Teaching Plan

BCACAC157: OBJECT ORIENTED PROGRAMMING USING C++

Faculty: Mrs. Lathika K.

Total hours:48H

UNIT-1 12 hrs

Session 1: **Principles of Object Oriented programming**: Basic Concepts, benefits, application.
Session 2: **Beginning with C++:** Program features, comments
Session 3**:** cin, cout, return statement
Session 4: Structure of a C++ program.
Session 5: **Tokens, expressions and control structures**: Tokens, keywords, identifiers
Session 6: basic and derived data types, symbolic constants, declaration of variables
Session 7: dynamic initialization of variables, reference variables
Session 8: The operators::, ::*, .*, delete, endl, new, setw.
Session 9: . Typecast operator, expression and implicit conversions, operator precedence.
Session 10: , control structures – while loop
Session 11: do-while loop
Session 12: if and switch statements.

UNIT-2 12 hrs

Session 1: **Functions in C++:** main function, Prototyping,
Session 2: call and return by reference, inline functions, default arguments
Session 3; arguments, function overloading
Session 4: mathematical functions
Session 5: **Classes and objects:** structures, specifying a class, creating objects,
Session 6: accessing class members, defining member functions
Session 7: making outside functions inline, nesting of member functions
Session 8: private member functions, arrays within a class, memory allocation for objects
Session 9: , static data members, static member functions
Session 10: arrays of objects, objects as function arguments
Session 11; friends functions, returning objects
Session 12 :const member functions, pointers to members.

UNIT-3

12hrs

Session 1: **Constructors and destructors**: Parameterized constructors
Session 2: multiple constructors, constructors with default arguments
Session 3: , dynamic initialization of objects, copy constructor
Session 4; dynamic constructors, constructing two dimensional arrays
Session 5: const object, destructors, memory allocation to an object using destructor
Session 6: **Operator overloading**: defining, overloading unary

Session 7: Overloading binary operators
Session 8: overloading binary operators using friend functions
Session 9: manipulation of strings using operator overloading
Session 10; rules for overloading operators,
Session 11; type conversions – basic to class, class to basic
Session 12;type conversion - one class to another class.

UNIT-4 **12hrs**

Session 1: **Inheritance**: Defining a derived class
Session 2: single inheritance, protected members
Session 3: multilevel inheritance, multiple inheritance
Session 4: hierarchical inheritance, hybrid inheritance
Session 5: virtual base classes, abstract classes
Session 6; constructors in derived classes, nesting of classes
Session 7: **Pointers**
Session 8: **virtual functions**
Session 9: **polymorphisms**: Pointers to objects, this pointer
Session 10: pointers to derived classes,
Session 11; virtual functions , pure virtual functions
Session 12: virtual constructors and destructors.

Text Book:
E Balagurusamy, Object Oriented Programming with C++, 5th Edition, Tata McGraw hill Publication.
Reference Books:
1. D Ravichandran, Programming with C++, Third Edition, McGraw hill 2011
2. Robert Lafore, Oriented Programming in C++, Galgotia Publications Pvt. Ltd, 2006.

Scheme of Evaluation:
The paper carries 100 marks out of which 80 marks will be allotted to external examination and 20 marks will be allotted to the internal assessment.
Internal assessment marks will be calculated as follows
1. Performance in 2 IA examinations will be converted out of 15 marks
2. Attendance 3 marks
3. Assignment 2 marks.
   Total 20 marks.

External examination marks will be as follows
1. 2 marks questions 10 out of 12 2 X 10 = 20 marks.
2. One full question out of 2 full questions in each unit carries 15 X 4 = 60 marks
   Total 80 marks.

OOPS concepts with C++		
UNIT-I		
Chapter 1	**Principles of Object Oriented Programming**	
1.1	Object oriented programming paradigm	
1.2	Basic concepts of object oriented programming	
1.2.1	Objects	
1.2.2	Classes	
1.2.3	Data abstraction and encapsulation	
1.2.4	Inheritance	
1.2.5	Polymorphism	
1.2.6	Dynamic binding	
1.2.7	Message passing	
1.3	Benefits of OOP	
1.4	Applications of OOP	
1.5	Assignment1	
Chapter 2	**Beginning With C++**	
2.1	What is C++?	
2.2	Applications of C++	
2.3	A Simple Program	
2.4	Preprocessor directive	
2.5	Header files	
2.6	Return type of main()	
2.7	Comments	
2.8	Variables	
2.9	Input operator	
2.10	Cascading I/O operators	
2.11	Structure of a C++ program	
2.12	Assignment 2	
Chapter 3	**Tokens, Expressions and Control Structures**	
3.1	Tokens	
3.2	Keywords	
3.3	Identifiers and constants	
3.4	Constants	
3.5	Basic data types	
3.6	Size and range of C++ basic data types	
3.7	User defined data types	
3.7.1	Structures and classes	
3.7.2	Enumerated Constants	
3.8	Derived data types	

UNIT-II

UNIT-I
CHAPTER 1
Principles of Object Oriented Programming

1.1 Object oriented programming paradigm

Some of the features of object oriented programming are:

1. Emphasis is on data rather than procedure.
2. Programs are divided into objects.
3. Data structures are designed such that they characterize the objects.
4. Functions that operate on data of an object are tied together in the data structure.
5. Data is hidden and cannot be accessed by external functions.
6. Objects may communicate with each other through functions
7. New data and functions can be added easily whenever necessary
8. It follows bottom up approach in program design.

1.2 Basic concepts of object oriented programming

Some of the concepts included in object oriented programming are:

- Objects
- classes
- data abstraction and encapsulation
- inheritance
- polymorphism
- dynamic binding
- message passing

1.2.1 Objects

Objects are the basic run time entities in an object oriented system. They may represent a person, a place, a bank account or any item that the program has to handle. They may also represent user defined data such as vectors, time and lists. When a program is executed, the objects interact by sending messages to one another. Each object contains data and code to manipulate data.

