between object to be determined at run time, rather than fixing them to be encoded in state compile time.

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 sense customised by the object of indeterminate type (indeterminate in source code, but, not at run time).

(2) 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 object 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. You also need not assign each alternative procedure a different name.

Dynamic binding is possible even if dynamic typing does not exist in a programming language but it is not very useful. For e.g. 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 well 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 a 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 class 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) Dynamic Loading: dynamic loading encourages modular development with an added flexibility that the entire program may not be developed before a program can be used, the program 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.

One of the important benefits of dynamic loading is that it makes applications extensible.

Q21. What is UML? What are the benefits of UML for system designers?**List types of UML diagrams.**

Ans. The Unified Modeling Language (UML) is a language for specifying, visualising, 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 compiler system.

Benefits of UML are as follows:

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

The UML define various types of diagrams:

- **Use case Diagrams:** Use cases are used in almost every project they are helpful in exposing requirements and planning the project.
- **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. They 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.
- **Interaction Diagrams:** Interaction diagrams model the behaviour of use cases by describing the way groups of objects interact to complete the task. They are used when you want to model the behaviour of several objects in a use case. They demonstrate how the objects collaborate for the behaviours.
- **State Diagrams:** State diagrams are used to describe the behaviour of a system in the state and response of a class. They describe the behaviour of a class in response to external stimuli.

State diagrams describe all the possible states of an object as events occur. Each diagram usually represents objects of a single class and tracks the different states of its objects through the system.

Q22. Explain the concepts of encapsulation and information hiding in an object oriented paradigm, with the help of an example in C++.

Ans. In computer science, the principle of information hiding is the hiding of design decisions in a computer program that are most likely to change, thus protecting other parts of the program from change if the design decision is changed. Protecting a design decision involves providing a stable interface which shields the remainder of the program from the implementation (the details that are most likely to change). In modern programming languages, the principle of information hiding manifests itself in a number of ways, including encapsulation (given the separation of concerns) and polymorphism.

The term encapsulation is often used interchangeably with information hiding, while some make distinctions between the two.

Information hiding serves as an effective criterion for dividing any piece of equipment, software or hardware, into modules of functionality. For instance a car is a complex piece of equipment. In order to make the design, manufacturing, and maintenance of a car reasonable, the complex piece of equipment is divided into modules with particular interfaces hiding design decisions. By designing a car in this fashion, a car manufacturer can also offer various options while still having a vehicle which is economical to manufacture.

For instance, a car manufacturer may have a luxury version of the car as well as a standard version. The luxury version comes with a more powerful engine than the standard version. The engineers designing the two different car engines, one for the luxury version and one for the standard version, provide the same interface for both engines. Both engines fit into the engine bay of the car which is the same between both versions. Both engines fit the same transmission, the same engine mounts, and the same controls. The differences in the engines are that the more powerful luxury version has a larger displacement with a fuel injection system that is programmed to provide the fuel air mixture that the larger displacement engine requires.

With all of these changes, most of the car is the same between the standard version and the luxury version. The engineers design the car by dividing the task up into pieces of work which are assigned to teams. Each team then designs their component to a particular standard or interface which allows the sub-team flexibility in the design of the component while

at the same time ensuring that all of the components will fit together. As can be seen by this example, information hiding provides flexibility. This flexibility allows a programmer to modify functionality of a computer program during normal evolution as the computer program is changed to better fit the needs of users. When a computer program is well designed decomposing the source code solution into modules using the principle of information hiding, evolutionary changes are much easier because the changes typically are local rather than global changes.

C++ supports the properties of encapsulation and information hiding through the creation of classes. A class implements encapsulation by forming object of it. That object can call public member functions or public data of class using dot(.) operator.

Information hiding is advised using keyword private which doesn't let data to be accessible by object of the class.

Eg.

```
class A {  
    int p;  
    float f;  
public: void getdata()  
{  
    cin>>p >>f ;  
}  
};  
void main()  
{  
    A a1;  
    a1.getdata();           //accessible  
    a1.p=2;                //error (hidden)  
    a1.f=3.2;               //error (hidden)  
}
```

Here, a1 (object) is an encapsulated unit, containing attributes and functionality together.

Q23. Write down advantages and disadvantages of 'dynamic binding'.

Ans. Advantages

- (1) Dynamic binding facilitates more flexible and extensible software architectures, e.g.

- (a) Not all design decisions need to be known during the initial stages of system development, they may be postponed until run time.
 - (b) Complete source code is not required to extend the system, i.e. only headers and object code.
- (2) This aids both flexibility and extensibility:
- (a) **Flexibility:** Easily recombine existing components into new configurations.
 - (b) **Extensibility:** Easily adds new components.

Disadvantages

- Dynamic binding is less powerful than pointers to functions, but more comprehensible and less error prone, i.e. since the compiler performs type checking at compile time.
- There may be some additional time and space overhead from using dynamic binding.
- Dynamically bound objects are difficult to store in shared memory.
- It is less efficient than static binding.

Q24. Differentiate between function and operator overloading. Also give one example of each.

Ans. Function overloading is the practice of declaring the same function with different signatures. The same function name will be used with different number of parameters and parameters of different type. But overloading of functions with different return types is not allowed.

```
#include<iostream.h>
//prototype declaration for overloading function volume()
int volume(int);
double volume(double, int);
long volume(long, int, int);
```

```
void main()
{
cout<<volume(10) <<endl;
cout<<volume(2.5, 8) <<endl;
cout<<volume(100L, 75, 15) <<endl;
}
```

```
int volume(int s)                                //cube
{
    return(s*s*s);
}

double volume(double r,int h)                     //cylinder
{
    return(3.14519*r*r*h)
}

long volume(long l, int b, int h)//rectangular box
{
    return(l*b*h)
}
```

The output is:

1000
157.2595
112500

Operator Overloading

Operator overloading is extremely powerful and useful feature of Object Oriented programming.

In C++, you can overload any of the built-in operators, such as + or * to suit particular applications.

Example: Imagine that you're running a restaurant and you want to write a program to handle your billing, print your menus, and so on. We create a MenuItem class:

```
class MenuItem
{
private:
    float price;
    char name[ 40 ];

public:
    MenuItem::MenuItem(float itemPrice, char *itemName);
    float MenuItem::GetPrice(void);
};
```

Your program could define a MenuItem object for each item on the menu. When someone orders, you'd calculate the bill by adding together the price of each MenuItem like this:

```
MenuItem chicken( 8.99, "Chicken Gullybaba" );
MenuItem wine( 2.99, "Gully" );
float total;
```

```
total = chicken->GetPrice() + wine->GetPrice();
```

This particular diner had the chicken and a glass of wine. The total is calculated using the member function GetPrice().

Operator overloading provides an alternative way of totaling up the bill.

```
total = chicken + wine;
```

By adding the price of chicken to the price of wine.

Note: Currently, the compiler would complain if you tried to use a nonintegral type with the + operator.

In C++, you can “reprogram” this operation by giving the + operator a new meaning. To do this, we need to create a function to overload the + operator:

```
float operator+(MenuItem item1, MenuItem item2)
{
    return(item1.GetPrice() + item2.GetPrice());
}
```

Notice the name of this new function. Any function whose name follows the form:

```
operator<C++ operator>
```

Is said to overload the specified operator. When you overload an operator, you're asking the compiler to call your function instead of interpreting the operator as it normally would.

Calling an Operator Overloading Function

When the compiler calls an overloading function, it maps the operator's operands to the function's parameters. For example, suppose the function:

```
float operator+(MenuItem item1, MenuItem item2)
{
    return(item1.GetPrice() + item2.GetPrice());
}
```

Is used to overload the + operator. When the compiler encounters the expression chicken + wine it calls operator+(), passing chicken as the first parameter and wine as the second parameter. operator+()'s return value is used as the result of the expression.

It is important to note that:

The number of operands taken by an operator determines the number of parameters passed to its overloading function.

For example, a function designed to overload a unary operator takes a single parameter; a function designed to overload a binary operator takes two parameters.

Q25. What are public and private inheritance? Why do we need different access specifiers?

Ans. Inheritance is an important feature of classes; in fact, it is integral to the idea of object oriented programming. Inheritance allows you to create a hierarchy of classes, with various classes of more specific natures inheriting the general aspects of more generalised classes. In this way, it is possible to structure a program starting with abstract ideas that are then implemented by specific classes. For example, you might have a class Animal from which class dog and cat inherit the traits that are general to all animals; at the same time, each of those classes will have attributes specific to the animal dog or cat.

Public inheritance: The most open level of data hiding is public. Anything that is public is available to all derived classes of a base class, and the public variables and data for each object of both the base and derived class is accessible by code outside the class. Functions marked public are generally those the class uses to give information to and take information from the outside world; they are typically the interface with the class. The rest of the class should be hidden from the user using private or protected data (This hidden nature and the highly focused nature of classes is known collectively as encapsulation). The syntax for public is:

public:

Everything following is public until the end of the class or another data hiding keyword is used.

```
class base{  
private: int number;  
};  
class derived :public base{  
public: void f()  
{  
    ++number; //private base member not accessible  
}
```

```
}
```

```
};
```

The compiler error message is

'base::number' is not accessible in the function derived::f();

Here, only if the number is public then you can access it.

Private inheritance: Private is the highest level of data-hiding. Not only are the functions and variables marked private not accessible by code outside the specific object in which that data appears, but private variables and functions are not inherited (in the sense that the derived class cannot directly access these variables or functions). The level of data protection afforded by protected is generally more flexible than that of the private level. On the other hand, if you do not wish derived classes to access a method, declaring it private is sensible. The syntax for private is:

```
private:  
class base{  
public: int number;  
};  
class derived :private base{  
public: void f()  
{  
    ++number;           //public base member accessible  
}  
};  
  
class derived 2:private derived{  
public: void g()  
{  
    ++number;           //access to number is prohibited  
}  
};
```

The compiler error message is

'base::number' is not accessible in the function derived2::g();

Since public members of a base class are inherited as private in the derived class, the function derived::f() has no problem accessing it. However, when another class is derived from the class derived, this new class inherits number but cannot access it. If derived::g() were to call upon derived::f(), there is no problem since derived::f() is public and inherited into derived2 as private.

Ie. In derived2 we can write,

```
void g() {  
f();  
}
```

We need different access specifiers:

Any class can inherit from any other class, but it is not necessarily good practice to do so. Inheritance should be used when you have a more general class of objects that describes a set of objects. The features of every element of that set (of every object that is also of the more general type) should be reflected in the more general class. This class is called the base class. Base classes usually contain functions that all the classes inheriting from it, known as derived classes, will need. Base classes should also have all the variables that every derived class would otherwise contain.

Now look at an example of how to structure a program with several classes. Take a program used to simulate the interaction between types of organisms, trees, birds, bears, and other creatures inhabiting a forest. We have base classes for the animals and the plants. Then we want classes for specific types of animals: pigeons and vultures, bears and lions, and specific types of plants: oak and pine, grass and flower.

The classes share data. A derived class has access to most of the functions and variables of the base class. There are, however, ways to keep a derived class from accessing some attributes of its base class. The keywords public, protected, and private are used to control access to information within a class. It is important to remember that public, protected, and private control information both for specific instances of classes and for classes as general data types. Variables and functions designated public are both inheritable by derived classes and accessible to outside functions and code when they are elements of a specific instance of a class. Protected variables are not accessible by functions and code outside the class, but derived classes inherit these functions and variables as part of their own class. Private variables are neither accessible outside the class when it is a specific class nor are available to derived classes.

Q26. What is an interaction diagram? Explain the steps involved in drawing such a diagram, with the help of example.

Ans. Interaction Diagrams: Interaction diagrams model the behavior 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 behavior of several objects in a use case. They demonstrate how the objects collaborate for the behavior. Interaction diagrams do not give a in depth representation of the behavior.

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 demonstrate the behavior of objects in a use case by describing the objects and the messages they pass. The diagrams are read left to right and descending. The example below shows an object of class 1 start the behavior 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.

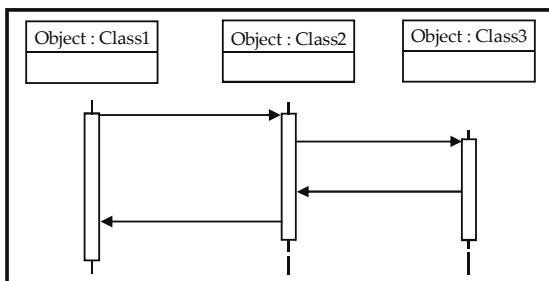


Fig. 9.4

Collaboration diagrams: 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 below shows, or for more detailed and complex diagrams a 1, 1.1, 1.2, 1.2.1... scheme can be used.

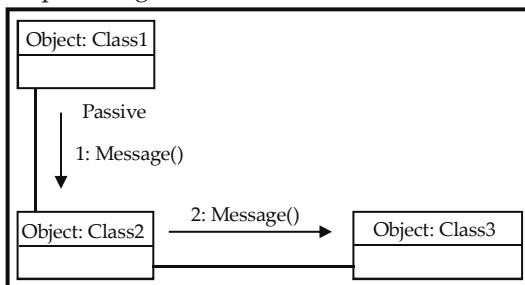


Fig. 9.5

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

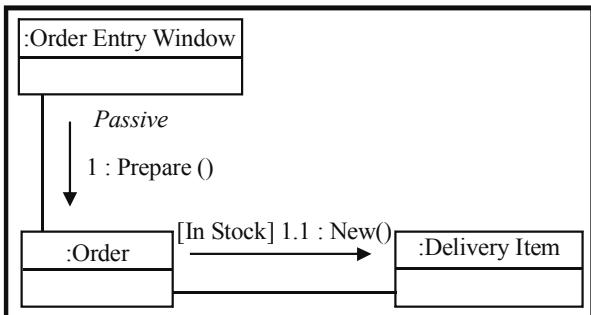


Fig. 9.6

Q27. Explain the concept of virtual functions. Give three examples of the use of such functions.

Ans. In object-oriented programming (OOP), a virtual function or virtual method is a function whose behavior, by virtue of being declared “virtual”, is determined by the definition of a function with the same signature furthest in the inheritance lineage of the instantiated object on which it is called. This concept is a very important part of the polymorphism portion of object-oriented programming (OOP).

The concept of the virtual function solves the following problem:

In OOP when a derived class inherits from a base class, an object of the derived class may be referred to (or cast) as either being the base class type or the derived class type. If there are base class functions overridden by the derived class, a problem then arises when a derived object has been cast as the base class type. When a derived object is referred to as being of the base’s type, the desired function call behavior is ambiguous.

The distinction between virtual and not virtual is provided to solve this issue. If the function in question is designated “virtual” then the derived class’s function would be called (if it exists). If it is not virtual, the base class’s function would be called.

For example, a base class **Animal** could have a virtual function **eat**. Subclass **Fish** would implement **eat()** differently than subclass **Wolf**, but you can invoke **eat()** on any class instance referred to as **Animal**, and get the **eat()** behavior of the specific subclass.

This allows a programmer to process a list of objects of class **Animal**, telling each in turn to eat (by calling **eat()**), with no knowledge of what

kind of animal may be in the list. You also do not need to have knowledge of how each animal eats, or what the complete set of possible animal types might be.

The following is an example in C++:

```
#include <iostream>
class Animal
{
public:
    virtual void eat() { std::cout << "I eat like a generic Animal.\n"; }
};

class Wolf : public Animal
{
public:
    void eat() { std::cout << "I eat like a wolf!\n"; }
};

class Fish : public Animal
{
public:
    void eat() { std::cout << "I eat like a fish!\n"; }
};

class OtherAnimal : public Animal
{
};

int main()
{
    Animal *anAnimal[4];
    anAnimal[0] = new Animal();
    anAnimal[1] = new Wolf();
    anAnimal[2] = new Fish();
    anAnimal[3] = new OtherAnimal();

    for(int i = 0; i < 4; i++)
```

```
anAnimal[i]->eat();
```

```
}
```

Output with the virtual method eat:

I eat like a generic Animal.

I eat like a wolf!

I eat like a fish!

I eat like a generic Animal.

Output without the virtual method eat:

I eat like a generic Animal.

I eat like a generic Animal.

I eat like a generic Animal.

```
class Window // Base class for C++ virtual function example
```

```
{
```

```
public:
```

```
virtual void Create() // virtual function for C++ virtual function example
```

```
{
```

```
cout << "Base class Window" << endl;
```

```
}
```

```
};
```

```
class CommandButton : public Window
```

```
{
```

```
public:
```

```
void Create()
```

```
{
```

```
cout << "Derived class Command Button " << endl;
```

```
// Overridden C++ virtual  
// function
```

```
}
```

```
};
```

```
void main()
```

```
{
```

```
Window *x, *y;
```

```
x = new Window();
x->Create();

y = new CommandButton();
y->Create();
}
```

The output of the above program will be,

Base class Window

Derived class Command Button

Here when we remove virtual.

The output of the above program will be,

Base class Window

Base class Window

Q28. Explain any five advantages of object oriented programming.

Ans. The following are the basic advantages of object-oriented programming:

- **Modular Design:** The software built around OOP are modular, because they are built on objects and we know objects are entities in themselves, whose internal working is hidden from other objects and is decoupled from the rest of the program.
- **Simple approach:** the objects, model real world entities, which results in simple program structure.
- **Modifiable:** Because of its inherent properties of data abstraction and encapsulation, the internal working of objects is hidden from other objects thus, any modification made to them should not affect the rest of the system.
- **Extensible:** The extension to the existing program for its adaptation to new environment can be done by simply adding few new objects or by adding new features in old classes/ types.
- **Reusable:** Objects once made can be reused in more than one program.

Q29. Explain any three advantages of inheritance.

Ans. 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.

- **Less Labour intensive:** The programmer does not have to rewrite the same long similar programs just because the applications to be developed has slightly different requirements.
- **Well Organised:** The objects are well organised in a way that they follow some hierarchy.