1.2.2 Classes

Objects are variables of type class. Once a class has been defined, we can create any number of objects belonging to that class. Each object is associated with the data of type class with which they are created. A class is thus a collection of objects of similar type. For example, mango,

apple and orange are the members of the class fruit. Classes are user defined data types and behave like the built in types of a programming language.

For example, cat frisky; will create an object frisky belonging to the class cat.

1.2.3 Data abstraction and encapsulation

The wrapping up of data and functions into a single unit (called class) is known as encapsulation. The data is not accessible to the outside world and only those functions which are inside the class can access it. This insulation of the data from direct access by the program is called data hiding or information hiding.

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concepts of abstraction and are defined as a list of abstract attributes such as size, weight or cost and the functions to operate on these attributes. The attributes are sometimes called data members because they hold information. The functions that operate on these data are called methods or member functions.

Since the classes use the concepts of data abstraction, they are known as abstract data types (ADT).

1.2.4 Inheritance

Inheritance is the process by which objects of one class acquire the properties fo objects of another class. It supports the concept of **hierarchical classification**. Fro example, the bird 'robin' is a part of the class 'flying bird' which is again a part of class 'bird'. The principle behind this sort of division is that each derived class share common characteristics with the class fro which it is derived.

In OOP, the concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes.

1.2.5 Polymorphism

Polymorphism is another important OOP concept. Polymorphism means that the ability to take more than one form. An operation may exhibit different behaviors in different instances. For example, consider the operation of addition. For 2 numbers, the operation will generate a sum. If the operands are strings, then the operation would generate a third string by concatenation. The process f making an operator to exhibit different behaviors in different instances is known as operator overloading.

Using a single function name to perform different types of tasks is known as function overloading.

1.2.6 Dynamic binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding also known as late binding means that the code associated with a given procedure call is not known until the time of the call at run time. It is associated with polymorphism and inheritance. a function call associated with a polymorphic reference depends on the dynamic type of reference.

1.2.7 Message passing

An object oriented program consists of a set of objects that communicate with each other. The process of programming in an object oriented language, therefore involves the following basic steps:

1. creating classes that define objects and their behavior
2. creating objects from class definitions
3. Establishing communication objects.

Objects communicate with each other by sending and receiving information. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving objects that generates the desired result. Message passing involves specifying the name of the object, the name of the function and the information to be sent.

1.3 Benefits of OOP

The principle advantages are

- Through inheritance, we can eliminate redundant code and extend the use of existing classes.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from the beginning. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure programs that can not be invaded by code in other parts of the program.
- It is possible to have multiple instances of an object to co-exist without any interference.
- It is possible to map objects in the problem domain in those in the program.
- It is easy to partition the work in a project based on objects.
- Object oriented systems can be easily upgraded from small to large systems.
- Software complexity can be easily managed.

1.4 Applications of OOP

The applications of OOP include

- Real time systems
- Simulation and modeling
- Object oriented databases
- Hypertext, hypermedia and expertext
- AI and expert systems
- Neural networks and parallel programming
- Decision support and office automation systems
- CIM/CAM/CAD systems

1.5 Assignment1

1) What is Object Oriented Programming?
2) Give any two difference between object oriented programming and procedural programming.
3) List any four features of Object Oriented Programming.
4) What is data encapsulation and Data abstraction?
5) How data hiding concept is implemented in C++?
6) What is Polymorphism? Give example.
7) What do you mean by message passing?
8) What is dynamic binding?
9) Mention any four advantages of OOPs.
10) Mention any four application areas of OOPs.

Chapter 2
Beginning with C++

2.1 What is C++?

C++ is an object oriented programming language. It was develop by **Bjarne stroustrup** at AT & T Bell labs in the early 1980's. He was an admirer of Simula67 and strong supporter of C and wanted to combine the best features of both the languages and create a more powerful language that could support object oriented programming features. The result was C++. There fore C++ is an extension of C with a major addition of class construct features of Simula67. Since the class was a major addition to the original C language, he initially called the new language ' C with classes'. However later in 1983, the name was changed to C++. The idea of C++ comes from the C increment operator C++, thereby suggesting that C++ is an incremented version of C.

2.2 Applications of C++

C++ is a versatile language for handling very large programs. It is suitable for virtually any programming task including development of editors, compilers, databases, communication systems and any complex real life application programs.

- Since C++ allows us to crate hierarchy related objects, we can build special object oriented libraries which can be used later by many programmers.
- While C++ is able to map the real world problem properly, the C part of C++gives the language the ability to get close to the machine level details.
- The C++ programs are easily maintainable and expandable. When a new feature needs to be implemented, it is very easy to add to the existing structure of an object.
- It is expected that C++ will replace C as a general purpose language in the near future.

2.3 A Simple Program

The simple program HELLO.CPP has many interesting parts.

HELLO.CPP demonstrates the parts of a C++ program.

```
#include <iostream.h>
void main()
{
        cout << "Hello World!\n";
}
0pp
Hello World!
```

The file iostream.h is included in the file. The first character is the # symbol, which is a signal to the preprocessor. Each time you start your compiler, the preprocessor is run. **include** is a preprocessor instruction that says, "What follows is a filename. The angle brackets around the filename tell the preprocessor to look in all the usual places for this file. The file **iostream.h** (Input-Output-Stream) is used by cout, which assists with writing to the screen.

Every C++ program has a main() function. In general, a function is a block of code that performs one or more actions. When your program starts, main() is called automatically.
main(), like all functions, must state what kind of value it will return. The return value type for main() in HELLO.CPP is void, which means that this function will not return any value at all.

All functions begin with an opening brace ({) and end with a closing brace. Everything between the opening and closing braces is considered a part of the function.

The object cout is used to print a message to the screen. These two objects, cout and cin, are used in C++ to print strings and values to the screen. A string is just a set of characters.
Here's how cout is used: type the word cout, followed by the output redirection operator (<<).

Whatever follows the output redirection operator is written to the screen. If you want a string of characters written, enclose them in double quotes ("). The final two characters, \n, tell cout to put a new line after the words Hello World.

2.4 Preprocessor directive

#include <iostream.h> this statement is not a program statement. It starts with a # and is called the preprocessor directive. Preprocessor directive is an instruction to the compiler. The preprocessor directive #include tells the compiler to insert another file into your sour file. The type of file usually included by #include is a header file.

2.5 Header files

In the above program, the preprocessor directive #include tells the compiler to add the header file **iostream** to Hellow.cpp file before compiling. **Iostream** is an example for a header file. It is concerned with basic input/output operations and contains declarations that are needed by the cout, cin and >> and << operators.

2.6 Return type of main()

In C++, main() returns an integer type value to the operating system. Therefore every main() in C++ should end with a return 0 statement. Otherwise a warning or an error might occur. Since main() returns an integer type value, return type for main()is specified as int. the following main() without type and return will run with a warning

```
main()
{
        …….
        …….
}
```

2.7 Comments

Comments are simply text that is ignored by the compiler, but that may inform the reader of what you are doing at any particular point in your program.

C++ comments come in two flavors: the double-slash (//) comment, and the slash-star (/*) comment. The double-slash comment, which will be referred to as a C++-style comment, tells the compiler to ignore everything that follows this comment, until the end of the line.

The slash-star comment mark tells the compiler to ignore everything that follows until it finds a star-slash (*/) comment mark. These marks will be referred to as C-style comments. Every /* must be matched with a closing */.

2.8 Variables

In C++ a variable is a place to store information. A variable is a location in your computer's memory in which you can store a value and from which you can later retrieve that value. For example **float number1, number2, sum, average;**

All variables must be declared before they are used.

2.9 Input operator

The statement cin >> number; is an input statement and causes the program to wait fro the user to type a number. The number typed is placed in the variable number1. Cin is a predefined object in C++ that corresponds to the standard input stream.

The operator >> is known as extraction operator or get from operator. It takes the value from the keyboard and assigns it to the variable on its right.

2.10 Cascading I/O operators

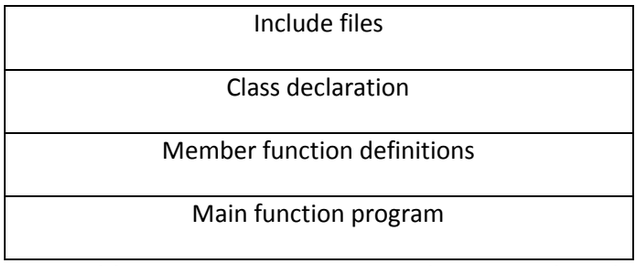
The statement cout << “Sum=” << sum << “\n”’ first sends the string “Sum=” to cout and then sends the value of sum. Finally it sends the newline character so that the next output will be in the new line. The multiple use of /,, in one statement is called cascading.

We can also cascade input operator >> as follows.

```
Cin>>number1>>number2;
```

The values are assigned from left to right. That is if we key in 2 values say 10 and 20, then 10 will be assigned to number 1 and 20will be assigned to number2.

2.11 Structure of a C++ program



A typical C++ program would contain 4 sections as shown in the above figure. The class declarations are placed in a header file and the definitions of the member functions go into another file. Finally the main program that uses the class is placed in the third file which includes the previous two files as well as any other file required.

2.12 Assignment2

1) Give syntax of **cin** and **cout** operators in C++.
2) What is a variable?
3) What is a comment? How many types of comments are available in C++?
4) Give the structure of a C++ program
5) What is cascading operators?

Chapter 3
Tokens, expressions and control structures

3.1 Tokens

The smallest individual units in a program ar known as tokens. C++ has the following tokens.

- Keywords
- Identifiers
- Constants
- Strings
- Operators

3.2 Keywords

These are reserved identifiers and can not be used as names for the program, variables or other user defined program elements. Some of the C++ keywords are given in the following table.

Asm	Double	New	Switch	Auto	Else	Operator	Twmplate
Break	Enum	Private	This	Case	Extern	protected	Throw
Catch	Float	Public	Register	Try	For	Friend	Class
Inline	Virtual	Delete	Const	Goto	Short	Union	while

3.3 Identifiers and constants

Identifiers refer to the names of the variables, functions, arrays, classes etc created by the programmer. Each language has its own rules for naming these identifiers. The rules are

1. Only alphabetic characters, digits and underscores are permitted.
2. The name cannot start with a digit
3. Upper and lower case letters are distinct
4. A declared keyword cannot be used as a variable name.

A major difference between C and C++ is the limit of the length of a name. While ANSI C recognizes only the first 32 characters in a name, ANSI C++ places no limits on its length and all characters in a name are significant.