Q30. Explain the purpose of the following with the suitable example of each:

(i) Use Case Diagram

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

Actors: are users of system, including human beings and other system components.

Use Cases: includes services provided to users of the system.

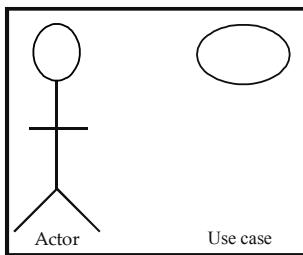


Fig. 9.7

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.

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 contains more might become visible.

Example: A user placing an order with a sales company might follow these steps:

- Browse catalogue and select items
- Call sales representation
- Supply shipping information

- Supply payment information
- Receive confirmation number from salesperson

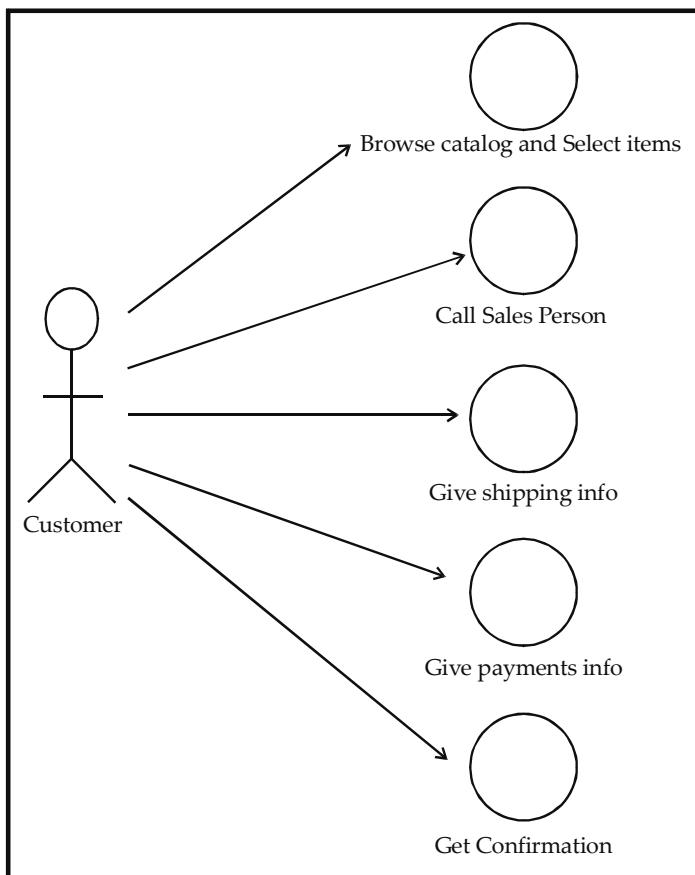


Fig. 9.8

(ii) Interaction Diagram

Ans. Interaction diagram 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.

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 behaviours. They do not give a in-depth representation of the behaviour. Sequence diagrams generally show the sequence of events that occur. Collaboration diagrams demonstrate have objects are statically connected.

The following example shows a simple collaboration diagram for placing an order.

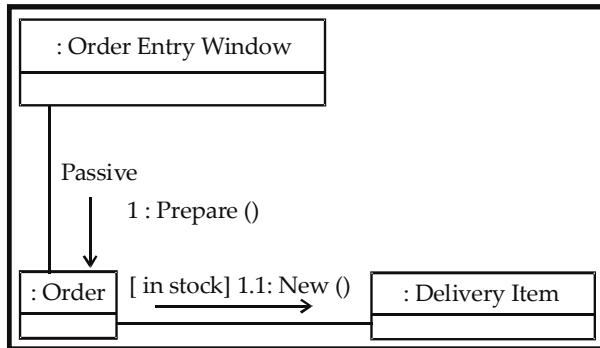


Fig. 9.9

Q31. List the various restrictions with operator overloading. Write a C++ program to illustrate the overloading of Increment operator.

Ans. In operator overloading, 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 operator functions namely operator=, operator [], operator () and operator ? Must be non-static member functions. Due to this, their first operands will be values. An operator function should be either a member or take at least one class object arguments. 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.

Overloading of increment operator

Since ++ can use as postfix as well a prefix operator, we can have two different overloaded functions.

The following is a 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 ++ U  
{  
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 ++;  
}
```

Q32. Write a C++ program to print reverse of a 5-digit number.

Ans. #include <iostream.h>

```
void main ()  
{  
int num, rev = 0, rem, temp;  
cout << "Enter any number";  
cin >> num.;  
temp = num;  
while (temp > 0)  
{ rem = temp % 10;  
temp = temp / 10;  
rev = rev * 10 + rem; }  
cout << "Reverse of" << num << "is" << rev; }
```

Q33. What are various visibility modes in C++? Explain each of them with suitable examples.

Ans. Various visibility modes in C++ are: Private, Public and Protected.

Public Mode: In this mode all members in C++ class are public members that can be accessed by any function, whether member function of a class or non-member function of a class.

Example:

```
class X  
{  
public:  
int a;  
int sqr(int a)  
{  
return a * a;  
}  
};  
X obj1;  
void main()  
{
```

```
int b;  
obj1.a = 10;           //valid as public member  
b = obj1.sqr(15);    //valid as public member  
}
```

Private Mode: In private mode all the class member are private members and are hidden from outside world. The private members implement the OOP concept of data hiding. The private members of a class can be used only by member functions and friend functions of the class in which it is declared.

Protected Mode: In this mode all members are protected members that can be used only by the member functions and friends of the class in which it is declared. The protected members are similar to private members that cannot be accessed by non-member functions. The difference between protected and private members is that private members can be inheritable but private members are not inheritable.

Q34. Write a C++ program to accept a 5-digit number and report whether it is divisible by 3, 5, 7, 9 or not.

Ans. #include <iostream.h>

```
#include <conio.h>  
void main()  
{  
    int num;  
    cout<<"Enter a five digit number";  
    cin>>num;  
    if(num % 3 == 0)  
        cout<<num<<"is divisible by 5";  
    if(num % 7 == 0)  
        cout<<num<<"is divisible by 7";  
    if(num % 9 == 0)  
        cout<<num<<"is divisible by 9; }
```

Q35. Define modular programming. Also write it's characteristics.

Ans. Modular Programming: The technique of hierarchical decomposition of a procedural program, to specify the various modules or tasks that are to be completed in an order to solve a problem, is known as Modular Programming.

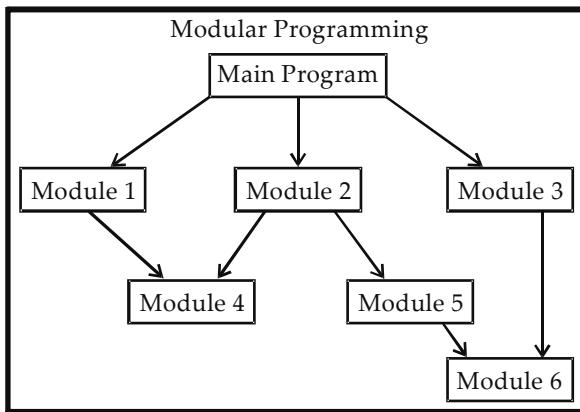


Fig. 9.10: Typical Structure of Modular Programming

Main Characteristics of Modular Programming are:

- Emphasis is given on algorithms.
- Large programs are divided into smaller modules.
- Most of the modules share global data.
- Modules transform data from one form to another.
- Employs a top-down approach in program design.
- Data move openly around the system from one form to another.

Q36. How are objects created and initialised? Explain with the help of an example.

Ans. Object Creation and Initialisation in C++: A class description will not reserve any space in the memory. It provides only a template or a blue print to create data objects. To create actual data, objects are created. Object reserves memory for individual data members.

Values can be assigned to individual data members using objects. Objects are created in C++ with following syntax:

```
classname objectname; //object declaration and creation//
```

Here class becomes the datatype for object. A class has same relationship with an object as that of a datatype to a variable.

The member functions of the class are invoked using object as:

```
objectname.member Function( );
```

Two scopes of declaration of objects in C++ are:

(1) Global Object: If objects are declared outside all function bodies in a class. This object is globally available to all functions in the program, i.e. object can be used anywhere in the class.

Example:

```
class X
{
public:
int a;
void fun(void);
};

X obj1;
int main( )
{
    obj1.a = 10;           //creation and initialisation of object//
    obj1.fun();            //object obj1 is global//

.....
}

void fun1(void)
{
    obj1.a = 20;           //object obj1 is global//
    obj1.fun();
.....
}
```

(2) Local Object: An object is said to be a local object if it is declared within a function, which means that this object is locally available to the function that declares it and it cannot be used outside the function declaration.

Q37. How are arrays declared in C++? How can pointers manipulate array elements? Give an example.

Ans. Array Declaration in C++: Array is a named list for a finite number n of similar data elements. Each of the data elements can be referenced respectively by a set of consecutive numbers, usually 0, 1, 2, 3 ... n.

Array declaration can be done in C++ as: datatype arrayname[size];

Example: int arr[10];

Here arrayname is arr and its size is 10 elements each of type int.

Array can be one dimensional, two dimensional or multi dimensional.

Array elements can be manipulated using pointers.

C++ treats the name of an array as a pointer, i.e. the base memory address of array elements, i.e. C++ interprets array name as the address of its first element.

Example: A C++ program showing relations between array and pointers, i.e. How array elements can be accessed and manipulated using pointers.

```
#include <iostream.h>
void main(void)
{
int num[10] = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};
cout << "Value of num =" << num << endl;
cout << "Value of num[0] =" << num[0] << endl;
cout << "Value of &num[0] =" << &num[0] << endl;
cout << "Value of num[5] =" << num[5] << endl;
cout << "Value of *(num + 5) =" << *(num + 5) << endl;
int *ptr;
ptr = num;
cout << "Value of *(ptr + 3) =" << *(ptr + 3) << endl;
cout << "Value of ptr[3] =" << ptr[3] << endl;
cout << "Value of num[3] =" << num[3];
cout << "\n Value After Increment" << endl;
ptr = ptr + 1;
cout << "Value of *(ptr + 3) =" << *(ptr + 3);
cout << "Value of ptr[3] =" << ptr[3];
cout << "Value of num[3] =" << num[3];
```

Q38. Design a class complex representing complex numbers. Overload '<<' operator for this class complex.

Ans. Class A

```
{ 
public:
int x;
int y;
}
A operator+ (const A& x, const A& y)
{
A retObj;
retObj.y = x.y + y.y;
return retObj;
}
```

```
ostream& operator<<(ostream& out, A& x )
{
out << x.x << " " << x.y << endl;
return out;
}

int main()
{
A a,b,c;
a.x = 5;
a.y = 10;
b.x = 4;
b.y = 15;
c=a+b;
cout << c;
cout << a+b;
}
```

Q39. Write short notes on Command Line arguments.

Ans. The command line arguments are typed by the user and are delimited by the user and are delimited by a space.

The first argument is always the filename and contains the program to be executed.

The main() functions which we have been using up to now without any arguments can take two arguments main(int argc, char*argv[]).

The first argument argc represents the number of arguments in the command line. The second argument argv is an array of char type pointers that points to the command line arguments. The size of this array will be equal to the value of argc.

Q40. How are dynamic objects created and destroyed? Explain it with the help of an example.

Ans. Objects are dynamically created (allocated) and deleted (destroyed or deallocated) using operators *new* and *delete*. These operators are intended as replacements for the C storage allocation functions such as *malloc* and the storage deallocation function *free*, which are also available in C++. There are several advantages to using operators *new* and *delete* instead of using the functions *malloc* and *free*. For example:

- *new* does not require specification of the object size; it computes that automatically from the specified type.

- *new* returns a pointer of the correct type. The pointer returned by *malloc* must typically be coerced to the desired pointer type. In ANSI C, the pointer returned by *malloc* is a *void* pointer which is automatically coerced to the desired pointer type.
- operators *new* and *delete*, respectively, invoke the appropriate constructor and the destructor to initialise the newly created object and to cleanup prior to deleting the object.

Operators *new* and *delete* call the global operator functions *new* and *delete*, or if defined class-specific operator functions with the same names to allocate and deallocate objects. Most programmers usually do not write their own customised storage allocation and deallocation operator functions (which are invoked by the operators *new* and *delete*) because they find the global functions to be adequate. Occasionally, programmer may have to write such functions to meet system design requirements (e.g. execution speed and storage allocation from a specific memory pool).

Q41. Explain the working of any one bitwise operator giving suitable examples.

Ans. Left shift and right shift operator: The bitwise left shift (`<<`) shifts operator bits to the left. For example, consider the number 3, which is binary 0011. Left shifting by 1 (`3 << 1`) changes 0011 to 0110, which is decimal 6. Note how each bit moved 1 place to the left. Left shifting by 2 (`3 << 2`) changes 0011 to 1100, which is decimal 12. Left shifting by 3 (`3 << 3`) changes 0011 to 1000. Note that we shifted a bit off the end of the number! Bits that are shifted off the end of the binary number are lost.

The bitwise right shift (`>>`) operator shifts bits to the right. Right shifting by 1 (`3 >> 1`) changes 0011 to 0001, or decimal 1. The rightmost bit shifted off the end and was lost!

Although our examples above are shifting literals, you can shift variables as well:

[view source](#)

print?

- `unsigned int nValue = 4;`
- `nValue = nValue << 1; // nValue will be 8`

Rule: When dealing with bit operators, use `unsigned` variables.

Programs today typically do not make much use of the bitwise left and right shift operator in this capacity. Rather, you tend to see the bitwise left shift operator used with `cout` in a way that doesn't involve shifting bits at all! If `<<` is a bitwise left shift operator, then how does `cout << "Hello, world!"`; print to the screen? The answer is that `cout` has overridden (replaced) the default meaning of the `<<` operator and given it a new meaning.

Q42. Consider the following class hierarchy.

Create the class hierarchy using C++, having at least one constructor for each class. Assuming that all parallelograms are either rectangles or rhombuses, write a polymorphic function to calculate the areas of the figures.

Ans. Class parallelogram

```
{  
int length;  
public:  
Parallelogram() {length = 0;}  
long area() = 0;  
};  
Class Rectangle : public parallelogram  
{  
int width;  
Public;  
Rectangle () {length = 0; width = 0;}  
long area ()  
{  
return length*width;  
}  
void input () { cin >> length >> width; }  
};  
Class Rhombus : public parallelogram  
{  
int height;  
public:  
Rhombus ()  
{  
length = 0; height = 0;  
}  
void input () {  
cin >> length >> height;  
}  
long area ()
```

```
{  
return length*height;  
}  
};
```

Q43. Write a menu-driven program in C++ to add, modify and search a student record in a file.

Ans. #include <iostream.h>

```
#include <string.h>  
class student  
{  
int rollno;  
char name [25]  
public:  
void input ()  
{  
cout << "\n Enter roll no.";  
cin >> roll no;  
cin << "\n enter name:";  
cin >> name;  
}  
int getroll no () { return rollno.  
void setname (char, *n) {strcpy (name, n);}  
void display"()  
{  
cout << "\n Roll no: " << roll no << " Name:"  
<< name ;  
}  
};  
int main ()  
{  
student s;  
char n [25];  
int ch=0; rollno;  
fstream file ("student. dat, ios:: in | ios:: app);  
do {
```