3.4 Constants

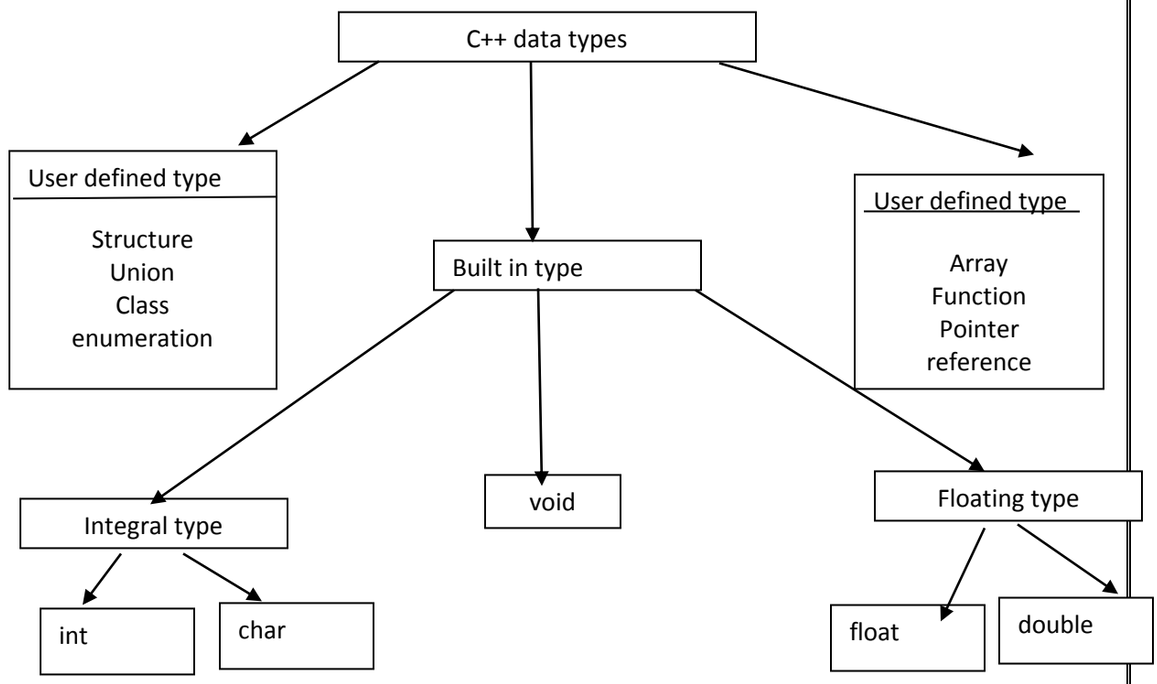
Refer to fixed values that do not change during the execution of the program.

Like C, C++ supports several kinds of literal constants. Thy include integers, characters, floating point numbers and strings. For example

```
123          //decimal integer
12.34        //floating point integer
O37          //octal integer
Ox2          //hexa decimal integer
"C++"        //string constant
'A'          //character constant
```

3.5 Basic data types

Data types in C++ can be classified under various categories as shown the figure.



3.6 Size and range of C++ basic data types

Type	Bytes	Range
Char	1	-128 to 127
Unsigned char		0 to 255
Signed char	11	-128 to 127
Int	2	-32768 to 32767

Unsigned int	2	0 to 65535
Signed int	2	-31768 to 32767
Short int	2	-31768 to 32767
Unsigned short int	2	0 to 65535
Sighned short int	2	-32768 to 32767
Long int	4	-2147483648 to 2146483647
signed long int	4	2147483648 to 2146483647
Unsigned long int	4	0 to 4294967295
Float	4	3.4E-38 to 3.4E+38
Double	8	1.7E -308 to 1.7E+308
Long double	10	3.4E -4932 to 1.1E+4932

Both C and C++ compilers support all the built in data types. With the exception of void, the basic data types may have several modifiers preceding them. The modifiers signed, unsigned, long and short may be applied to character and integer basic data types. However the moodier long may also be applied to double.

The type void was introduced in ANSI C. two normal uses of void are

1. To specify the return type of a function when it is not returning any value
2. To indicate an empty argument list to a function.

For example, **Void function1(void);**
Another use of void is in the declaration of generic pointers. For example,

Void *gp; //gp becomes the generic pointer

A generic pointer can be assigned a pointer value of any basic data type, but it may not be dereferenced. For example,

Int *ip; //int pointer

Gp=ip; //assign int pointer to void pointer

Are valid statements. But the statement,

***ip=*gp;** is illegal.

3.7 User defined data types

3.7.1 Structures and classes

We have used user defined data types such as **struct** and **union** in C. while these data types legal in C++, some more features have been added to make them suitable for object oriented programming. C++ also permits us to define another user defined data type known as **class,** which can be used to declare variables. The class variables are known as objects.

3.7.2 Enumerated Constants

Enumerated constants enable you to create new types and then to define variables of those types whose values are restricted to a set of possible values. For example, you can declare COLOR to be an enumeration, and you can define that there are five values for COLOR: RED, BLUE, GREEN, WHITE, and BLACK.

The syntax for enumerated constants is to write the keyword **enum**, followed by the type name, an open brace, each of the legal values separated by a comma, and finally a closing brace and a semicolon. Here's an example:

enum COLOR { RED, BLUE, GREEN, WHITE, BLACK };

This statement performs two tasks:

1. It makes `COLOR` the name of an enumeration, that is, a new type.
2. It makes `RED` a symbolic constant with the value `0`, `BLUE` a symbolic constant with the value `1`, `GREEN` a symbolic constant with the value `2`, and so forth.

Every enumerated constant has an integer value. If you don't specify the value, the first constant will have the value 0, and the next value is 1 and so on. If you write

enum Color { RED=100, BLUE, GREEN=500, WHITE, BLACK=700 };

Then RED will have the value 100; BLUE, the value 101; GREEN, the value 500; WHITE, the value 501; and BLACK, the value 700.

Demonstration of enumerated constants.

```
#include <iostream.h>
int main()
{
        enum Days { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };
        Days DayOff;
        int x;
```

```
        cout << "What day would you like off (0-6)? ";
        cin  >> x;
        DayOff = Days(x);
        if (DayOff == Sunday || DayOff == Saturday)
                cout << "\nYou're already off on weekends!\n";
        else
                cout << "\nOkay, I'll put in the vacation day.\n";
        return 0;
}
```

Output:

What day would you like off (0-6)? 1

Okay, I'll put in the vacation day.

What day would you like off (0-6)? 0

You're already off on weekends!

3.8 Derived data types

3.8.1 Arrays

The application of arrays in C++ is similar to that in C. the only exception is the way character arrays are initialized. When initializing a character array in ANSI C, the compiler will allow us to declare the array size as the exact length of the string constant. For example,

Char str[3]=”xyz”;

Is valid in ANSI C. but in C++, the size should be one larger than the number of characters in the string.

Char str[4]=”xyz”;

3.8.2 Functions

Functions have undergone major changes in C++. Some of these changes are simple. Many of these modifications and improvements were driven by the requirements of the object oriented concept of C++.

3.8.3 Pointers

Pointers are declared and initialized as in C. for example

Int *ip; //int pointer

Ip=&x; //address of x assigned to ip

*ip=10; //10 is assigned to x

C++ adds the concept of constant pointer and pointer to constant.

Char *const p1=”Good”; //constant pointer

We can not modify the address that p1 is initialized to.

Int const *p2=&m; //pointer to a constant

P2 is declared as pointer to constant. It can point to any variable of correct type, but the contents of what it points to cannot be changed.

We can also declare both the pointer and the variable as constants in the following way.

Const char * const cp=”xyz”;

This statement declares cp as a constant pointer to the string which has been declared a constant. In this case, neither the address assigned to the pointer cp nor the contents it points to can be changed

3.9 Symbolic constants

There are two ways of creating symbolic constants in C++

1. Using the qualifier const
2. Defining a set of integer constants using **enum** keyword.

In both C and C++, any value declared as **const** cannot be modified by the program in any way. However, there are some differences in implementation. In C++, we can use **const** in a constant expression, such as

Const int size=10;

Char name[size];

As with **long** and **short**, if we use the **const** modifier alone, it defaults to int. for example,

Const size=10; means const int size=10;

The named constants are just like variables except that their values cannot be changed.

In ANSI C, const values are global in nature. They are visible outside the file in which they are declared. However they can be made local by declaring them as static. To give a const values an external linkage so that it can be referenced form another file, we must explicitly define it as an extern in C++. Example

Extern const total=100;

Another method of naming integer constant is by enumeration as given below.

Enum{x,y,z};

This defines x,y,z as integer constant with value 0,1,and 2 respectively. This is equivalent to

Const x=0;

Const y=1;

Const z=2;

We can also assign values to x,y,z explicitly. Example

Enum{x=100, y=50, z=200};

3.10 Declaration of variables

In C, all variables must be declared before they are used in executable states. However, there is a significant difference between C and C++ with regard to the place of their declaration in the program. C requires all the variables to be defined at the beginning of a scope. When we read a C program, we usually come across a group of variable declarations at the beginning of each scope level.

C++ allows the declaration of a variable anywhere in the scope. This means that a variable can be declared right at the place of its first use. This makes the program mush easier to write and reduces the errors that may be caused by having to scan back and forth.

3.11 Dynamic initialization of variables

C++ allows initialization of the variables at runtime. This is referred to as dynamic initialization. For example,

…….
……
Int n=strrlen(string);
…….
Float area=3.142*rad*rad;

The above are valid initialization statement. Thus both the declaration and initialization of a variable can be done simultaneously at the place where the variable is used for the first time.

3.12 Reference variables

C++ uses a new kind of variable known as reference variable. A reference variable provides an alternative name for a previously defined variable. For example, if we make the variable **sum** a reference to the variable **total,** then sum and total can be used interchangeably to represent that variable. A reference variable is created as follows:

Datatype &referencename=variable name

Example:

```
Float total =100;
Float &sum=total;
```

Here sum is the alternative name declared to represent the variable total. Both the variables refer to the same data object in the memory. Now, the statements

cout << total; and cout <<sum;

Both print the value 100. The statement total=total+10; will change the value of both total and sum to 110. Similarly sum=0; will change the value of both variables to zero.

The following references are also used

```
Int x;
Int *p=&x;
Int &m=*p;
```

The above set of declarations causes m to refer to x which is pointed to by the pointer p.

3.13 Operators in C++

C++ has a rich set of operators. All C operators are valid in C++ also. C++ introduces some new operators. They are

>>	Extraction operator
<<	Insertion operator
::	scope resolution operator
->*	pointer to member operator
.*	pointer to member operator

Delete	memory release operator
Endl	line feed operator
New	memory allocation operator
Setw	field width operator

3.14 Memory management operators

C uses malloc() and calloc() functions to allocate memory dynamically at runtime. Similarly it uses free() to free dynamically allocated memory. C++ defines 2 unary operator new and delete that perform the task of allocating and freeing the memory in a better and easier way.

An object can b created by using new and destroyed by using delete. The new operator can be used to create objects of any typ. It takes the following general form:

Pointer-variable=new data type;

Here pointer-variable is a pointer of type data type. The new operator allocates sufficient memory to hold a data object of data type and returns the address of the object. The data type may b any valid data type. The pointer variable holds the address of the memory space allocated. For example,

P=new int;
Q=new float;

Where p is a pointer of type **int** and q is a pointer of type **float**. Alternatively, we can combine the declaration of pointers and their assignments as follows:

Int *p=new int;
Int *q=new float;

Similarly, the statements

*p=25;
*q=7.5;

Assign 25 to the newly created **int** object and 7.5 to **float** object.

When the data object is no longer needed, it is destroyed to release the memory space for reuse. The general form of it is

Delete pointer-variable;

The pointer variable is a pointer that points to a data object created with new. Example

```
delete p;
delete q;
```

3.15 Scope resolution operator

The scope resolution operator is used to uncover a hidden variable. It takes the following form:

:: variable_name;

This operator allows access to the global version of a variable. For example, ::count means the global version of the variable count.