```
cout << "\n 1. Add";
cout << "\n 2. Modify:";
cout << "\n 3. Search";
cout << "\n 4. Exit";
cout << "\n Enter your choice:";
cin >> ch;
switch (ch)
{ case 1: s.input ();
file.write ((char) & s, size of (s));
break;
case 2: cout << "\n enter roll no:";
cin >> roll no:
file. seekg(0);
do { file.read ((char)&s, size of (s));
if (file.oof ()) {cout << "Not found"; break;}
else if (s.getroll no () == roll no)
{ s.input ();
file.write ((char *)& s, size of (s));
break;
}
}
break;
case 3: cout << "\n Enter roll no:";
cout >> roll no;
file.seek g(0);
do{file.read ((char*)s, size of (s));
if (file. Of (1) { cout << "not found"; break)
else if (s. get roll no () == roll no.)
{ s.display (); break ;}
:
case 4: break;
otherwise: cout ("(" in invalid choice, try again");
}while (ch; = 4);
return 0;
}
```

Q44. What are the major benefits of Object Oriented Programming?

Ans. The following are some of the major benefits of OOP:

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

Q45. Explain the meaning of polymorphism. Describe how polymorphism is accomplished in C++ taking a suitable example?

Ans. Polymorphism is the ability to use an operator or method in different ways. Polymorphism gives different meanings or functions to the operators or methods. Poly, referring to many, signifies the many uses of these operators and methods. A single method usage or an operator functioning in many ways can be called polymorphism. Polymorphism

refers to codes, operations or objects that behave differently in different contexts.

Below is a simple example of the above concept of polymorphism: $6 + 10$

The above refers to integer addition.

The same `+` operator can be used with different meanings with strings: "Amanta" + "Kumar"

The same `+` operator can also be used for floating point addition: $7.15 + 3.78$

Polymorphism is a powerful feature of the object oriented programming language C++. A single operator `+` behaves differently in different contexts such as integer, float or strings referring the concept of polymorphism. The above concept leads to operator overloading. The concept of overloading is also a branch of polymorphism. When the exiting operator or function operates on new data type it is overloaded. This feature of polymorphism leads to the concept of virtual methods.

Polymorphism refers to the ability to call different functions by using only one type of function call. Suppose a programmer wants to code vehicles of different shapes such as circles, squares, rectangles, etc. One way to define each of these classes is to have a member function for each that makes vehicles of each shape. Another convenient approach the programmer can take is to define a base class named `Shape` and then create an instance of that class. The programmer can have array that hold pointers to all different objects of the vehicle followed by a simple loop structure to make the vehicle, as per the shape desired, by inserting pointers into the defined array. This approach leads to different functions executed by the same function call. Polymorphism is used to give different meanings to the same concept. This is the basis for Virtual function implementation.

In polymorphism, a single function or an operator functioning in many ways depends upon the usage to function properly. In order for this to occur, the following conditions must apply:

- All different classes must be derived from a single base class. In the above example, the shapes of vehicles (circle, triangle, ractangle) are from the single base class called `Shape`.
- The member function must be declared virtual in the base class. In the above example, the member function for making the vehicle should be made as virtual to the base class.

Q46. Why to Overload Operators? What are the General rules for operator overloading in C++?

Ans. Most fundamental data types have pre-defined operators associated with them. For example, the C++ data type `float`, together with the operators

`+, -, *, and /,` provides an implementation of the mathematical concepts of an integer. This is purely a convenience to the user of a class. Operator overloading isn't strictly necessary unless other classes or functions expect operators to be defined.

To make a user-defined data type as natural as a fundamental data type, the user-defined data type must be associated with the appropriate set of operators. Operators are defined as either member functions or friend functions. The purpose of operator overloading is to make programs clearer by using conventional meanings for `==`, `[]`, `+`, etc. Operators can be overloaded in any way from those available like globally or on the basis of class by class. While implementing the operator overloading this can be achieved by implementing them as functions. Whether it improves program readability or causes confusion depends on how well you use it. In any case, C++ programmers are expected to be able to use it.

The user can understand the operator notation more easily as compared to a function call because it is closer to the real-life implementation. Thus, by associating a set of meaningful operators, manipulation of an ADT can be done in a conventional and simpler form. Associating operators with an ADT involves overloading them. Although the semantics of any operator can be extend, we can't change its syntax associativity, precedence, etc.

The following rules constrain how overload operators are implemented:

- You cannot define new operators, such as^{**}.
- You cannot redefine the meaning of operators when applied to built-in data types.
- Overloaded operators must either be a non static class member function or a global function.
- Obey the precedence, grouping, and number of operands dictated by their typical use with built-in types. Therefore, there is no way to express the concept "add 2 and 3 to an object of type Point," expecting 2 to be added to the x coordinate and 3 to be added to the y coordinate.
- Unary operators declared as member functions take no arguments; if declared as global functions, they take one argument.
- Binary operators declared as member functions take one argument; if declared as global functions, they take two arguments.
- If an operator can be used as either a unary or a binary operator (`&`, `*`, `+`, and `-`), you can overload each use separately.

- Overloaded operators cannot have default arguments.
- All overloaded operators except assignment (operator=) are inherited by derived classes.
- The first argument for member-function overloaded operators is always of the class type of the object for which the operator is invoked (the class in which the operator is declared, or a class derived from that class). No conversions are supplied for the first argument.
- Overloading + doesn't overload +=, and similarly for the other extended assignment operators.
- You may not redefine ::, sizeof, ?:, or . (dot).
- =, [], and -> must be member functions if they are overloaded.
- ++ and – need special treatment because they are prefix and postfix operators.
- There are special issues with overloading assignment (=). Assignment should always be overloaded if an object dynamically allocates memory.

Q47. Differentiate between object-oriented and object-based programming languages.

Ans. Difference between object-oriented and object-based programming languages are as follows:

Table 9.2

	Object Based Language	Object Oriented Language
Support of features	Object Based Language does not support all the features of OOPs.	Object Oriented Language supports all the features of OOPs.
Inheritance	Object Based Language does Not Support OOPs feature i.e. Inheritance.	Object Oriented Language supports all the Features of OOPs including Inheritance.
Sample	Visual Basic is an Object based Programming Language because you can use class and Object here but cannot inherit one class from another class i.e. it does not support Inheritance.	Java is an Object Oriented Languages because it supports all the concepts of OOPs like Data Encapsulation, Polymorphism Inheritance, Data Abstraction, Dynamic Binding etc.
Example	Javascript, VB are example of Object Based Language.	C#, Java, VB. Net are example of Object Oriented Languages.

Q48. How is the working of member function different from friend function and a non member function?

Ans. A friend function is used for accessing the non-public members of a class. A class can allow non-member functions and other classes to access its own private data, by making them friends. Thus, a friend function is an ordinary function or a member of another class. The friend function is written as any other normal function, except the function declaration of these functions is preceded with the keyword friend. The friend function must have the class to which it is declared as friend passed to it in argument.

Some important points for using friend functions in C++:

- The keyword friend is placed only in the function declaration of the friend function and not in the function definition.
- It is possible to declare a function as friend in any number of classes.
- When a class is declared as a friend, the friend class has access to the private data of the class that made this a friend.
- A friend function, even though it is not a member function, would have the rights to access the private members of the class.
- It is possible to declare the friend function as either private or public.
- The function can be invoked without the use of an object. The friend function has its argument as objects, seen in example below.

Example:

```
#include <iostream>
using namespace std;
class Amanta
{
private:
int a,b;
public:
void test()
{
a=100;
b=200;
}
friend int compute(Amanta e1);
};
```

```
int compute(Amanta e1)
{
    return int(e1.a+e1.b)-5;
}
void main()
{
    Amanta e;
    e.test();
    cout << "The result is:" << compute(e);
    //Calling of Friend Function with object as argument.
}
```

Q49. Write down the Characteristics of Pure Virtual Functions.

Ans. Following are the characteristics of Pure Virtual Functions:

- A class member function can be declared to be pure virtual by just specifying the keyword ‘virtual’ in front and putting ‘=0’ at the end of the function declaration.
- Pure virtual function itself do nothing but acts as a prototype in the base class and gives the responsibility to a derived class to define this function.
- As pure virtual functions are not defined in the base class thus a base class can not have its direct instances or objects that means a class with pure virtual function acts as an abstract class that cannot be instantiated but its concrete derived classes can be.
- We cannot have objects of the class having pure virtual function but we can have pointers to it that can in turn hold the reference of its concrete derived classes.
- Pure virtual functions also implements run time polymorphism as the normal virtual functions do as binding of functions to the appropriate objects here is also delayed up to the run time, that means which function is to invoke is decided at the run time.
- Pure virtual functions are meant to be overridden.
- Only the functions that are members of some class can be declared as pure virtual that means we cannot declare regular functions or friend functions as pure virtual.
- The corresponding functions in the derived class must agree be compatible with the pure virtual function’s name and signature that means both must have same name and signature.

- For abstract class, pure virtual function is must.
- The pure virtual functions in an abstract base class are never implemented. Because no objects of that type are ever created, there is no reason to provide implementations, and the ADT (Abstract Data Type) works purely as the definition of an interface to objects which derive from it.
- It is possible, however, to provide an implementation to a pure virtual function. The function can then be called by objects derived from the ADT, perhaps to provide common functionality to all the overridden functions.

Q50. What you learn from C++ character Set, Tokens, Identifiers, Keywords?

Or

What do you mean by identifier in C++?

Or

What do you mean by keyword in C++?

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

Table 9.3

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

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

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

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

- Keywords
- Identifiers
- Literals

- Punctuators
- Operators

Identifiers

A symbolic name is generally known as an identifier. Valid identifiers are a sequence of one or more letters, digits or underscore characters (_). Neither spaces nor punctuation marks or symbols can be part of an identifier. Only letters, digits and single underscore characters are valid.

In addition, variable identifiers always have to begin with a letter. In no case can they begin with a digit. Another rule for declaring identifiers is that they cannot match any keywords of the C++ programming language. The rules for the formation of identifiers can be summarised as:

An identifier may include of alphabets, digits and/or underscores. It must not start with a digit.

C++ is case sensitive, i.e. upper case and lower case letters are considered different from each other. It may be noted that TOTAL and total are two different names. It should not be a reserved word.

A member function with the same name as its class is called constructor and it is used to initialise the objects of that class type with an initial value. Objects generally need to initialise variables or assign dynamic memory during their process of creation to become operative and to avoid returning unexpected value during their execution. For example, to avoid unexpected results in the example given below we have initialised the value of rollno as 0 and marks as 0.0.

Keywords

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

List of Keywords:

asm	double	new	switch
auto	else	operator	template
break	enum	private	this
case	extern	protected	try
catch	float	public	typedef
char	for	register	union
class	friend	return	unsigned
const	goto	short	virtual
continue	if	signed	void

default	inline	sizeof	volatile
delete	int	static	while
do	long	struct	

Q51. What is the difference between variable and constant in C++ programming language?

Ans. A constant remains fixed with any kind of change in variation or the quantity whereas variables vary with certain quantities. A variable is allocation in a memory, which holds a piece of data; it can be represented by anything apart from numbers whereas a constant can be any numbers or characters.

If we talk in reference with C++ Language:

- Variable is the named memory location whose value or we can say whose data can be changed during execution of program whereas Constant is the named memory location whose value can't be changed or whose value is fixed during the execution of the program.
- Variable's syntax is: data_type variable_name; and Constant's syntax is: const data_type constant_name=value.
- Variable's example: int a; Constant's example is: const int a=28.
- Variables can be initialised after its declaration whereas constants must be initialised at the time of their declaration otherwise they will take a garbage/junk value.

Q52. Illustrate a Simple C++ Program.

Ans. The best way to start learning a programming language is by writing a program. A simple C++ program has four sections and these are shown in following C++ program.

Simple C++ Program:

```
#include <iostream.h>           //Section: 1 – The include Directive
using namespace std;           //Section: 2 – Class declaration and
                               //member functions
int main ()                   //Section: 3 – Main function definition
{
    cout << "Hello World!";    //Section: 4 – Declaration of an object
    return 0;
}
```

Output:

Hello World!

This is one of the simplest programs that can be written in C++ programming language. It contains all the fundamental components which every C++ program can have. Line by line explanation of the codes of this program and its sections is given below:

Section: 1 – The include Directive

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

Lines beginning with a hash sign (#) are directives for the pre-processor. They are not regular code lines with expressions but indications for the compiler's pre-processor. In this case the directive #include <iostream> tells the pre-processor to include the iostream standard file. This specific file (iostream) includes the declaration of the basic standard input-output library in C++, and it is included because its functionality is going to be used later in the program.

Section: 2 – Class declaration and member functions

```
using namespace std;
```

All the elements of the standard ANSI C++ library are declared within namespace std;. The syntax of this command is: using namespace std;. In order to access its functionality we declare all the entities inside namespace std;. This line is very often used in C++ programs that use the standard library and defines a scope for the identifiers that are used in a program.

Section: 3 – Main function definition

```
int main ()
```

This line corresponds to the beginning of the definition of the main function. The main function is the point by where all C++ programs start their execution, independently of its location within the source code. It does not matter whether there are other functions with other names defined before or after it – the instructions contained within this function's definition will always be the first ones to be executed in any C++ program. For that same reason, it is essential that all C++ programs have a main function.

The word main is followed in the code by a pair of parentheses (()). That is because it is a function declaration: In C++, what differentiates a function declaration from other types of expressions is these parentheses that follow its name. Optionally, these parentheses may enclose a list of parameters within them.

Section: 4 – Declaration of an object

Right after these parentheses we can find the body of the main function enclosed in braces ({}). What is contained within these braces is what the function does when it is executed.

```
cout << "Hello World!" ;
```

This line is a C++ statement. A statement is a simple or compound expression that can actually produce some effect. In fact, this statement is used to display output on the screen of the computer. cout is the name of the standard output stream in C++, and the meaning of the entire statement is to insert a sequence of characters.

Cout is declared in the iostream standard file within the std namespace, so that's why we needed to include that specific file and to declare that we were going to use this specific namespace earlier in our code.

Notice that the statement ends with a semicolon character (;). This character is used to mark the end of the statement and in fact it must be included at the end of all expression statements in all C++ programs.

```
return 0;
```

The return statement causes the main function to finish.

Q53. Briefly explain Data types in C++ programming.

Or

Write the different types of data types.

Ans. In C++ programming, we store the variable in our computer's memory, but the computer has to know what kind of data we want to store in them. The amount of memory required to store a single number is not the same as required by a single letter or a large number. Further, interpretation of different data is different inside computers memory.

The memory in computer system is organised in bits and bytes. A byte is the minimum amount of memory that we can manage in C++. A byte can store a relatively small amount of data: one single character or a small integer. In addition, the computer can manipulate more complex data types that come from grouping several bytes, such as long numbers or non-integer numbers.

Data Type in C++ is used to define the type of data that identifiers accepts in programming and operators are used to perform a special task such as addition, multiplication, subtraction, and division, etc of two or more operands during programming.

C++ supports a large number of data types. The built in or basic data types supported by C++ are integer, floating point and character type. A brief discussion on these types is shown in fig. 9.11.

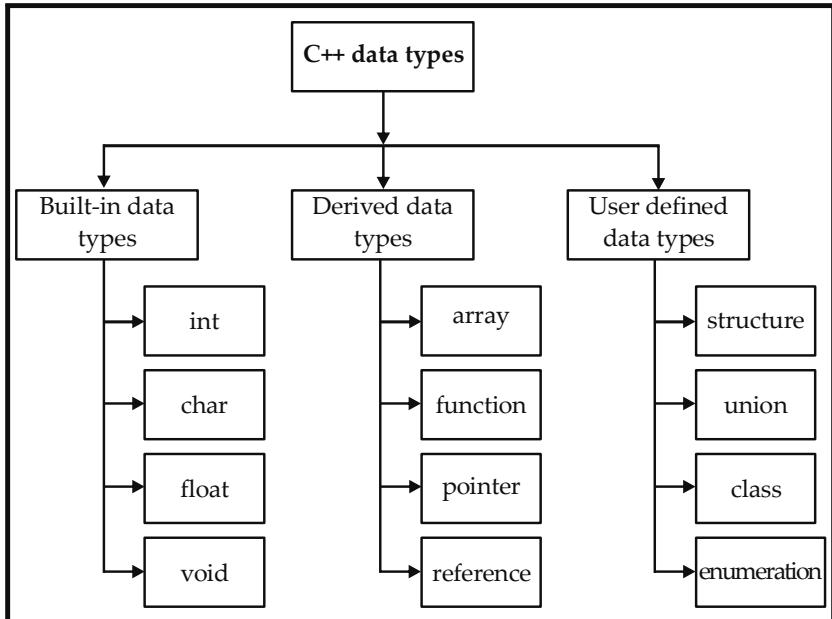


Fig: 9.11

(1) **Built-in Data Types:** There are four types of built-in data types as shown in the fig. 9.11. Now discuss each of these and the range of values accepted by them one by one.

(a) **Integer Data type (int):** An integer is an integral whole number without a decimal point. These numbers are used for counting. For example 26, 373, -1729 are valid integers. Normally an integer can hold numbers from -32768 to 32767.

The int data type can be further categorised into following:

- (i) Short
- (ii) Long
- (iii) Unsigned

The short int data type is used to store integer with a range of - 32768 to 32767, However, if the need be, a long integer (long int) can also be sued to hold integers from -2, 147, 483, 648 to 2, 147, 483, 648 to 2, 147, 483, 648. The unsigned int can have only positive integers and its range lies up to 65536.

(b) **Floating point data type (float):** A floating point number has a decimal point. Even if it has an integral value, it must include a decimal point at the end. These numbers are used for

measuring quantities. Examples of valid floating point numbers are: 27.4, -92.7, and 40.03.

A float type data can be used to hold numbers from 3.4×10^{-38} to $3.4 \times 10^{+38}$ with six or seven digits of precision. However, for more precision a double precision type (double) can be used to hold numbers from 1.7×10^{-308} to $1.7 \times 10^{+308}$ with about 15 digits of precision. Summary of Basic fundamental data types as well as the range of values accepted by each data type is shown in the table 9.4.

(c) Void data type: It is used for following purposes:

- (i) It specifies the return type of a function when the function is not returning any value.
- (ii) It indicates an empty parameter list on a function when no arguments are passed.
- (iii) A void pointer can be assigned a pointer value of any basic data type.

(d) Char data type: It is used to store character values in the identifier. Its size and range of values is given in Table 9.4.

Table 9.4: Basic Fundamental Data Type

Name	Description	Size*	Range
Char	Character or small integer.	1 byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer.	2 bytes	signed: -32768 to 32767 unsigned: 0 to 65535
Int	Integer.	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer.	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
Bool	Boolean value. It can take one of two values: true or false.	1 byte	True or false
Float	Floating point number.	4 bytes	$+/-3.4e +/- 38$ (~7 digits)
Double	Double precision floating point number.	8 bytes	$+/-1.7e +/- 308$ (~15 digits)
long double	Long double precision floating point number.	8 bytes	$+/- 1.7e +/- 308$ (~15 digits)
Wchar_t	Wide character.	2 or 4 bytes	1 wide character

Note: The values of the column Size and Range given in the table above, depends on the computer system on which the program is compiled. The values shown above are those found on 32-bit computer systems. But for other systems, the general specification is that int has the natural size suggested by the system architecture (one “word”) and the four integer type’s char, short, int and long must each one be at least as large as the one preceding it, with char being always one byte in size. The same applies to the floating point types float, double and long double, where each one must provide at least as much precision as the preceding one.

- (2) **Derived Data types:** C++ also permits four types of derived data types. As the name suggests, derived data types are basically derived from the built-in data types. There are four derived data types. These are:
- (i) Array,
 - (ii) Function,
 - (iii) Pointer, and
 - (iv) Reference.
- (3) **User Defined Data Types:** C++ also permits four types of user defined data types. As the name suggests, user defined data types are defined by the programmers during the coding of software development. There are four user defined data types. These are:
- (i) Structure,
 - (ii) Union,
 - (iii) Class, and
 - (iv) Enumerator.

Q54. What do you mean by I/O formatting in C++ programming language?

Ans. C++ supports input/output statements, which can be used to feed new data into the computer or obtain output on an output device such as: VDU, printer etc. It provides both formatted and unformatted stream I/O statements. The following C++ streams can be used for the input/output purpose.

Input/Output

Output is accomplished by using cout, which opens a “stream” to the standard output device (the screen). Data is inserted into the output stream using the << (insertion) operator. Input is accomplished by using cin, which opens a “stream” from the standard input device (keyboard). Data is retrieved from the stream by using the >> (extraction) operator.

Note: Every cin should be prefaced by a cout to prompt the user.

- **Include <iostream.h>:** The lines in the above program that start with symbol '#' are called directives and are instructions to the

compiler. The word include with '#' tells the compiler to include the file iostream.h into the file of the above program. File iostream.h is a header file needed for input/output requirements of the program. Therefore, this file has been included at the top of the program.

- **void main ():** The word main is a function name. The brackets () with main tells that main () is a function. The word void before main () indicates that no value is being returned by the function main (). Every C++ program consists of one or more functions. However, when program is loaded in the memory, the control is handed over to function main () and it is the first function to be executed.
- **The curly brackets and body of the function main ():** Each C++ program starts with function called main (). The body of the function is enclosed between curly braces. These braces are equivalent to Pascal's BEGIN and END keywords. The program statements are written within the brackets. Each statement must end by a semicolon, without which an error message is generated.

I/O Formatting Functions

- **Comments in C++:** A comment is a statement in the program body to enhance the reading and understanding of the program. Comments are included in a program to make it more readable. If a comment is short and can be accommodated in a single line, then it is started with double slash sequence in the first line of the program. The syntax of short and one line comment is:

```
//Comment line...
```

However, if there are multiple lines in a comment, it is enclosed between the two symbols /* and */

Everything between /* and */ is ignored by the complier. The syntax of multiple line comment is

```
/* Start of multiple line comment
```

```
.....
```

```
.....
```

```
.....End of multiple line comment */
```

- **Unformatted Console I/O Functions:** The I/O functions such as getch(), putchar(), get(), and put() etc. are called unformatted console I/O functions. The header file for these functions is <stdio.h> and should be included in the beginning of the program. The meaning and use of these functions can be illustrated as follows:

getch() This function is used to accept the input character which is typed by the keyboard during the execution of C++ program.

putchar() It displays the character on the screen at the current location of cursor.

Get (), and put () are the string functions in C++ programming language.

- **Setw – I/O Formatting in C++:** In C++ programming language, stew () function is used to set the number of characters to be used as the field width for the next insertion operation. The field width determines the minimum number of characters to be written in some output representations.

This manipulator is declared in header <iomanip.h>, along with the other parameterised manipulators. This header file declared the implementation-specific requirement for setw (). To understand the use of sew () function in C++ programming; let us look at the following programs:

Program:

A program to insert a tab character between two variables by using setw ()

```
//setw example
#include <iostream.h>
#include <iomanip.h>
void main (void)
{
    int x, y, z;
    x = 400;
    y = 500;
    z = 600;
    cout << x << '\t' << y << '\t' << z << endl;
}
```

Output

```
400      500      600
```

Here in the above program setw function has been used to insert a tab between three variables while displaying the content on the screen of computer.

- **Inline Functions:** An inline function is written in one line when they are invoked. These functions are very short, and contain one

or two statements. Inline functions are functions where the call is made to inline functions. The actual code then gets placed in the calling program.

Normally, a function call transfers the control from the calling program to the function and after the execution of the program returns the control back to the calling program after the function call. These concepts of function save program space and memory space and are used because the function is stored only in one place and is only executed when it is called. This execution may be time consuming since the registers and other processes must be saved before the function gets called.

The extra time needed and the process of saving is valid for larger functions. If the function is short, the programmer may wish to place the code of the function in the calling program in other for it to be executed. This type of function is best handled by the inline function.

The inline function takes the format as a normal function but when it is compiled it is compiled as inline code. The function is placed separately as inline function, thus adding readability to the source program. When the program is complied, the code present in function body is replaced in the place of function call.

General Format of inline Function

The general format of inline function is as follows:

inline datatype function_name(arguments)

The keyword inline specified in the above example, designates the function as inline function. For example, if a programmer wishes to have a function named exforsys with return value as integer and with no arguments as inline it is written as follows: inline int exforsys () .

Program:

```
#include <iostream.h>
using namespace std;
int exforsys (into);
void main ()
{
    int x;
    cout << "n Enter the Input Value:" ;
    cin>>x;
    cout << "n The Output is:" << exforsys(x);
}
```