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
int m=10;                                   // gloabl m
main()
{
        int m=20;                           //m redelared, local to main
        int k=m;
        {
                int m=30;                   // m redeclared again local to inner block

                cout << " we are in the inner block" << endl;
                cout << " k is" << k <<endl;
                cout << "m is"<< m <<endl;
                cout << "::m is " << ::m <<endl;
        }
                cout<< "we are i n the outer block" <<endl;
                cout << "m is " << m << endl;
                cout << "::m is " << ::m <<endl;
        getch();
}
```

Output

```
We are in inner block
K is 20
M is 30
::m is 10

We are in outer block
M is 20
::m is 10
```

In the above program, the variable m is declared at three places, namely, outside the main() function, inside the main() and inside the inner block. Note that ::m always refer to the global m. always ::m refers to the value 10 not 20

3.16 Type cast operator

C++ permits explicit type conversion of variables or expressions using the type cast operator. Traditional C casts are augmented in C++ by a function call notation as a syntactic alternative. The following 2 versions are equivalent.

(type-name) expression // c notation
Type-name (expression) //C++ notation

Examples:

Average=sum/(float)i; //c notation
Average=sum/flaot(i); //c++ notation

3.17 Manipulators

Manipulators are operators that are used to format the data display. The most commonly used manipulators are endl and setw.

3.17.1 The endl manipulator

Endl causes a linefeed to be inserted into the stream so that subsequent text is displayed on the next line. It has the same effect as that of '\n' character. It is an example of a manipulator. Manipulators are instructions to the output stream that modify the output in various ways.

3.17.2 The setw manipulator

The setw manipulator changes the field width of the output. The setw manipulator causes the number or string that follows it in the stream to be printed with a field n character wide where n is the argument to setw(n).

For example, **cout << setw(5) << sum << endl;**

The manipulator setw(5) specifies a field width 5 for printing the value of the variable sum. This value is right justified within the filed as shown below.

		3	4	5

The following example illustrates the use of setw and endl

```
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
void main()
{

        int basic=950, allowance=95, total=11045;
        clrscr();
                cout << setw(10) << "Basic" << setw(5) << basic << endl;
                cout<< setw(10) << "Allowance" << setw(5) << allowance << endl;
                cout<< setw(10) << "Total" << setw(5) << total << endl;
                getch();
        }
        return 0;
}
```

3.18 Expressions and their type

An expression is a combination of operators, constants and variables. It may also include function calls which return values. Expressions may be of the following seven types.

- Constant expressions
- Integral expressions
- Float expressions
- Pointer expressions
- Relational expressions
- Logical expressions
- Bitwise expressions

An expression may also use combinations of th above expressions. Such expressions are known as compound expressions.

3.18.1 Constant expressions

Constant expressions consists of only constant values. Examples are

15
20 + 5 / 2.0
‘x’

3.18.2 Integral expressions

Integral expressions are those which produce integer results after implementing all the automatic and explicit type conversions.

M
M*n-5
M?*'x'
5+int(2.0)

Where m and n are integer variables.

3.18.3 Float expressions

Float expressions are those which produce floating point results. Examples are

X+y
X+y/10
5+float (10)
10.75

Where x and y are floating point variables.

3.18.4 Pointer expressions

Pointer expressions produce address values. Examples are

&m
Ptr
Ptr+1

Where m is a variable and ptr is a pointer.

3.18.5 Relational expressions

Relational expressions produce results o f type **bool** which takes a value true or false. examples are

X<=y
A+b==c+d
M+n >100

When arithmetic expressions are used on either side of a relational operator, they will be evaluated first and then results are compared. Relational expressions are also known as Boolean expressions.

3.18.6 Logical expressions

Logical expressions combine two or more relational expressions and produces bool type values. Examples are

```
a>b &&x==10
x==10 ||y==5
```

3.18.7 Bitwise expressions

Bitwise expressions are used to manipulate data at bit level.thy are basically used for testing or shifting bits.

```
X<<3        //shift three it position to left
y>>1        //shift one bit position to right
```

3.18.8 Special assignment expressions

Chained assignment

```
X=(y=10);
Or
x=y=10
```

first 10 is assigned to y and then to x.

a chained statement can not be used to initialize variables at the time of declaration. For example, the statement **float a=b=12.34;** is invalid this may be written as **float a=12.34, b=12.34;**

Embedded assignment

X=(y=50)+10; where (y=50) is an assignment expression known as embedded assignment. Hr the value 50 is assigned to y and then the result 50+10=60 is assigned to x. this statement is similar to

```
Y=50;
X=y+10;
```

3.18.9 Compound assignment

Like C, C++ supports a compound assignment operator which is a combination of the assignment operator with a binary operator. For example, the simple assignment statement x=x+10 may be written as x+=10; the operator +=is known as compound assignment operator or short hand assignment operator.

The general form of the compound assignment operator is **Variable op=variable2;**
Where op is a binary arithmetic operator. This means that **varaible1=variable1 op variable2**

3.19 Implicit conversions

We can mix data types in expressions. For example, **m=5+2.75** is a valid statement. Whenever data types are mixed in an expression, C++ performs the conversions automatically. This process is known as implicit or automatic conversions.

When the compiler encounters an expression, it divides the expressions into sub expressions consisting of one operator or one or two operators. For example, if one of the operand is an int and other is a float, the int is converted into a float because a float is wider than an int.

3.20 Operator precedence

C++ enables us to add multiple meaning to the operators, yet their association and precedence remain the same. For example, the multiplication operator is higher precedence than the add operator. The following lists C++ operators and their meanings, precedence, associativity.

Operator	Associativity
::	Left to right
→, (), [] ,postfix ++, postfix --	Left to right
Prefix ++, prefix --, ~ ! , unary + , unary –,unary –, unary *, unary &, (type) sizeof ,new ,delete	Right to left
➔ **	Left to right
*/%	Left to right
+ -	Left to right
<< >>	Left to right
<<= >>=	Left to right
==, !=	Left to right
&	Left to right
^	Left to right
|	Left to right
&&	Left to right

||	Left to right
?:	Left to right
=*, +/, +%, +=	Left to right
<<=, >>=, +&, ^=, |=	Right to left
comma	Left to right

3.21 The if Statement

The if statement enables you to test for a condition (such as whether two variables are equal) and branch to different parts of your code, depending on the result.
The simplest form of an if statement is this:

```
if (expression)
{
        Statement block;
}
```

Where expression is any valid C++ expression. The expression must be enclosed in parentheses. If the expression evaluates to true, the statement will be executed. Otherwise ignored and control moves on to the next instruction in the program.

For example

```
If (a>b)
        A=a+2;
        B=b+2;
```

The above if statement specifies that when the expression a>b is true, execute the statement a=a+2, otherwise the control of the program should go to the statement immediately following it, that is b=a+2.
The code in the if body can consist of a single statement or a block of statement enclosed by braces.

```
if (expression)
{
   statement1;
   statement2;
   statement3;
}
```

Example

```
Main()
{
```

```
        Int x;
Cout << “enter a number” ;
Cin >> x;
If (x>100)
{
        Cout << “ the number” << x;
        Cout << “is greater than 100\n”;
}
Getch();
}
```

3.22 The if-else statement

The syntax is

```
If (expression)
{
        True block statements;
}
Else
{
        Flase block statements
}

Statement x
```

Here the **expression** is true, the true block statements are executed and then the control jumps to **statement-x** ignoring false block statements. If the **expression** is false, the false block statements are executed ignoring the true block statements and then the control jumps to statement-x

3.23 Nested if statements

A nested if which is within another **if** or **else**. An else statement always refers to the nearest if statement that is within the same block as the else and is not already associated with an **if.** The syntax is

```
If(condition 1)
{
        If (condition 2)
        {
                statement A;
        }
        else
        {
```

```
                Statement B;
        }
}
else
{
        Statement c;
}

Statement x;
```

If both conditions1 and 2 evaluates to true, only then statement A will be executed. If condition1 evaluates to false, then expression C will be executed and then control is transferred to statement x. I if condition1 evaluates to true and condition 2 evaluates to false, then expression B will be executed.

For example, consider the following segment of a program

```
If (a>b)
{
        If (c>d)
                X=y;
}
Else
X=z;
```

If the condition (a>b) is false, then the statement x=z will be executed otherwise it continues to perform the second test. If the condition (c>d) is true, then the statement x=y will be executed. Otherwise the control comes to the statement next to if..else body.

3.24 The Else..if ladder

There is another way of putting ifs together when multipath decisions are involved. A multipath decision is chain of ifs in which the statement associated with each else is an if. It takes the following general form:

```
If (condition 1)
        Statement 1;
Else if (condition 2)
        Statement 2;
Else if (condition 3)
        Statement 3;
Else if (condition n)
        Statement n;
Else
```

Default statement;
Statement x;

This construct is known as the **else if** ladder. The conditions are evaluated from the top to the downwards. As soon as a true condition is found, the statement associated with it is executed and the control is transferred to the statement-x skipping the rest of the ladder. When all the n condition conditions become false, then the final else containing the default statement will be executed.

3.25 The switch statement

The switch statement is an extension of the familiar if..else statement. The switch statement tests the value of an expression against a list of integer or character constants. When a match is found, the statement associated with the constant are executed.
The general form of the switch statement is:

```
Switch(expression)
{
        Case label 1:
                Statement sequence
                Break;

        Case label 2:
                Statement sequence
                Break;

        Case label 3:
                Statement sequence
                Break;

        Default:
                Default block
                Break;
}

Statement x;
```

When the switch statement is executed, the expression is evaluated first, and the value is compared with the case **label**(constant) value in the given order. If a **label** matches with the value of the expression, then the control is transferred directly to the group of statement following that **label**. If none of the **label** values with the value of the expression, the statement against **default** is executed. The **default** is optional in a **switch** statement. When it is not present and the value of the control expression does not match with the value of any of the

case **labels**, then no action will take place. In this case control is transferred to the statement that follows the **switch** construct.

The **break** statement at the end of each block signal the end of a particular **case** and causes an **exit** from the **switch** statement, transferring the control to the statement that follows the switch construct.

3.26 Decision making and looping

A segment of code that is executed repeatedly is called a loop. A program loop consists of 2 segmnets, one is body of the loop and other is known as the control statement. The control statement tests certain conditions and then directs the repeated execution of the statements contained in the body of the loop.

There are 2 types of loops depending on the position of the control statement in the loop. They are entry controlled and exit controlled.

In the entry controlled loop, the conditions are tested before the start of the loop execution,. If the condition is not satisfied, the body of the loop will not be executed. In case of exit controlled loop, the test is performed at the end of the body of the loop and therefore the body is executed unconditionally for the first time.

3.27 Loop control structures

There are 3 types of loops.

1) The while loop
2) The do..while loop
3) The for loop

3.27.1 The while loop (entry controlled loop)

The general form of the while loop is

```
While(test condition)
{
        Statement 1;
        Statement 2;
        Statement 3;
        ……….
}
```

The test condition may be any expression. The value of the test condition is evaluated. If the result is true, then the statements inside the loop are executed. Then the test condition is again evaluated. It it is again true, the statements are evaluated again. This process continues until

the test condition becomes false. when the test condition becomes false, the loop is terminated immediately and the control comes outside the loop and starts executing the statements outside the while loop.

For example,

```
While(i<=10)
        I=i+1;
```

Is same as

```
while(i<=10)
{
        I=i+1;
}
```

3.27.2 The do..while loop(exit controlled loop)

This structure is also called the “post tested” looping statement. In this structure the checking of a condition is done at the end. Here the set of statements in the structure is executed again and again until the test condition is true. If the test condition becomes false, control is transferred out of the structure.

The general form of the structure is