```
inline int exforsys(int x 1)
{
    return 5*x1;
}
```

Output

Enter the input value: 10

The output is > 50

Press any key to continue

Q55. Discuss operators in C++ and its types.

Ans. C++, has a rich set of operators. Operators are the term used to describe the action to be taken between two data operands. Expressions are made by combining operators between operands. C++ supports six types of operators:

(1) Arithmetical operators: An operator that performs an arithmetic (numeric) operation such as +, -, *, /, or % is called arithmetic operator. Arithmetic operation requires two or more operands. Therefore these operators are called binary operations. The table 9.5 shows the arithmetic operators:

Table 9.5: Operators meaning with example

Operator	Meaning	Example	Answer
+	addition	8+5	13
-	subtraction	8-5	3
*	multiplication	8*5	40
/	division	10/2	5
%	modulo	5%2	1

(2) Relational operators: The relational operators shown in Table 9.6 are used to test the relation between two values. All relational operators are binary operators and therefore require two operands. A relational expression returns zero when the relation is false and a non-zero when it is true.

Table 9.6: Relational operators with example

Operator	Meaning	Example
==	Equal to	5==5
!=	Not equal to	5!=7
>	Greater than	7>5
<	Less than	8<9
>=	Greater than or equal to	8>=8
<=	Less than or equal to	9<=9

(3) Logical operators: The (!) operator is the C++ operator to perform the Boolean operation NOT. It has only one operand, located at its right, and the only thing that it does is to inverse the value of it, producing false if its operand is true and true if its operand is false. Basically, it returns the opposite Boolean value of evaluating its operand. Logical operators of C++ are given in Table 9.7.

Table 9.7: Logical operators

Operator	Meaning
&&	Logical AND
	Logical OR
!	Logical NOT

To understand the use of these operators in C++, let us take following example:

Example:

`!(5 == 5)` //evaluates to false because the expression at its right (`5 == 5`) is true.

`!(6 <= 4)` // evaluates to true because (`6 <= 4`) would be false.

`!true` // evaluates to false

`!false` // evaluates to true.

The logical operators `&&` and `||` are used when evaluating two expressions to obtain a single relational result. The operator `&&` corresponds with Boolean logical operation AND. This operation results true if both its two operands are true and false otherwise. The table 9.8 shows the result of operator `&&` by evaluating the expression `a && b`:

Table 9.8: Use of && Operators

Operand(a)	Operand(b)	Result
true	true	true
true	false	false
false	true	false
false	false	false

The operator `||` corresponds with Boolean logical operation OR. This operation results true if either one of its two operands is true, thus being false only when both operands are false themselves. To understand the use `||` OR operator, let us take the possible results of `a || b` in table 9.9.

Table 9.9: Use of || Operators

Operand(a)	Operand(b)	Result (a b)
true	true	true
true	false	true
false	true	true
false	false	false

Example:

((5 == 5) && (3 > 6)) // evaluates to false (true && false).

((5 == 5) || (3 > 6)) // evaluates to true (true || false).

(4) Bitwise Operator: In C++ programming language, bitwise operators are used to modify the bits of the binary pattern of the variables. Table 9.10 gives use of some bitwise operators:

Table 9.10: Use of Bitwise Operator

operator	Asm equivalent	Description
&	AND	Bitwise AND
	OR	Bitwise Inclusive OR
^	XOR	Bitwise Exclusive OR
~	NOT	Unary complement (bit inversion)
<<	SHL	Shift Left
>>	SHR	Shift Right

(5) Precedence of Operators: In case of several operators in an expression, we may have some doubt about which operand is evaluated first and which later. For example, let us take following expression:

a = 5 + 7 % 2

Here we may doubt if it really means:

A = 5 + (7 % 2) // with result of 6, or

a = (5 + 7) % 2 // with result of 0

The correct answer is the first of the two expressions, with a result of 6. Precedence order of some operators in C++ programming language is given in the Table 9.11.

Table 9.11: Precedence of operators in descending order

Level of Precedence	Operator	Description	Grouping
1	::	Scopc	Left-to-right
2	() [] . -> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid	postfix	Left-to-right
3	++ -- ~ ! sizeof new delete	unary (prefix)	Right-to-left
	* &	indirection and reference (pointers)	
	+ -	unary sign operator	
4	(type)	type casting	Right-to-left
5	. * ->*	pointer-to-member	Left-to-right
6	* / %	multiplicative	Left-to-right
7	+ -	additive	Left-to-right
8	<< >>	shift	Left-to-right
9	< > <= >=	relational	Left-to-right
10	== !=	equality	Left-to-right
11	&	bitwise AND	Left-to-right
12	^	bitwise XOR	Left-to-right
13		bitwise OR	Left-to-right
14	&&	logical AND	Left-to-right
15		logical OR	Left-to-right
16	? :	conditional	Right-to-left
17	= *= /= %= += -= >>= <<= &= ^= =	assignment	Right-to-left
18	,	comma	Left-to-right

Grouping defines the precedence order in which operators are evaluated in the case that there are several operators of the same level in an expression. Thus if you want to write complicated expressions and you are not completely sure of the precedence levels, always include parentheses. It will also make your code easier to read.

(6) Special Operators: Apart from the above operators that we have discussed above so far, C++ programming language supports some special operators. Some of them are: increment and decrement operator; size of operator; comma operator etc.

Increment and Decrement Operator: In C++ programming language, Increment and decrement operators can be used in two ways: they may either precede or follow the operand. The prefix version before the operand and postfix version comes after the operand. The two versions have the same effect on the operand, but they differ when they are applied in an expression. The prefix increment operator follows “change then use” rule and post fix operator follows “use then change” rule.

The size of operator: We know that different types of variables, constant, etc. require different amount of memory to store them. The sizeof operator can be used to find how many bytes are required for an object to store in memory.

Example:

sizeof (char) returns 1

sizeof (int) returns 2

sizeof (float) returns 4

if k is integer variable, the sizeof (k) returns 2.

The sizeof operator determines the amount of memory required for an object at compile time rather than at run time.

The comma operator

The comma operator gives left to right evaluation of expressions. It enables to put more than one expression separated by comma on a single line.

Example:

int i = 20, j = 25;

In the above statements, comma is used as a separator between the two statements.

(7) Escape Sequence: There are some characters which can't be typed by keyboard in C++ programming language. These are called non-graphic characters. An escape sequence is represented by backslash (\) followed by one more characters. The table 9.12 gives a listing of common escape sequences.

Table 9.12: Escape Sequence

Sequence	Task
\a	Bell (beep)
\b	Backspace
\f	Formatted
\n	Newline or line feed
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\?	Question mark
\\	Backslash
\'	Single quote
\"	Double quote
\xhh	Hexadecimal number (hh represents the number in hexadecimal)
\000	Octal number (00 represents the number in octal)
\0	Null

Punctuators

In C++ programming language, following characters are used as punctuators for enhancing the readability and maintainability of programs.

Brackets []	opening and closing brackets indicate single and multidimensional array subscript.
Parentheses ()	opening and closing brackets indicate functions calls, function parameters for grouping expressions etc.
Braces { }	opening and closing braces indicate the start and end of a compound statement.
Comma ,	it is used as a separator in a function argument list.
Semicolon ;	it is used as a statement terminator.
Colon :	it indicates a labelled statement or conditional operator symbol.
Asterisk *	it is used in pointer declaration or as multiplication operator
Equal sign =	it is used as an assignment operator.
Pound sign #	it is used as pre-processor directive





SCHOLARSHIP



We Care for You!

यदि आप **Corona** की वजह से **IGNOU** की
फीस का इंतजाम नहीं कर पा रहे हैं तो नीचे
दिए गए मोबाइल फोन पर संपर्क करें।

Call or Whatsapp. 9212232302

Sample Papers

C++ Programming: BCS-031

Sample Paper-I

Note: All Questions are compulsory.

Q1. What is the meaning of data abstraction? What is encapsulation? How does C++ achieve these two concepts? Describe with the help of an example.

Ans. Refer to Chapter-2, Q.No.-2

Q2. Design a template class that is used to create a circular linked list.

Ans. Refer to Chapter-7, Q.No.-3

Q3. What are the advantages of object oriented programming? How is it different from procedural programming? What are the disadvantages of object oriented systems?

Ans. Refer to Chapter-1, Q.No.-1

Q4. What is a reference variable? What is the usage of references in parameter passing? Why are references needed when call by value exists? What is the purpose of “const” keyword with respect to references?

Ans. Refer to Chapter-4, Q.No-5

Q5. Write short notes on *any two* of the following:

(i) **Exception Handling**

Ans. Refer to Page No.-161 [Exception Handling]

(ii) **Multiple and multilevel inheritance**

Ans. Refer to Page No.-95 [Multilevel inheritance] and Page No.-96 [Multiple inheritance]

(iii) **&& and || Operator**

Ans. Refer to Chapter-9, Q.No.-16(xii)

C++ Programming: BCS-031

Sample Paper-II

Note: All Questions are compulsory.

Q1. Differentiate between object-oriented and object-based programming languages.

Ans. Refer to Chapter-9, Q.No.-47

Q2. Design and implement the class stack implement constructor, destructor and other suitable function.

Ans. Refer to Chapter-3, Q.No.-3

Q3. Explain Pure Virtual Functions with example. Write down the Characteristics of Pure Virtual Functions.

Ans. Refer to Page No.-103 [Pure Virtual Functions] and Chapter-9, Q.No.-50

Q4. What is the purpose of using friend functions? Describe with an example. Can a class be made friend of other class? Justify your answer.

Ans. Refer to Chapter-4, Q.No.-1

Q5. Write short notes on *any two* of the following:

(i) Stream Class Hierarchy

Ans. Refer to Page No.-138 [Stream Class Hierarchy]

(ii) Bridge Function

Ans. Refer to Chapter-4, Q.No.-9

(iii) Scope Resolution Operator

Ans. Refer to Page No.-16 [Scope Resolution Operator]

C++ Programming: BCS-031

December, 2012

Note: Question number 1 is **compulsory** and carries 40 marks. Attempt any **three** questions from the rest.

- Q1.** (a) Explain features of structured programming paradigm in brief. Also list its advantages and disadvantages with respect to Object Oriented Paradigm.
- (b) What is encapsulation? Explain how it is different from information hiding with the help of an example programme to manage Books issue and return in a Library.
- (c) What is data type? List any four built in data types and their size in C++. Also explain need of derived data type in C++ programming.
- (d) What is an object? Explain how objects in C++ are created and destroyed, with the help of programme to create Bank-Account Class, having data members name, account-number, balance and member function display balance.
- (e) Explain the concept of copy constructor with the help of an example programme.
- (f) What is function template? Write a function template SUM to add two numbers.
- Q2.** (a) Write a program in C++, which take two 3×3 matrices as input and find sum of them. Implement suitable constructor and destructor for this programme.
- (b) What is message passing? Explain how message passing is used in C++ programming with an example.
- Q3.** (a) What is access control specifier? Explain the need of different access control specifiers with example.
- (b) What is constructor? Explain advantage of constructor with the help of an example.

- (c) Write a C++ program to create a class Book, to keep the records of books available in your library. This program should have proper constructor and member functions, to get the details such as publisher, author and price etc. of the books. Make necessary assumptions where ever required.
- Q4. (a) Explain need of operator overloading. Also explain why some operators can not be overloaded? Write a C++ program to overload '+' operator to add two character strings.
(b) What is data stream? Explain stream hierarchy in C++.
(c) What is friend function? Explain its advantage with the help of an example.
- Q5. (a) What is polymorphism? Explain advantage of polymorphism. Also write a C++ programme to explain use of virtual function.
(b) What is exception? Explain how exception handling is done in C++ with the help of a programme. Also describe what will happen if an exception is thrown out side of a try block and *why?*

○○○



Gullybaba Publishing House (P) Ltd.

ISO 9001:2008 & 14001:2004 Certified Co.

Offering

Help Books

that are unparallel in quality
For



IGNOU & Other Universities

VISIT:

GullyBaba.com

C++ Programming: BCS-031

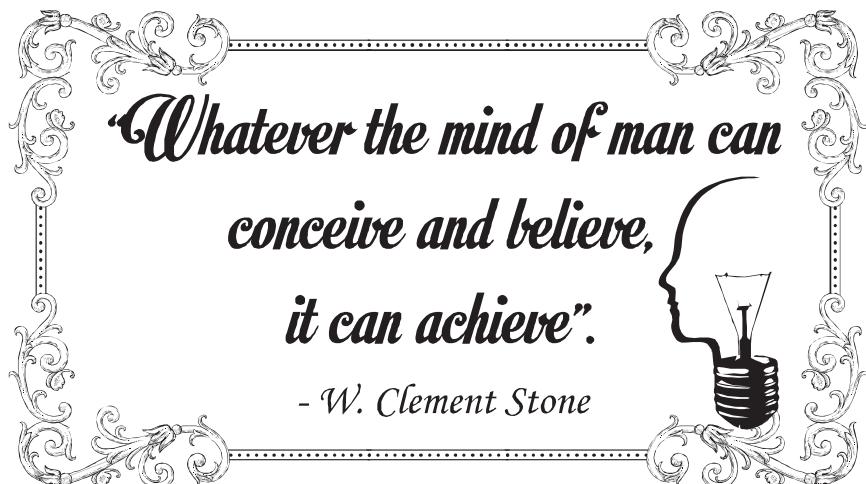
June, 2013

Note: Question number 1 is **compulsory** and carries 40 marks. Attempt any **three** questions from the rest.

- Q1.** (a) What is object-oriented programming paradigm? Explain advantages of object-oriented programming paradigm over structured programming paradigm.
- (b) Write a C++ programme to create a class NUMBER with a constructor to initialise object of NUMBER class with three integer data values. Define a function *largest* to find the largest number among the three member data.
- (c) What is need of memory management in C++ programming? Explain in brief about memory management process in C++.
- (d) What is a virtual function? Write a programme in C++ to create class Doctor with a virtual function salary. Derive class Visiting-Doctor and implement function *salary* in it.
- (e) What is operator overloading? Briefly explain general rules of operator overloading.
- (f) What is stream manipulator? Explain use of *setw ()* and *setprecision ()* as stream manipulator.
- Q2.** (a) Differentiate C++ programming language from C programming language in terms of parameter passing in functions.
- (b) What is data type? Draw hierarchy of data types in C++.
- (c) Write a C++ programme to implement simple calculator to perform '+, -, *, /' on two integer operands. Your programme should have methods for reading data and for performing arithmetical operations.
- Q3.** (a) Write a C++ programme to open an existing file and insert the text "My C++ File" at the end of it.

- (b) What is reusability of code? Write a C++ programme to create a class student, with basic data members such as name, address, age. Create a class PG_student by inheriting from student class. PG_student class should have a function to display, name, address, subject of the student.
- Q4. (a) What is need of exception handling in C++ programming? Explain with an example how exceptions are handled in C++. Briefly describe the hierarchy of exception classes in C++ standard library.
- (b) What is template class? Explain advantages of template class. Create a template class for Linked-List data structure.
- Q5. (a) What is function overloading? How it is different from function overriding? Explain with an example of each.
- (b) How function calls are matched with overloaded functions in C++? Explain with the help of a C++ programme.

○○○



C++ Programming: BCS-031

December, 2013

Note: Question number 1 is **compulsory** and carries 40 marks. Attempt any **three** questions from the rest.

- Q1.** (a) Explain the basic characteristics of object oriented programming (OOP). Also explain any three advantages of OOP over procedural programming languages.
- (b) What is an operator? List the various types of operators used in C++.
- (c) What is meant by object initialisation? What is its need? Explain with the help of a suitable example.
- (d) What are friend functions? Explain two merits and two demerits of using friend functions, with the help of an example.
- (e) What is slope resolution operator? Explain its use with the help of a C++ programme.
- (f) What is virtual function? Explain advantage of using virtual function in C++, with the help of an example.
- Q2.** (a) What is exception handling? How is it performed in C++? Explain with the help of an example.
- (b) Write an object oriented programme in C++ to read a set of integer numbers. Upto n, where n is defined by the user and print the contents of the array in the reverse order using a class template.
- Q3.** (a) Write a programme in C++ to find the largest of any three numbers using a member function defined in a class.
- (b) What is static member? Explain use of static data member and static member function with the help of an example programme in C++.
- Q4.** (a) List the merits and demerits of single inheritance over multiple inheritance.

- (b) What is polymorphism? Explain any three advantages of polymorphism.
- (c) What is container? List main types of container in C++. Also list some common member functions of container classes.
- Q5. Write short note on the followings:
- (a) Abstract classes
 - (b) Input and output streams
 - (c) Operator overloading
 - (d) Class and objects

○○○

“It doesn't matter who you are, where you come from. The ability to triumph begins with you – always”.

-Oprah Winfrey



C++ Programming: BCS-031

June, 2014

Note: Question number 1 is **compulsory** and carries 40 marks. Attempt any **three** questions from the rest.

- Q1.** (a) Explain the basic characteristics of object oriented languages. How is object oriented programming language better than structured programming language?
- (b) What do you mean by copy constructor? Explain it with a suitable C++ program.
- (c) What is meant by comparison and logical operators? How are they different from the arithmetic and assignment operators, explain with the help of an example.
- (d) Explain function template with the help of an example.
- (e) What is looping in C++? What are the advantages of using loops in C++? Also list the various looping options available in C++.
- (f) What is a structure in C++, and how a structure is different from a class? Explain with example.
- Q2.** (a) What is exception handling? What are the keywords used to handle the exception in C++? Write a C++ program to handle divide by zero exception.
- (b) Write a programme in C++ using operator template for the binary numbers to perform a simple arithmetic operations such as add and subtract.
- Q3.** (a) Write a programme in C++ that prints numbers and its cubes from 1 to 10 by using if-then-else and for loop.
- (b) Explain the use of *continue* statement in C++, with example.
- (c) Explain how setting of field width and setting of precision may be done in C++.

- Q4.** (a) Write a C++ programme to create vehicle class and derive Car, Truck and Bike classes from the Vehicle class. Also define proper constructors for each of these classes.
- (b) Explain the concept of virtual function with the help of an example.
- (c) What are access control specifiers? Explain difference between private and public access control specifiers.
- Q5.** Write short note on the following:
- (a) Encapsulation
- (b) Message Passing
- (c) Function Overloading
- (d) File pointers and operations.

○○○

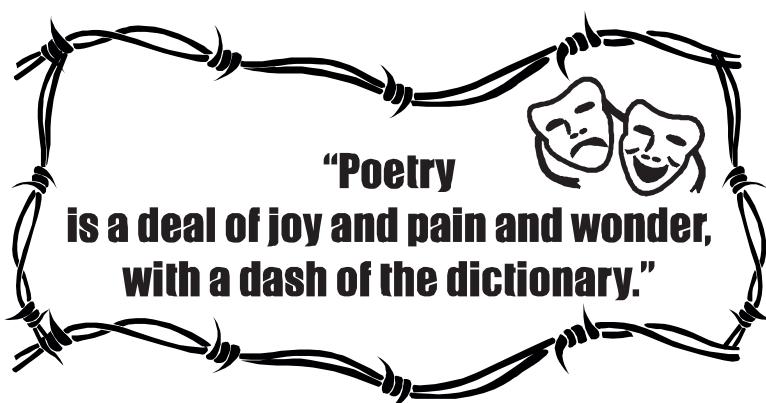
C++ Programming: BCS-031

December, 2014

Note: Question number 1 is **compulsory** and carries 40 marks. Attempt any **three** questions from the rest.

- Q1.** (a) Why did people change over from structured programming to object-oriented programming? Explain it in brief with the help of an example.
- (b) Write C++ program to find factorial of a given number using copy constructor.
- (c) What do you mean by *inline* function? What is its importance in object-oriented programming languages. Write C++ program to clarify the concept of inline function.
- (d) Differentiate class template and function template with the help of an example.
- (e) Write C++ program to implement multilevel inheritance. Provide the necessary comments to clarify the availability of data members and member functions in different classes.
- (f) What is an object in C++? Explain how an object can be passed as an argument to a function with the help of an example.
- Q2.** (a) What do you mean by constructors ? Write the characteristics of constructors. Write a program to illustrate the use of constructor in C++ programming.
- (b) What happens if we don't use the virtual function in the inheritance? Explain the importance of virtual function in the reference of the above, with the help of an example.
- Q3.** (a) What are the needs of operator overloading in the program? Why can't some operators be overloaded? Write C++ program to add two complex numbers using plus (+) operator overloading.
- (b) Why does abstract class play an important role in object-oriented programming? Write C++ program of abstract class which uses the concept of pure virtual function.

- Q4. (a) What do you mean by polymorphism? How is run time polymorphism different from compile time polymorphism? Give example(s) to support the above differentiation.
- (b) What do you mean by the file stream operations? Write C++ program to demonstrate the reading from disk file and writing the result to the disk file.
- (c) What is friend function? Explain its concept with the help of a suitable example.
- Q5. (a) Describe all types of containers that are available in C++ with their importance, in detail.
- (b) Explain a situation when you will use multiple *catch* statements in a C++ program for exception handling.
- (c) Write a C++ program to find the area of a circle.



C++ Programming: BCS-031

June, 2015

Note: Question number 1 is **compulsory** and carries 40 marks. Attempt any **three** questions from the rest.

- Q1.** (a) Explain object oriented concepts. How is object oriented language different from structured programming language?
(b) What is inheritance? Explain different types of inheritance supported by C++.
(c) Differentiate between default constructor and parameterized constructor with the help of an example.
(d) What is an abstract class? How do you create an abstract class? What is the purpose of creating an abstract class in object oriented programming paradigm? Explain with the help of an example.
(e) Write a C++ program to add two complex numbers. In this program you need to create complex class and define proper constructor for object initialization.
- Q2.** (a) What do you understand by friend function? Write a C++ program to find out the sum of n given numbers using friend function.
(b) Explain the difference between private, protected and public access specifier with respect to class and its object. Write a program in this support.
- Q3.** (a) What do you mean by operator-overloading? List the operators which cannot be overloaded. Write a C++ program for unary minus (-) operator overloading.
(b) Explain the concept of virtual function with its important characteristics. Write a C++ program to illustrate the importance of pure virtual function.
- Q4.** (a) What do you mean by *this* pointer? Explain the use of *this* pointer with the help of an example.

- (b) Write a C++ program to display the price-list of five vegetables. Use precision() function to set precision 2 for display price.
- (c) What is function template? Write a function template to swap two given numbers.
- (a) What is containership? Write the important containers available in C++ with their importance.
- (b) What do you mean by exception handling? Write the syntax of try, throw and catch expressions. Write a program to catch all the exceptions in C++ programs.

Poetry....

allows me to express emotions
that lie dormant in
the darkness of my soul until
at last pen meets paper.

~ Bonnidette Lantz



C++ Programming: BCS-031

December, 2015

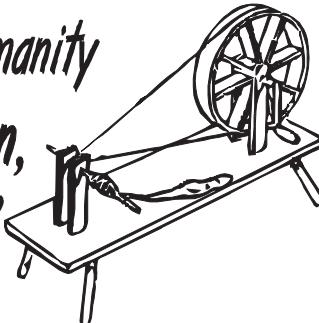
Note: Question number 1 is **compulsory** and carries 40 marks. Attempt any **three** questions from the rest.

- Q1.** (a) What is structured programming? Write the disadvantages of structured programming.
- (b) How do you input non-graphic characters in the C++ programming language? What do these escape sequences represent?
- (i) \n
(ii) \a
(iii) \v
- (c) Write a C++ program to demonstrate the use of switch statement.
- (d) Write the appropriate statements to create a function template printarray that can display the values contained in array passed as parameter to the function. The function must be able to accept integer, float and character arrays as arguments.
- (e) Describe the concept of classes and objects through examples.
- (f) What are Container Classes? List the different types of containers and give three examples of each.
- (g) List any five common examples of exceptions.
- Q2.** (a) Consider an example of declaring the examination result of BCA students of IGNOU. Design three classes: Student, Exam and Result. The Student class has data members such as those representing roll no., name, etc.
- Create the class Exam by inheriting Student class. The Exam class adds fields representing the marks scored in 6 subjects.
- Derive the Result from the Exam class, and it has its own fields such as total marks.

Write an interactive program to model this relationship by showing the three classes. Also, show how you can print out the values of each member of these classes.

- (b) Define member function. Explain the purpose of member function.
- Q3. (a) Define friend function. Discuss memory requirements for classes, objects, data members and member functions with an example.
- (b) Write a C++ program to print Student details: Student Name, Father's Name, Mother's Name, Student Address, Zip code, Student Roll No. using constructor and destructor.
- Q4. (a) Explain briefly the importance of pure virtual function in the software development paradigm. Write a C++ program with abstract class having pure virtual function.
- (b) Explain exceptions in C++ program through an example.
- Q5. (a) Explain Inheritance and Multi-Inheritance with an example. Write a C++ program which has Inheritance and Multi-Inheritance.
- (b) Define Operator Overloading. Write the general rules for Operator Overloading.

*"The greatness of humanity
is not in being human,
but in being humane."*



C++ Programming: BCS-031

June, 2016

*Note: Question number 1 is **compulsory** and carries 40 marks. Attempt any **three** questions from the rest.*

Q1. (a) What is Software Re-usability? Explain in the context of C++ with an example.

Ans. Software re-usability is primary attribute of software quality. C++ strongly supports the concept of reusability. C++ features such as classes, virtual functions, and templates allow designs to be expressed so that re-use is made easier (and thus more likely), but in themselves such features do not ensure re-usability. Do we really need re-use? This question can best be answered by an analogy from automobile industry. Consider the design and creation of a new car model. The automotive engineer does not design a new car from scratch. Rather, the engineer borrows from the design of existing cars. For example, the engine design from an existing car may be used in a new model. If the engine design has been used in a previous model, design problems have likely been resolved. Thus, development costs are reduced because a new engine does not need to be designed and tested. Finally, consumer maintenance costs are reduced because machines and others who must maintain the car are already familiar with the operation of the engine.

The American Heritage Dictionary defines quality as “a characteristic or attribute of something”. Re-usability is the degree to which a thing can be reused. Software re-usability represents the ability to use part or the whole system in other systems which are related to the packaging and scope of the functions that programs perform. The need for re-usability comes from the observation that software systems often follow similar patterns; it should be possible to exploit this commonality and avoid reinventing solutions to problems that have been encountered before.

There are many advantages of re-usability. They can be applied to reduce cost, effort and time of software development. It also increases the productivity, portability and reliability of the software product.

- (b) Define visibility of a class member. Why are different types of visibility modes needed in derivation of a derived class?

Ans. Refer to Chapter-9, Q.No.-33

- (c) Explain C++ streams and stream classes with an example.

Ans. Refer to Chapter-7, Page No.-138 [Stream Class Hierarchy] and Refer to Chapter-9, Q.No.-54

- (d) List the C++ operators that cannot be overloaded. Give reasons for any two of these explaining why they cannot be overloaded.

Ans. Generally the operators that can't be overloaded are like that because overloading them could and probably would cause serious program errors or it is syntactically not possible. Following section describe the reason for not overloading some of the operators defined in C++ language.

(1) **size of operator:** The size of operator returns the size of the object or type passed as an operand. It is evaluated by the compiler not at runtime so you can not overload it with your own runtime code. It is syntactically not possible to do.

(2) **? :** (conditional operator): All operators that can be overloaded must have at least one argument that is a user-defined type. That means you can't overload that operator which has no arguments. But it does not suite for? : (conditional operator) as it does not take name as parameter. The reason we cannot overload? : is that it takes 3 argument rather than 2 or 1. There is no mechanism available by which we can pass 3 parameters during operator overloading.

- (e) Explain file pointers and operations. Write a C++ program which demonstrates the use of put () – get () and read () – write () .

Ans. We can further control the reading and writing operations from file by manipulating the file pointers. Every file in C++ is marked by two pointers, one for input (called get pointer) and one used for output (called put pointer). The get pointer can be set to a specified location in order to reach from that desired location in the file. These pointers are automatically incremented every time after every read and write operation. You may be wondering how we have been able to read and write in our earlier programs without setting these pointers. Actually every time we open a file for input, the input pointer is automatically set to the beginning of the file.

We can manipulate these file pointers by invoking member functions of the file stream class. The commonly used functions for this purpose include seekg(), seekp(), tellg(), tellp() etc. The seekg() function moves the input (get) pointer to a specified location. The seekp() function moves to output (put) pointer to a specified location. The tellg() and tellp() functions tell the current position of get and put pointers, respectively. The general syntax for using seekg() and seekp() is given below:

```
seekg(offset, refposition);
seekp(offset, refposition);
```

Where, offset represents the number of bytes, the file pointer is to be moved from the location specified by the parameter refposition. The refposition may take one of the three positions defined in the ios class:

ios::beg – start of the file

ios::cur – current position of the pointer

ios::end – end of file.

The seekg() function moves the get pointer whereas the seekp() function moves the put pointer as per the parameters specified in the statement.

Functions to read from and write to files

Reading and writing operations in files are made simpler by C++ by providing some member functions in the file stream class. The functions put() and get() are used for handling a single character at a time. Functions read() and write() are designed to handle blocks of binary data. We also have functions like getline().

Two programming examples demonstrating use of put() – get() and read() - write() are given below:

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>
int main()
{
char name [80];
cin >> name;
int len = strlen(name);
fstream file;
file.open("text", ios::in | ios::out);
for (int i = 0; i < len; i++)
file.put(name[i]);
file.seekg(0);
char c;
while (file)
{
file.get(ch);
```

```
cout << ch;  
}  
return (0);  
}
```

(f) Differentiate between information hiding and encapsulation.

Ans. Refer to Chapter-9, Page No.-13-14 (Encapsulation and Data Hiding)

(g) What do you mean by global variable and local variable in C++? Distinguish with an example.

Ans. Refer to Chapter-2, Page No.-16-17 (The Scope Resolution Operator (::))

Q2. (a) Write a C++ program to demonstrate break and continue statement.

Ans. (a) break statement

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

Output:

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

(b) continue statement

```
// continue loop example  
#include <iostream.h>  
using namespace std;
```

```
int main ()  
{  
for (int n=10; n>0; n--) {  
if (n==5) continue;  
cout << n << ", ";  
}  
cout << "FIRE!\n";  
return 0;  
}
```

Output:

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

- (b) Write a C++ program which explains user-defined stream manipulators.**

Ans. C++ allows users to design their own customised stream manipulators. These user-defined manipulators may be non-parameterized or parameterized ones. The general syntax for creating a user-defined stream manipulator without any argument is as follows:

Ostream & manipulator (ostream & output)

```
{  
.....  
..... Statements for formatting  
.....  
return output;  
}
```

Here, the manipulator is the name of user-defined manipulator to be created. For example, we can create the user defined manipulator unit that displays “INR” after each numerical value displayed.

ostream & unit (ostream & output)

```
{  
output << "INR";  
return output;  
}
```

A more complex user-defined manipulator show can be defined as follows:

ostream & show (ostream & output)

```
{
```

```
output.setf (ios::showpoint);
output.setf (ios::showpos);
output << setw (10);
return output;
}
```

The following program presents a full blown example of designing user-defined stream manipulators. This program creates two user-defined manipulators currency and form, which are used to display output values in a specific manner.

```
# include <iostream.h>
#include <iomanip.h>
ostream & currency(ostream & output)
{
output << "INR";
return (output);
}
ostream & form(ostream & output)
{
output.setf (ios::showpos);
output.setf (ios::showpoint);
output.fill ('*');
output.precision (2);
output << setiosflags (ios::fixed) << setw (10);
return (output);
}
int main()
{
cout << currency << form << 5465.4;
return (0);
}
```

The output of the program would be INR **+5465.40.

Another example code which makes use of stream manipulators and other formatting techniques is given below. This program creates two user-defined manipulators area and volume, which are used to display output values in a specific manner. The value of area is displayed with SQ.MTS unit and the volume is displayed with CUBIC MTS unit.

```
# include <iostream.h>
#include <iomanip.h>
ostream & area(ostream & output)
{
    output << "SQ.MTS";
    return (output);
}
ostream & volume(ostream & output)
{
    output.precision (4);
    output << setiosflags (ios::fixed) << setw (15);
    output << "CUBIC MTS"
    return (output);
}
int main()
{
    cout << area << 3000;
    cout << volume << 9000;
    return (0);
}
```

Q3. (a) Distinguish between class templates and function templates, through an example.

Ans. Refer to Chapter-7, Page No.-135-139 (Function Templates)

(b) Write an interactive C++ program which reads two integer numbers x and y, and an operator. It then performs the following operations.

- (i) x + y**
- (ii) x - y**
- (iii) x * y**
- (iv) x / y**

If any other operator is entered, the program prints out an error message.

Ans. #include <iostream.h>

```
#include<stdlib.h>
```

```
class A
```

```
{
```

```
int x,y;
public :
void input_xy(void);
int get_x(void);
int get_y(void);
int add(void);
int sub(void);
};

class B : public A
{
public :
int mul(void);
int div(void);
void display(int opt, int res);
};

void A :: input_xy()
{
cout<< "\n Enter the value of x and y :"<<endl;
cin>>x>>y;
}

int A :: get_x()
{
return x;
}

int A :: get_y()
{
return y;
}

int A :: Add()
{
return (x+y);
}

int A :: Sub()
{
```

```
return (x-y);
}
void B :: Mul()
{
return(get_x()*get_y());
}
void B :: Div()
{
return(get_x()/get_y());
}
void B :: display(int choice, int result)
{
cout<< "\n The value of a is :" <<x<< endl;
cout<< "\n The value of b is :" <<y<< endl;
switch(choice)
{
case 1 : cout << "\n The sum of x and y is :" <<result<<endl;
break;
case 2 : cout << "\n The subtraction of x and y is :" <<result<<endl;
break;
case 3 : cout << "\n The multiplication of x and y is :" <<result<<endl;
break;
case 4 : cout << "\n The division of x and y is :" <<result<<endl;
break;
}
}
void main()
{
B objb;
int choice;
int result;
objb.input_xy();
while (1)
{
```

```
cout << " Operations on two numbers ..." << endl;
cout << " 1. Addition" << endl;
cout << " 2. Subtraction" << endl;
cout << " 3. Multiplication" << endl;
cout << " 4. Division" << endl;
cout << " 5. Quit" << endl;
cout << " Enter choice:" << endl;
cin >> choice;
switch(choice)
{
    case 1 : result=objb.add();
    objb.display(choice,result);
    break;
    case 2 : result=objb.sub();
    objb.display(choice,result);
    break;
    case 3 : result=objb.mul();
    objb.display(choice,result);
    break;
    case 4 : if (y!=0)
    {
        result=objb.div();
        objb.display(choice,result);
    }
    else
        cout << "\nDivide by zero error:" << endl;
    break;
    case 5 : exit(1);
    break;
    default :
        cout << " Bad option selected" << endl;
        continue;
}
```

Q4. (a) What are base and derived classes? Create a base class called Stack and a derived class called Mystack? Write an interactive C++ program to show the operations of a stack.

Ans. Inheritance is a property by which one class inherits the feature of another class. The class which inherits the feature from another class is called derived class and class from which another class takes feature is called base class.

```
#include <iostream.h>
#include<stdlib.h>
#define Max_Size 5 //Maximum stack size
class Stack
{
protected :
int stack[Max_Size];
int top;
public :
Stack (void)
void push (int item);
void pop (int &item);
};
class MyStack : public Stack
{
public :
int push(int item);
int pop(int &item);
void stackContent(void);
};
Stack::Stack()
{
top=-1; //Stack empty
}
void Stack::push(int item)
{
top++;
stack[top]=item;
}
```

```
void Stack::pop(int &item)
{
item=stack[top];
top--;
}
int MyStack :: push(int item)
{
If (top<Max_Size-1)
{
Stack::push(item);
return 1; //push operation successful
}
cout<< "\n Stack Overflow :"<< endl;
return 0;
}
int MyStack :: pop(int &item)
{
If (top>=0)
{
Stack::pop(item);
return 1; //push operation successful
}
cout<< "\n Stack Underflow :"<< endl;
return 0;
}
void MyStack :: stackContent(void)
{
int stop;
stop=top;
for (int i=0; i<=stop;i++)
cout<< ":"<<stack[i];
}
void main()
{
MyStack stack;
```

```
int choice;
int item;
while (1)
{
    cout << "\nStack Operation ..." << endl;
    cout << "\n1. Item to push?" << endl;
    cout << "2. Item to pop" << endl;
    cout << "3. Quit" << endl;
    cout << "Enter choice:" << endl;
    cin >> choice;
    switch(choice)
    {
        case 1 : cout << "Enter the item:" << endl;
        cin >> item;
        cout << "\n Stack content before push operation:" ;
        stack.stackContent();
        if ((stack.push(item))==1)
        {
            cout << "\n Stack content after push operation:" ;
            stack.stackContent();
        }
        break;
        case 2 : cout << "\n Stack content before pop operation:" ;
        stack.stackContent();
        if ((stack.pop(item))==1)
        {
            cout << "\n Stack content after pop operation:" ;
            stack.stackContent();
            cout << " popped item:" << item;
        }
        break;
        case 3 : exit(1);
        break;
    default :
```

```
cout << " Bad option selected" << endl;
continue;
}
}
}
```

- (b) Define Polymorphism. Explain various types of polymorphism with examples.**

Ans. Refer to Chapter-9, Q.No.-4

In Polymorphism we have 2 different types those are as follows:

Compile Time Polymorphism

Compile time polymorphism means we will declare methods with same name but different signatures because of this we will perform different tasks with same method name. This compile time polymorphism also called as early binding or method overloading.

Method Overloading or compile time polymorphism means same method names with different signatures (different parameters)

Example:

```
public class Class1
{
    public void NumbersAdd(int a, int b)
    {
        Console.WriteLine(a + b);
    }
    public void NumbersAdd(int a, int b, int c)
    {
        Console.WriteLine(a + b + c);
    }
}
```

In above class we have two methods with same name but having different input parameters this is called method overloading or compile time polymorphism or early binding.

Run Time Polymorphism

Run time polymorphism also called as late binding or method overriding or dynamic polymorphism. Run time polymorphism or method overriding means same method names with same signatures.

In this run time, polymorphism or method overriding we can override a method in base class by creating similar function in derived class this can

be achieved by using inheritance principle and using “virtual & override” keywords.

In base class, if we declare methods with virtual keyword then only we can override those methods in derived class using override keyword.

Example:

```
//Base Class
public class Bclass
{
    public virtual void Sample1()
    {
        Console.WriteLine("Base Class");
    }
}

// Derived Class
public class DClass : Bclass
{
    public override void Sample1()
    {
        Console.WriteLine("Derived Class");
    }
}

// Using base and derived class
class Program
{
    static void Main(string[] args)
    {
        // calling the overriden method
        DClass objDc = new DClass();
        objDc.Sample1();

        // calling the base class method
        Bclass objBc = new DClass();
        objBc.Sample1();
    }
}
```

If we run above code we will get output like as shown below:
Output

Derived Class

Derived Class

Q5. (a) Write a C++ program which explains the usage of Try, Throw and Catch.

Ans. Refer to Chapter-8, (Process of error handling)

(b) Write a C++ program for finding the difference between two times that are given in 24 hour format. So "19:00:00"-“3:30:00”=15:30:00, while “09:00:00”-“13:30:00”= “19:30:00”.

Ans. void difftime(int time_1, int time_2)

```
{  
    // time_1 , time_2 in the format  
    // XXYY  
    int hr_1 = time_1 / 100;  
    int hr_2 = time_2 / 100;  
    // to extract hr from XXYY divide it by 100  
    hr_2 = (hr_1 > hr_2) ? (hr_2 + 24) : hr_2;  
    // say hr_1 is 0900 i.e., 9 hr 00 min  
    // and hr_2 is 1330 i.e., 13 hr 30 min  
    int min_1 = time_1 % 100;  
    int min_2 = time_2 % 100;  
    // to extract min from XXYY modular divide it by 100  
    int diffmin, diffhr;  
    if (min_2 > min_1)  
    {  
        diffmin = min_2 - min_1;  
    }  
    else  
    {  
        diffmin = (min_2 + 60) - min_1;  
        hr_2--;  
    }  
    // because we are borrowing 1 hr/60 min from hr_2  
    diffhr = hr_2 - hr_1;  
    cout << " Time difference : " << diffhr << " hour " << diffmin << "  
minute." << endl;  
}
```

C++ Programming: BCS-031

December, 2016

*Note: Question number 1 is **compulsory** and carries 40 marks. Attempt any three questions from the rest.*

- Q1.** (a) Explain the basic features of an object-oriented language. Why did people change over from structured programming to object-oriented programming?
(b) Abstract class provides a base upon which other classes may be built. Justify the above statement with the help of an example.
(c) What do you mean by inheritance? Explain the advantages of using multiple inheritance in C++ with the help of an example.
(d) Explain the importance of a constructor in object-oriented programming. Differentiate between copy constructor and default constructor in C++ with the help of an example.
(e) What is function overloading? Give its advantages in a C++ program. Also write a C++ program to show function overloading.
(f) How does a virtual function differ from a pure virtual function? Also give an example of a pure virtual function.
- Q2.** (a) What is exception handling ? What is the sequence of events when an exception occurs? Write a C++ program that uses exception handling to handle the errors caused, when a number is divided by zero.
(b) Differentiate among private, public and protected access modifiers. Also explain their meaning when a derived class inherits from a base class using public, protected or private keywords, with the help of an example.
- Q3.** (a) Write a program to add two complex numbers by using binary operator overloading. Write comments in the program wherever it is required, to give more clarity to the program.
(b) Write a program in C++ to calculate the factorial of a given number.

- (c) Explain the association of dynamic binding and run-time polymorphism, with example.
- Q4. (a) How is unformatted I/O different from formatted I/O ? Explain.
(b) Write a C++ program to create a Book class. Define constructor and destructor for this class. Also define the methods to show the title and price of the books.
(c) Explain the use of the following operators in C++:
(i) &
(ii) ?:
(iii) ::
(iv) &&
- Q5. Write short notes on the following:
(a) New and Delete Operator
(b) Parameterized Constructor
(c) Class Template
(d) Pure Virtual Function

○○○

Gullybaba Publishing House (P) Ltd.
ISO 9001 & ISO 14001 CERTIFIED CO.

Offering
Help Books
that are unparalleled in quality for
IGNOU & NIOS
& Other Universities

VISIT:
GullyBaba.com

C++ Programming: BCS-031

June, 2017

Note: Question number 1 is **compulsory** and carries 40 marks. Attempt any **three** questions from the rest.

- Q1.** (a) Why are object oriented programming languages more popular than structured programming languages ? Differentiate between structured and object oriented programming languages.
(b) Explain ambiguity resolution in multiple inheritance. What happens if we don't use a virtual function in the inheritance?
(c) Write a C++ program to create a matrix class. Define constructor and destructor for this class. Also define a method to find the sum of two matrices.
(d) Define the Standard Template Library. How is the class template different from the function template? Explain.
(e) Differentiate between private, protected and public access modifiers with the help of an example for each.
(f) How is constructor different from the 'constructor with argument'? Explain by using an example.
- Q2.** (a) Write a program to demonstrate the catching of all exceptions. What happens when a raised exception is not caught by catch-block (in the absence of catching all exception blocks)?
(b) Write a program to implement the overloading of << operator.
- Q3.** (a) How do we declare static class data? Explain the syntax and rules to define static class data.
(b) Write a short program to implement the concept of passing object as argument.
(c) Write the general form of the user-defined manipulators. Design a single manipulator format to provide the following output specifications for printing float values:

- (i) 10 column width
- (ii) Right Justified
- (iii) Two-digit precision
- (iv) Filling of unused places with *
- (v) Trailing zeroes shown

- Q4. (a) What is the importance of Abstract Class? Write a program to implement the concept of abstract class in C++. Also explain why an abstract class cannot be instantiated.
- (b) Write a program to calculate the factorial of a given number by using copy constructor. Also write comments in your program wherever required.
- Q5. Write short notes on the following:
- (a) Destructor
 - (b) Pure Virtual Function
 - (c) Friend Function
 - (d) Multiple Inheritance

ooo

*Whenever you do
a thing, act as if all
the world watching.
- Thomas Jefferson*

C++ Programming: BCS-031

December, 2017

*Note: Question number 1 is **compulsory** and carries 40 marks. Attempt any **three** questions from the rest.*

Q1. (a) What do you mean by Abstraction and Encapsulation? How are the two terms interrelated?

Ans. Refer to Chapter-2, Q.No.-2

Encapsulation means wrapping up of data and functions which operate on the data into a single unit and ensures that only essential features get represented without representing the detail background, *i.e.*, called Abstraction. Therefore, both are interrelated.

(b) What is a Reference Variable? What is its usage?

Ans. Refer to Chapter-4, Q.No.-5

(c) Identify the errors in the following code segment:

```
int main ()  
{  
    cout<< "Enter two numbers";  
    cin >> num>> auto;  
    float area = length * breadth;  
}
```

Ans. int main ()

```
{  
    Area A1, A2;  
    int temp;  
    A1. GetLength ();  
    temp = A1. AreaCalculation ();  
    A1. DisplayArea(temp);  
    cout << endl << "Default Area when value is not taken from user"
```

```
temp = A2. AreaCalculation ( );
A2. DisplayArea (temp);
return 0;
}
```

- (d) Why will the function given in the following code fragment not work? What should be done to make it work?

```
int main ()
{
    float sum (float , float);

    :
}

void calc (void)
{
    float x, y, s;
    cin >> x >> y
    s = sum (x, y);
    :
}
```

Ans. // code fragment 1

```
void display(float x, float y);
{
    printf("Value of x %.2f",x);
    printf("Value of y %.2f",y);
}
```

// code fragment 2

```
{void display_report(int code, int number)}
{
    printf("%d",code);
    printf("%d",number);
}
```

// code fragment 3

```
float sum(float test1, float quiz1)
{
    float sum, average;
```

```
sum = test1 + quiz1;  
average = sum/2.0;  
return sum, average;  
}
```

- (e) **What is a Friend Function? What is the significance of friend functions?**

Ans. Refer to Chapter-4, Q.No.-1

- (f) **What do you mean by Static Data Members of a class? Explain the characteristics of static data members.**

Ans. Refer to Chapter-4, Page No.-65

A static member variable has certain special characteristics. They are:

- It is initialised to zero when the first object of its class is created. No other initialisation is permitted.
- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is visible only within the class, but its lifetime is the entire program.
- Static variables are normally used to maintain values common to the entire class.
- The type and the scope of each static member variable must be defined outside the class definition.
- Because, the static data members are stored separately rather than as a part of an object.
- They are also known as class variables.

- (g) **What do you understand by a Default Constructor? How is a default constructor equivalent to a constructor with default arguments?**

Ans. A default constructor is a constructor that either has no parameters, or if it has parameters, all the parameters have default values.

If no user-defined constructor exists for a class A and one is needed, the compiler implicitly declares a default parameter less constructor A::A(). This constructor is an inline public member of its class. The compiler will implicitly define A::A() when the compiler uses this constructor to create an object of type A. The constructor will have no constructor initializer and a null body.

Now, Refer to Chapter-2, Page No.-16

- (h) **What is Function Overloading? Compare default arguments with function overloading.**

Ans. Refer to Chapter-5, Page No.-83

- (i) When should one derive a class publicly or privately? Give a suitable example in support of your answer.**

Ans. Like the normal inheritance concept, various exception classes can be derived from a common base class. If a catch handler catches a pointer or reference to an exception object of a base-class type, it also can catch a pointer or reference to all objects of class publicly derived from that base class- this allows for polymorphic processing of related errors. Using inheritance with exceptions enables an exception handler to catch related errors with concise notation. One approach is to catch each type of pointer or reference to a derived-class exception object individually, but a more concise approach is to catch pointers or references to base-class exception objects instead. Also, catching pointers or references to derived-class exception object individually is error prone, especially if we forget to test explicitly for one or more of the derived-class pointer or reference types.

- (j) What are Iterators? List the five types of iterators supported by STL in C++.**

Ans. Iterators are used to access container class elements. They are called iterators because of their use in traversing the elements (from one to another) of a container class. In this sense they are quite similar to pointers. Iterators hold state information sensitive to the particular containers on which they operate, thus, iterators are implemented appropriately for each type of container. Certain iterator operations are uniform across containers. For example, ++ operation on an iterator moves it to the next element of the container. If iterator i points to a particular element, then, i++ points to the "next" element and *I refers to the element pointed by i.

There are five broad types of iterators supported by the STL. These are listed in the following table:

Table: Types of Iterators

Iterator	Access method	Movement	I/O Capability
Input	Linear	Forward only	Read only
Output	Linear	Forward only	Write only
Forward	Linear	Forward only	Read/Write
Bidirectional	Linear	Forward & Backward	Read/Write
Random	Random	Forward & Backward	Read/Write

Q2. (a) How does the functioning of a function differ when

- (i) an object is passed by value?**

- (ii) an object is passed by reference?**

Ans. Refer to Gullybaba.com "download section"

- (b) What is Operator Overloading? List the operators which cannot be overloaded. Give reasons behind it.**

Ans. Refer to Chapter-9, Q.No.-24 & Q.No.-19

- (c) What is 'this' Pointer? Explain the significance of 'this' pointer with the help of an example.

Ans. Refer to Chapter-9, Q.No.-5(a)

- Q3.** (a) What is the difference between call-by-value and call-by-reference in a user defined function in C++? Give an example to illustrate the difference.

Ans. Refer to Chapter-4, Page No.-64

- (b) What is Message Passing? Explain how message passing is used in C++ programming with example.

Ans. In object oriented languages, we can consider a running program under execution as a pool of objects where objects are created for 'interaction' and later destroyed when their job is over. This interaction is based on 'messages' which are sent from one object to another asking the recipient object to apply one of its own methods on itself and hence, forcing a change in its state. The objects are made to communicate or interact with each other with the help of a mechanism called message passing. The methods of any object may communicate with each other by sending and receiving messages in order to change the state of the object. An Object may communicate with other objects by sending and receiving messages to and from their methods in order to change either its own state or the state of other objects taking part in this communication or that of both. An object can both send and receive messages.

The messages are sent and received by passing various variables among specific methods using the signatures (a term that is not prevalent in common parlance) of the methods. Every method has a well defined and structured signature. The signature of a method is composed of: a) a type, associated with the variable whose value after execution of the method, would be returned to the object that would invoke the method; b) the types of a specific number of variables and the order associated with these variables whose values would be passed to the method before execution of the method starts. All these variables have a well defined format and corresponding values at any instant of time available for communication during the execution of the program. Following figure shows an example of a signature for a method 'evaluate'.

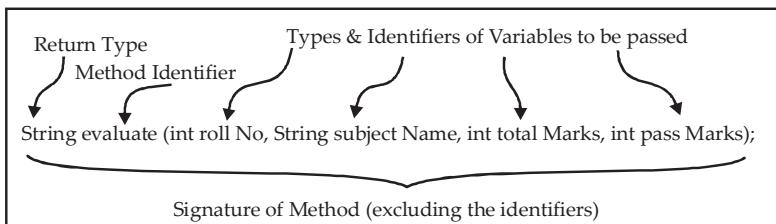


Fig: Signature of a Method

Q4. (a) What is the difference between overloading and overriding concepts in C++? Explain the usage of these concepts with suitable example code in C++?

Ans. Overloading is defining functions that have similar signatures, yet have different parameters.

Overriding is only pertinent to derived classes, where the parent class has defined a method and the derived class wishes to override that function.

Overriding	Overloading
Methods name and signatures must be same.	Having same method name with different signatures
Overriding is the concept of runtime polymorphism	Overloading is the concept of compile time polymorphism
When a function of base class is re-defined in the derived class called as overriding	Two functions having same name and return type, but with different type and /or number of arguments is called as overloading
It needs inheritance.	It doesn't need inheritance.
Method should have same data type.	Method can have different data types
Method should be public.	Method can be different access specifies

Example

Overriding

```
public class MyBaseClass
{
    public virtual void MyMethod()
    {
        Console.WriteLine("My BaseClass Method");
    }
}

public class MyDerivedClass : MyBaseClass
{
    public override void MyMethod()
    {
        Console.WriteLine("My DerivedClass Method");
    }
}
```

Overloading

```
int add(int a, int b)
int add(float a, float b)
```

Function Overriding

To cause late binding to occur for a particular function, C++ requires that we use the virtual keyword when declaring the function in the base class. Late binding occurs only with virtual functions, and only when we're using an address of the base class where those virtual functions exist, although they may also be defined in an earlier base class. To create a member function as virtual, we simply precede the declaration of the function with the keyword virtual. Only the declaration needs the virtual keyword, not the definition. If a function is declared as virtual in the base class, it is virtual in all the derived classes. The redefinition of a virtual function in a derived class is usually called function overriding. Notice that you are only required to declare a function virtual in the base class. All derived-class functions that match the signature of the base-class declaration will be called using the virtual mechanism. We can use the virtual keyword in the derived-class declarations (it does no harm to do so), but it is redundant and can be confusing.

Now, Refer to Chapter-9, Q.No.-12

- (b) What is an Exception in C++? Explain how exception handling is done in C++ with the help of a program. What will happen if exception is thrown outside of a try block? Give reasons for such a happening.**

Ans. Refer to Chapter-8, Page No.-161 - 165

- Q5. (a) Write a program in C++ to create a class Employee with basic data members such as name, address, age. Create a class Part_time employee which inherits from the Employee class. Part_time class should have a function to display the name, address and payment of the part-time employee.**

Ans. Refer to Chapter-6, Q.No.-9

- (b) Write a program in C++ to simulate the environment of a simple calculator.**

Ans. Following is a simple C++ program which is a menu-driven program based on simple calculation like addition, subtraction, multiplication and division according to user's choice:

```
/* C++ Program - Make Simple Calculator */  
#include<iostream.h>  
#include<stdio.h>  
#include<conio.h>  
#include<stdlib.h>  
void main()  
{
```

```
clrscr();
float a, b, res;
char choice, ch;
do
{
    cout<<"1.Addition\n";
    cout<<"2.Subtraction\n";
    cout<<"3.Multiplication\n";
    cout<<"4.Division\n";
    cout<<"5.Exit\n\n";
    cout<<"Enter Your Choice : ";
    cin>>choice;
    switch(choice)
        case '1' : cout<<"Enter two number : ";
                    cin>>a>>b;
                    res=a+b;
                    cout<<"Result = "<<res;
                    break;
        case '2' : cout<<"Enter two number : ";
                    cin>>a>>b;
                    res=a-b;
                    cout<<"Result = "<<res;
                    break;
        case '3' : cout<<"Enter two number : ";
                    cin>>a>>b;
                    res=a*b;
                    cout<<"Result = "<<res;
                    break;
        case '4' : cout<<"Enter two number : ";
                    cin>>a>>b;
                    res=a/b;
                    cout<<"Result = "<<res;
                    break;
    case '5' : exit(0);}
```

```
        break;
    default : cout<<"Wrong Choice..!!";
        break;
    }
cout<<"\n-----\n";
}while(choice!=5 && choice!=getchar());
getch();
}
```

When the above C++ program is compile and executed, it will produce the following result:

C:\TURBOC~1\Disk\TurboC3\SOURCE\999.EXE

1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Exit

Enter your Choice : 1

Enter two number : 4

6

Result = 10.000000

-
1. Addition
 2. Subtraction
 3. Multiplication
 4. Division
 5. Exit

Enter your Choice :

C++ Programming: BCS-031

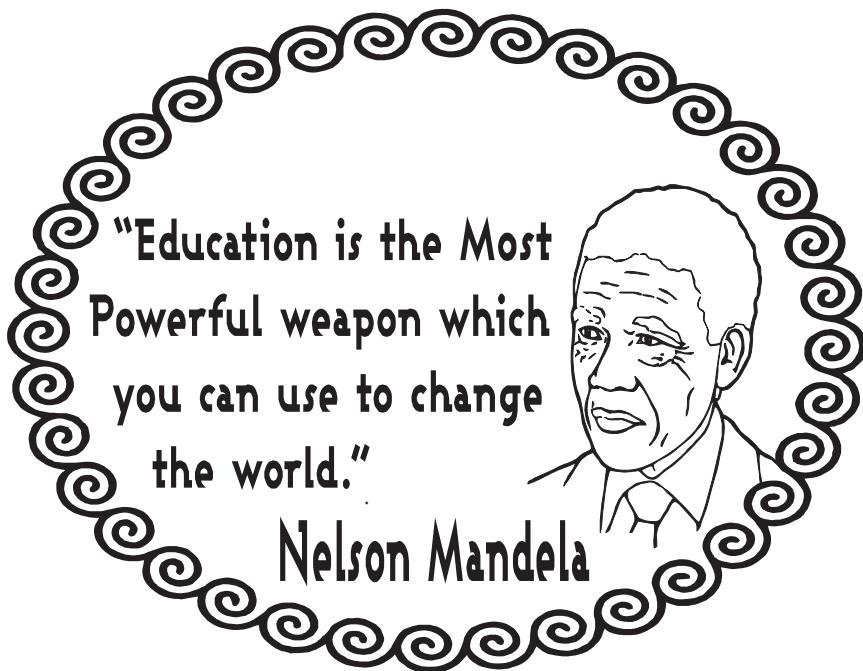
June, 2018

Note: Question number 1 is **compulsory** and carries 40 marks. Attempt any **three** questions from the rest.

- Q1.** (a) Define Abstraction and Encapsulation in object oriented programming. Explain class and object with the help of an example.
(b) What is polymorphism? Explain its advantages. Write a C++ program to demonstrate polymorphism.
(c) Write a C++ program to find the sum of two given numbers.
(d) What is a constructor? Explain the advantages of using constructors with the help of an example.
(e) What is a virtual function? Write a program in C++ to define a class "Teacher" with a virtual function "Salary". Derive class "Associate-Professor" from class teacher and implement the salary function.
(f) What is encapsulation? Are encapsulation and information hiding the same? Explain.
- Q2.** (a) Explain the different types of operators available in C++.
(b) What is inheritance? What are the different types of inheritance supported by C++? Explain how inheritance is implemented in C++.
- Q3.** (a) Write a C++ program to define class "Circle" and implement the following:
 - (i) a constructor
 - (ii) a member function to find area of a circle
 - (iii) a member function to find circumference of a circle
(b) How is function overloading implemented in C++? Explain with an example program.
- Q4.** (a) What is a friend function ? Explain how it is implemented in C++, with the help of an example program.

- (b) What is a template class? Create a template class for stack data structure.
- Q5. (a) What are , containers? Explain the use of List Container class with the help of an example.
- (b) What is an exception? How is it handled in C++? Explain it with the help of an example.
- (c) Explain the following with the help of an example:
- (i) Destructor
- (ii) Inline function

○○○



C++ Programming: BCS-031

December, 2018

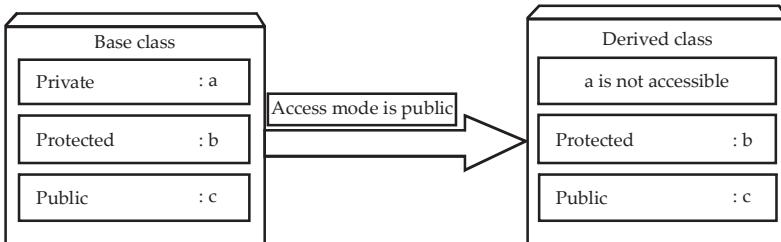
Note: Question number 1 is **compulsory** and carries 40 marks. Attempt any **three** questions from the rest.

Q1. (a) What is public access specifier? How is it different from private access specifier? Explain with the help of an example.

Ans. Public Access Specifier

If public access specifier is used while deriving class then the public data members of the base class becomes the public member of the derived class and protected members becomes the protected in the derived class but the private members of the base class are inaccessible.

Following block diagram explain how data members of base class are inherited when derived class access mode is public.



Sample program demonstrating public access specifier

```
// public access specifier.cpp
#include <iostream>
using namespace std;
```

```
class base
{
    private:
        int x;
    protected:
        int y;
```

```
public:  
int z;  
  
base() //constructor to initialize data members  
{  
    x = 1;  
    y = 2;  
    z = 3;  
}  
};  
class derive: public base  
{  
//y becomes protected and z becomes public members of  
class derive  
public:  
    void showdata()  
    {  
        cout << "x is not accessible" << endl;  
        cout << "value of y is " << y << endl;  
        cout << "value of z is " << z << endl;  
    }  
};  
int main()  
{  
    derive a; //object of derived class  
    a.showdata();  
    //a.x = 1; not valid : private member can't be accessed outside of class  
    //a.y = 2; not valid : y is now private member of derived class  
    //a.z = 3; not valid : z is also now a private member of derived class  
    return 0;  
}      //end of program
```

Output

x is not accessible
value of y is 2

value of z is 3

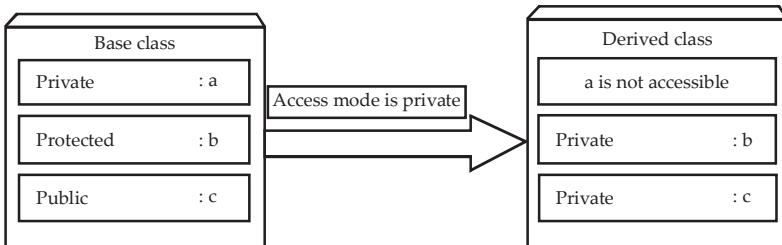
This is all about C++ access specifiers.

Private Access Specifier

If **private access specifier** is used while creating a class, then the public and protected data members of the base class become the private member of the derived class and private member of base class remains private.

In this case, the members of the base class can be used only within the derived class and cannot be accessed through the object of derived class whereas they can be accessed by creating a function in the derived class.

Following block diagram explain how data members of base class are inherited when derived class access mode is private.



Note: Declaring data members with private access specifier is known as data hiding.

Sample program demonstrating private access specifier:

```

// private access specifier.cpp
#include <iostream>
using namespace std;

class base
{
    private:
        int x;
    protected:
        int y;
    public:
        int z;
    base() //constructor to initialize data members
    {
        x = 1;
    }
}

```

```
y = 2;  
z = 3;  
}  
};  
  
class derive: private base  
{  
    //y and z becomes private members of class derive and x remains private  
    public:  
        void showdata()  
        {  
            cout << "x is not accessible" << endl;  
            cout << "value of y is " << y << endl;  
            cout << "value of z is " << z << endl;  
        }  
};  
int main()  
{  
    derive a; //object of derived class  
    a.showdata();  
    //a.x = 1; not valid : private member can't be accessed outside of class  
    //a.y = 2; not valid : y is now private member of derived class  
    //a.z = 3; not valid : z is also now a private member of derived class  
    return 0;  
} //end of program
```

Output

x is not accessible
value of y is 2
value of z is 3

Explanation:

When a class is derived from the base class with private access specifier the private members of the base class can't be accessed. So in above program, the derived class cannot access the member **x** which is private in the base class, however, derive class has access to the protected and public members of the base class.

Hence the function showdata in derived class can access the public and protected member of the base class.

- (b) **What do you mean by I/O formatting in C++ programming language? Explain with suitable examples.**

Ans. Refer to Chapter-9, Q.No.-54 (Pg. No.-235)

- (c) **What is static data member? Write the differences between the memory requirements of static data members and non-static data members.**

Ans. Static data members of class are those members which are shared by all the objects. Static data member has a single piece of storage, and is not available as separate copy with each object, like other non-static data members.

Difference: Whenever a class is instantiated, the memory is allocated for the created objects. Memory space for static data members is allocated only once during class declaration while memory space of on-static data members is allocated when objects of a class are created. Therefore, all objects of the class have access the same memory area allocated to static data members. When one of them modifies the static data member, the effect is visible to all the instance of the class i.e. objects.

- (d) **What do you mean by default constructor? Explain by taking examples.**

Ans. Refer to Dec-2017, Q.No.-1(g) (Pg. No.-287)

For instance, consider the following example:

```
class A {  
    int i;  
public:  
    void getval(void);  
    void prnval(void);  
    // member function definitions  
}  
A 0b 1;  
// uses default constructor for creating 0b1. Since user can use it,  
// that means, this implicitly defined default constructor is public  
// member of the class  
0b1. Getval();  
0b1. Getval();
```

- (e) What is inheritance? Write the types of inheritance supported by C++, with suitable illustrations.

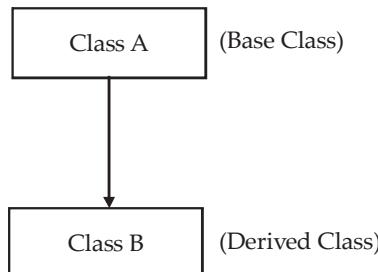
Ans. Refer to Chapter-6 (Pg. No.-89) (Inheritance)

Types of Inheritance

- Single Inheritance:
- Multiple Inheritance:
- Multi-level Inheritance
- Hierarchical Inheritance
- Hybird Inheritance
- Multi-path Inheritance

Types of Inheritance in C++

- **Single Inheritance:** In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.



Syntax:

```

class subclass_name : access_mode base_class
{
    //body of subclass
};

// C++ program to explain
// Single inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
public:
    Vehicle()
  
```

```
{  
    cout << "This is a Vehicle" << endl;  
}  
};  
  
// sub class derived from two base classes  
class Car: public Vehicle{  
  
};
```

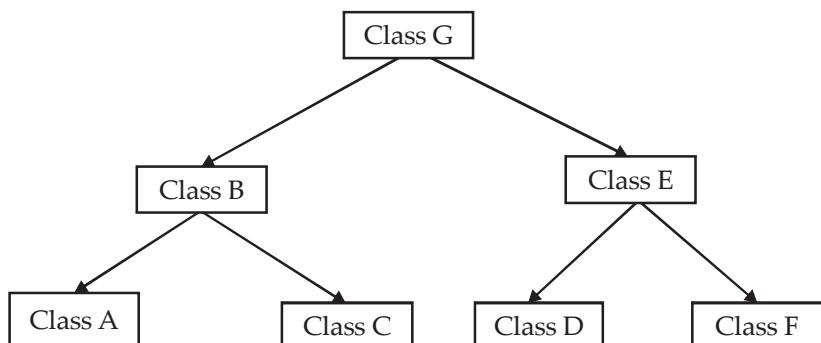
```
// main function  
int main()  
{  
    // creating object of sub class will  
    // invoke the constructor of base classes  
    Car obj;  
    return 0;  
}
```

Output:

This is a vehicle

Now, Refer to Chapter-6, (Pg. No.-95) (Multilevel Inheritance) and (Pg. No.-96) (Multiple Inheritance)

- **Hierarchical Inheritance:** In this type of inheritance, more than one sub class is inherited from a single base class. i.e. more than one derived class is created from a single base class.



```
// C++ program to implement
// Hierarchical Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle
{
public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};

// first sub class
class Car: public Vehicle
{

};

// second sub class
class Bus: public Vehicle
{

};

// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base class
    Car obj1;
    Bus obj2;
    return 0;
}
```

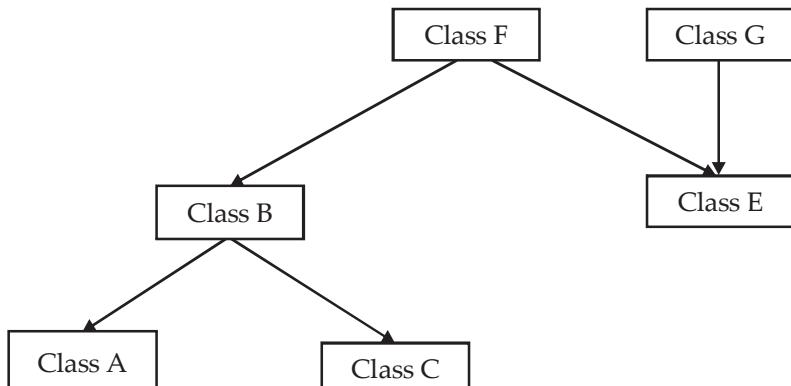
}

Output:

This is a Vehicle

This is a Vehicle

- **Hybrid (Virtual) Inheritance:** Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance. Below image shows the combination of hierarchical and multiple inheritance:



// C++ program for Hybrid Inheritance

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

// base class
class Vehicle
{
public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};

//base class
```

```
class Fare
{
public:
Fare()
{
    cout<<"Fare of Vehicle\n";
}
};

// first sub class
class Car: public Vehicle
{

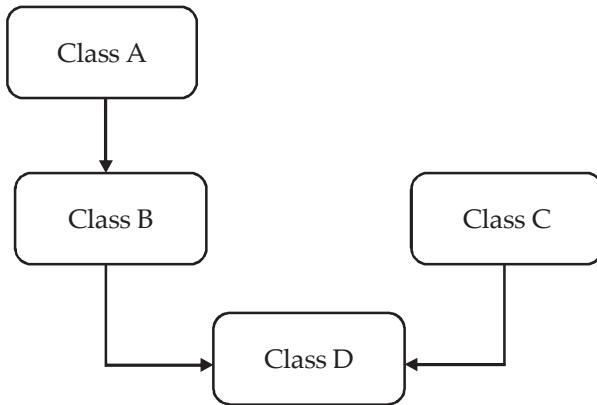
};

// second sub class
class Bus: public Vehicle, public Fare
{

};

// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base class
    Bus obj2;
    return 0;
}

Output:
This is a Vehicle
Fare of Vehicle
• Multipath Inheritance:
```



Multiple inheritance is a method of inheritance in which one derived class can inherit properties of base class in different paths. This inheritance is not supported in .NET Languages such as C#.

- (f) What is the purpose of `Push_back()`, `Push_front()`, `Pop_back()` and `Pop_front()` functions of a list?

Ans. `push_back()` – is used to insert an element at the back of a list
`push_front()` – is used to insert an element at the front of the list
`pop_back()` – deleting an element from the back of the list
`pop_front()` – deleting an element from the front of the list

- Q2. (a) Write appropriate statements to create a function template `Printarray` that can display the values contained in array passed as parameters to the function. The function should be able to accept integer, float and character arrays as arguments.

Ans. `template<typename T>`

```

void printarray(T a[], int n)
{
    for (int i=0; i<n-1; i++)
        cout << a[i] << " ";
}
  
```

- (b) Define exceptions and list any five common examples of exceptions.

Ans. The term exception itself implies an unusual condition. Exceptions are anomalies that may occur during execution of a program. Exceptions are not errors (syntactical or logical) but they still cause the program to misbehave. They are unusual conditions which are generally not expected by the programmer to occur. An exception in this sense is an indication of a problem that occurs during a programs s execution. The typical

exceptions may include conditions like divide by zero, access to an array outside its range, running out of memory etc.

List of five common examples: Insufficient memory to satisfy a new request, array subscript out of bounds, arithmetic overflow, division by zero, invalid function parameters.

(c) Write a C++ program to print 1234 right justified in a 10 digit field.

Ans. cout << setw (10) << 1234

Q3. (a) What are the data types in C++? Explain with examples.

Ans. Refer to Chapter-9, Q.No.-53 (Pg. No.-232)

(b) Explain the use of break statement with the help of an example.

Ans. Break Statement: The break statement is used to terminate the execution of the loop program. It terminates the loop in which it is written and transfers the control to the immediate next statement outside the loop. The break statement is normally used in the switch conditional statement. To understand, let us take the following C++ program:

Now, Refer to June-2016, Q.No.-2(a) (Pg. No.-268)

(c) Write a C++ program to generate Fibonacci series between 0 and 500.

Ans. C++ Program to Generate Fibonacci Series upto 1000

This C++ Program generates Fibonacci Series upto 1000. Here the first two numbers in the Fibonacci Series are 0 and 1 and the rest of the numbers are obtained by adding previous two numbers.

Here is source code of the C++ program which generates Fibonacci Series upto 1000. The C++ program is successfully compiled and run on a Linux system. The program output is also shown below.

```
1. /*
2. * C++ program to generate fibonacci series between 0 and 500
3. */
4. #include<iostream>
5. using namespace std;
6.
7. int main()
8. {
9.     int fib1 = 0, fib2 = 1, fib3 = 1;
10.
11.    cout << "The Fibonacci Series is follows : " << endl << fib1 << " "
12.      << fib2 << " ";
```

```
12.    while (fib1 + fib2 < 500)
13.    {
14.        fib3 = fib1 + fib2;
15.        fib1 = fib2;
16.        fib2 = fib3;
17.        cout << fib3 << " ";
18.    }
19.    cout << endl;
20.
21.    return 0;
22. }
```

\$ g++ main.cpp

\$./a.out

The Fibonacci Series is follows:

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377

Q4. (a) What are the differences between structures and classes in C++?

Ans. Refer to Page No.-21 [Structures] and Page No.-23 [Introduction to Classes]

(b) Write a C++ program to perform the following on two integer variables a and b:

- (i) a+b
- (ii) a - b
- (iii) a*b
- (iv) a/b

With proper provisioning for exceptions handling.

Ans. /* C++ Program – Addition, Subtraction, Multiplication, Division */

```
#include<iostream.h>
#include<conio.h>
void main ()
{
    clrscr ();
    int a, b, res;
    cout<<"Enter two number :";
    cin>>a>>b;
```

```
res=a+b;  
cout<<"\nAddition = "<<res;  
res=a-b;  
cout<<"\nSubtraction = "<<res;  
res=a*b;  
cout<<"\nMultiplication = "<<res;  
res=a/b;  
cout<<"\nDivision = "<<res;  
getch();  
}
```

Output:

Enter two number : 18, 6

Addition = 24

Subtraction = 12

Multiplication = 108

Division = 3

(c) What is an abstract class? Explain with an example. Also explain the features of an abstract class.

Ans. Refer to Chapter-6, (Pg. No.-100) (abstract classes)

Features of Abstract Class

- Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.
- Abstract class can have normal functions and variables along with a pure virtual function.
- Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.
- Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

Q5. (a) What do you mean by destructor? Explain with the help of examples.

Ans. Refer to Chapter-3, Page No.-27 [Destructors]

(b) What are the basic rules for operator overloading in C++? Explain briefly.

Ans. Refer to Chapter-9, Q.No.-46 (Pg. No.-223)

- (c) Explain in detail the importance of Virtual function and Pure Virtual function in the software development paradigm.

Ans. Refer to Chapter-9, Q.No.-27 (Pg. No.-204)

Pure Virtual Functions: It is normal practice to declare a function virtual inside the base class and redefine it in the derived classes. The function inside the base class is seldom used for performing any task. It only serves as a placeholder. For example, we have not defined any object of class media and therefore the function display () in the base class has been defined 'empty'. Such functions are called "do-nothing" functions.

A "do-nothing" function may be defined as follows:

```
Virtual void display () = 0;
```

Such functions are called pure virtual functions. A pure virtual function is a function declared in a base class that has no definition relative to the base class. In such cases, the compiler requires each derived class to either define the function or redeclare it as a pure virtual function. Remember that a class containing pure virtual functions cannot be used to declare any objects of its own. As stated earlier, such classes are called abstract base classes. The main objective of an abstract base class is to provide some traits to the derived classes and to create a base pointer required for achieving run time polymorphism.

Below Program demonstrates the use of pure virtual function:

Program Pure Virtual Function

```
#include <iostream>
using namespace std;
class Balagurusamy           //base class
{
public:
    virtual void example () = 0;      //Denotes pure virtual
Function Definition
};

class C:public Balagurusamy     //derived class 1
{
public :
    void example ()
    {
        cout<<"c text Book written by Balagurusamy";
    }
};
```

```
class oops : public Balagurusamy           //derived class 2
{
public:
    void example ()
    {
        cout<<"C++ text Book written by Balagurusamy";
    }
};

Void main ()
{
    Exforsys* arra [2];
```

```
C el;
oops e2;
arra [0] = &e1;
arra [1] = &e2;

arra[0] →example ( );
arra[1] →example ( );
}
```

The output of Program would be:

C text Book written by Balagurusamy
C++ text Book written by Balagurusamy

Its normal practice to declare a function virtual inside the base class and redefine it in the derived classes. The function inside the base class is seldom used for performing any task. It only serve as a placeholder. Such functions are called „do-nothing“ functions.

A „do-nothing“ function may be defined as follows:
virtual void show () = 0;

Such functions are called pure virtual functions.



C++ Programming: BCS-031

June, 2019

Note: Question number 1 is compulsory and carries 40 marks. Attempt any three questions from the rest.

- Q1.** What is object oriented programming paradigm? Explain advantages of object oriented programming paradigm over structured programming paradigm.
- (b) What is the need of memory management in C++? Explain the process of memory management in C++.
- (c) What is a virtual function? How does virtual function differ from function in C++? Explain with example.
- (d) What is operator overloading? Briefly explain general rules of operator overloading.
- (e) What is stream manipulator? Explain the use of `setw()` and `setprecision()` as stream manipulator.
- (f) What is an object in C++? Explain, how an object can be passed as an argument to a function, with the help of an example.
- (g) What is scope resolution operator? Explain the use of scope resolution operator with the help of a C++ program.
- (h) Explain any five relational operators in C++ with the help of examples.
- Q2.** (a) What do you understand by friend function? Write a C++ program, to find out the sum of n given numbers using friend function.
- (b) Explain the difference between public, private and protected, access specifiers with respect to classes in C++.
- (c) Differentiate between default constructor and parameterized constructor with the help of an example program in C++.
- Q3.** (a) What do you mean by operator-overloading in C++? List the operators which cannot be overloaded. Write a program in C++, to overload unary minus (–) operator.

- (b) What is exception handling? What are the keywords, used to handle the exception in C++? Write a C++ program to handle divide by zero exception.
- (c) How is structure different from a class? Explain with example.
- Q4. (a) What is static member? Explain the use of static data member and static member function, with the help of an example program in C++.
- (b) Explain the use of the following standard stream objects with the help of examples:
- (i) cin (ii) cout (iii) cerr (iv) clog
- (c) What is function template? Explain this concept, with the help of an example.
- Q5. Write short notes on the following (give example code in C++ for each):
- (a) Overriding concept in C++
- (b) Message passing
- (c) Encapsulation
- (d) Object initialisation and its need

○○○

C++ Programming: BCS-031

December, 2019

Note: (i) Question number 1 is **compulsory** and carries 40 marks. (ii) Attempt any **three** questions from the rest.

- Q1.** (a) Explain the use of 'Break' and 'Continue' statement in C++, with example program.
(b) What are Access control specifiers? Explain various types of access control specifiers.
(c) What is 'Copy constructor'? Explain it with the help of a suitable C++ program.
(d) Explain the concept of Friend function with suitable example code in C++.
(e) Compare while () and do-while () looping constructs with the help of suitable example for each.
(f) Write a program in C++, to find the largest of given three numbers by using a member function defined in a class.
(g) What is Object initialization? Why it is required, explain with the help of an example.
(h) Explain the usage of single inheritance and multiple inheritance.
- Q2.** (a) What is a Class template? How it is different from function template? Give example for each.
(b) What are Inline functions? Discuss their importance in programming. Write an example program in C++ to clarify the concept of Inline functions.
(c) What is Polymorphism? Give three advantages of polymorphism.
- Q3.** (a) Discuss the role of virtual functions in inheritance. What happens if we don't use virtual functions in inheritance? Give suitable example in support of your discussion.
(b) What are File stream operations? Write a program in C++ to demonstrate the file reading and writing operations.

- Q4.** (a) Explain the concept of parameter passing using call by value and call by reference with suitable examples.
(b) Compare Run time polymorphism and Compile time polymorphism. Give suitable example of each.
(c) What is destructor? Explain its use in C++ with the help of an example.
- Q5.** Write short notes on the following:
(a) Code Reusability (b) 'this' pointer
(c) Containers and its types in C++ (d) Stream manipulators

○○○

C++ Programming: BCS-031

June, 2020

Note: Question number 1 is compulsory and carries 40 marks. Attempt any three questions from the rest.

- Q1.** (a) Explain, how structured programming paradigm is different from object oriented paradigm.
(b) What is encapsulation? How is it different from information hiding? Explain with the help of an example.
(c) Explain the concept of copy constructor with the help of an example and program.
(d) What is access control specifier? Explain public access control specifier with example.
(e) Explain, how memory management is performed in C++.
(f) What is a stream manipulator? Explain the use of setw() and setprecision() as stream manipulator.
(g) What is operator overloading? Briefly discuss the general rules of operator overloading.
(h) What is scope resolution operator? Explain its use with the help of a C++ program.
- Q2.** (a) What is a container in C++? List main types of container in C++. Also list some common member functions of Container class.
(b) What is static member in C++? Explain the use of static data member and static member function with the help of an example and program in C++.
- Q3.** (a) What do you mean by polymorphism? How is runtime polymorphism different from compile time polymorphism? Give example(s) to support the above difference.
(b) What is a virtual function? How virtual function relates to inheritance? What happens if we don't use the virtual function in the inheritance? Explain with the help of an example and program.

- Q4.** (a) What is message passing? Demonstrate the utility of message passing with the help of an example code in C++.
- (b) Write a program in C++ to add two 3×3 matrices. Define proper class, constructor, destructor and methods in the program.
- Q5.** (a) Write a program in C++ to open an existing file and insert the text "My C++ file" at the end of it. Incorporate suitable comments, to improve the code readability.
- (b) What is a friend function? Explain two merits and two demerits of using friend functions, with the help of an example.

○○○

C++ Programming: BCS-031

December, 2020

Note: Question number 1 is compulsory and carries 40 marks. Attempt any three questions from the rest.

- Q1.** (a) What is construction? Explain the advantages of construction with the help of an example.
(b) What is function template? Write a function template SUM to add two numbers.
(c) List the merits and demerits of single inheritance over multiple inheritance.
(d) How is a structure in C++ different from a class? Explain with the help of example.
(e) What is an object in C++? Explain, how an object can be passed as an argument to a function with the help of an example.
(f) What is friend function? Explain its advantages with the help of an example.
(g) What is an inline function? Explain the advantages of inline function, with suitable example.
- Q2.** (a) What is exception? Explain, how exception handling is done in C++, with the help of a program. Also discuss, what will happen if an exception is thrown outside of a try block.
(b) What is template class? Explain the advantages of template class.
(c) Write a C++ program to find the average three given numbers. Define appropriate class and methods in the program.
- Q3.** (a) What is function overloading? How is it different from function overriding? Explain with an example program for each.
(b) What is virtual function? Write a program in C++ to create a class Doctor with a virtual function salary. Derive a class visiting Doctor and implement function salary in it.
- Q4.** (a) Explain, how function calls are matched in a C++ program in which functions are overloaded. Use appropriate example program for your explanation.

- (b) Write a C++ program to implement simple calculator to perform '+', '**'-**', '**'*'**', '**'/'**' on two operands. Your program should have methods for reading data and for performing arithmetic operations.

Q5. Write short notes on the following:

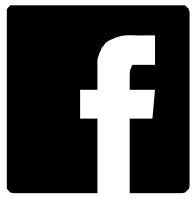
- (i) Copy constructor
- (ii) Access control specifier
- (iii) Stream manipulators
- (iv) Message passing
- (v) Scope resolution operator

○○○

ATTENTION IGNOU STUDENTS

Email at info@gullybaba.com
to Claim your FREE book

**"How to pass IGNOU exams
on time with Good Marks"**



/gphbooks

WE'D LOVE IT IF YOU'D LIKE US!

We're now on Facebook!

Like our page to stay on top of the useful,
greatest headlines & exciting rewards.



Our other awesome Social Handles:



gphbooks

For awesome &
informative videos
for IGNOU students



9350849407

Order now
through WhatsApp



gphbooks

We are
in pictures



gphbook

Words you get
empowered by



Why Students Choose GPH Books ?

-  Syllabus covered as prescribed by Universities/ Boards/Institutions.
-  Easily understandable language and format that help students prepare for exam in short period of time.
-  Published with exam-oriented approach, hence prepared in question-answer format which provides students the instant understanding of a correct answer.
-  Maximum solved previous year question papers included which help students to understand unique examination structure and equip them better for exam.
-  Both semesters' question papers (June-December) are included with solutions.
-  Instant updation of data as and when any change occurs.
-  Use of recycled paper.
-  Handy books and reasonable prices.
-  For every book sold, we contribute for society/institution/NGOs/underprivileged