```
do
{
        Statement 1;
        Statement 2;
        Statement 3;
        ……
}
While (test condition);
        Statement x;
```

The execution of this loop structure works as follows.

1. The set of statements in the structure is first executed once.
2. The test condition is then evaluated.
3. If the value of the test condition is false, the do-while is terminated and the control goes out of the structure.
4. If the value of the test condition is true, the control goes back to the beginning of the structure and the statements in the structure is executed again.

3.27.3 For loop (entry controlled loop)

For statement is also called the fixed execution looping structure. This structure is normally used when we know exactly how many times a particular set of statements to be repeated again and again. The for loop can either be used as the increment looping statement or the decrement looping statement. The general form of the for statement is

```
for(expression1; expression2; expression3)
{
        Statement 1;
        Statement 2;
        Statement 3;
        ……….
}
Statement n;
```

Where expression1 represents the initialization expression. Expression2 represents the expression for the final condition.expression3 represents the increment or decrement expression.
The execution of the for loop is as follows.

1. Expression1 is first evaluated, i.e the counter variable of the expression is assigned the initial value.
2. Expression2 is then executed i.e the value of the counter variable is checked to see whether it has exceeded the final value. If not, the statements in the block are executed once.
3. Control is sent back to the beginning of the block and expression3 is evaluated. That is the value of the counter variable is either incremented or decremented depending on the statements used.
4. Step 2 is repeated again and again until the counter variable exceeds the final value.

3.28 Jump statements

The jump statements unconditionally transfer program control within a function. There are 4 statements that perform an unconditional branching. They are goto, return, break and continue. We can use gooto and return statements anywhere in the program, where as break and continue are used inside the loops.

3.28.1 The goto statement

The goto statement is a simple statement used to transfer control from one point in a program to any other point in that program. The syntax is

goto label:
Statement 1;
Statement 2;
…….
Label: statement n;

Where label is a user defined identifier and can appear either before or after goto. This statement provides an unconditional jump to the statement indicated by the label. No declaration is required for the layer. The syntax of the label requires a colon(:) after the label.

Write a program to print first 10 numbers using goto statement

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

int main()
{
int a;
clrscr();
a=1;
inc:
        cout << a;
        a=a+1;
        if(a<=10) goto inc;
                return 0;
}
```

3.28.2 The break statement

The break statement has two uses. It can be used to terminate a case in the switch statement and/or to terminate loops. When the break statement is encountered inside a loop, the loop is immediately terminated and the program control passes to the statement following the loop. The break statement can be used in any C loop i.e while, do..while and for loop.

While(condition)
{
…….
…….
If(condition)
Break;
…….
…….
}

……

Write a program to illustrate the use of break statement

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
main()
{
        int i,n;
        clrscr();
        printf("Enter the number\n");
        scanf("%d",&n);
        if(n==2)
        {
                printf("prime");
                getch();
                return 0;
        }

        for(i=2; i<=sqrt(n); i++)
        {
                if(n%i==0)
                {
                        printf("not prime");
                        break;

                }
        }
        if(n%i!=0)
                printf("prime");
        getch();

}
```

3.28.3 The continue statement

In some programming situations we want to take the control back to the beginning of the loop. This can be done with the help of a continue statement. When the keyword continue is encountered inside any lop, control automatically passes to the beginning of the loop. The syntax is **continue;**

```
While(condition)
{
        Statement1;
```

```
        Statement2;
        If (condition)
                Continue;
                        Statement3;
                        Statement4;
}
```

Write a program to generate numbers which are divisible by 3 in the range of first 20 numbers using continue statement

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

int main()
{
        int n=20,i;
        clrscr();
        for(i=1; i<=n; i++)
        {
                if (i%3!=0)
                        continue;
        else
                        cout <<  i;
        }
        getch();
         return 0;
}
```

3.29 Assignment 3

1. Define i) Token ii) Identifier
2. What are the rules for declaring identifier?
3. Differentiate Basic and Derived data types.
4. What is user defined data type? Give two examples.
5. What are the uses of void data type in C++ ?
6. Give two methods of declaring symbolic constants in C++.
7. What do you mean by pointer constant? Give example.
8. What is pointer to the constant? Give example.
9. What do you mean by dynamic initialisation of variables in C++? Give example
10. What is reference variable?
11. What is scope resolution operator?
12. List any four operators in C++ not found in C.
13. How to allocate memory to pointer using new operator?

14. What are manipulators? Give example.
15. Give syntax and usage of setw and endl manipulators.
16. What is implicit type conversion?
17. What is operator precedence and associatively of operators.
18. Give the syntax of switch statement.
19. Compare while and do..while loops.
20. Explain increment and decrement operators with example
21. Write a note on type conversions in C++.
22. Explain precedence and associatively of the operators with reference following expression a+b*c/d-(e+f)/d
23. Explain for loop structure with example
24. Explain the use of break and continue statements in C++

UNIT-II

Chapter 4
Functions in C++

4.1 What Is a Function?

A function is a subprogram that can act on data and return a value. Every C++ program has at least one function, main(). When your program starts, main() is called automatically. main() might call other functions, some of which might call still others.

Each function has its own name, When a program calls a function, execution jumps to the function and then continues at the line after the function call. When the function returns, execution continues on the next line of the calling function.

Functions come in two varieties: user-defined and built-in. Built-in functions are part of your compiler package--they are supplied by the manufacturer for your use.

4.2 The main function

C does not specify any return type for the main() function which is the starting point for the execution of a program. The definition of main() looks like this.

```
main()
{
        // main program statements
}
```

This is valid because the main() in C does not return any value.

In C++, the main() returns a value of type into to the operating system. Therefore in C++, main must be declared as **int main()**

The functions that have return value should use the return statement for termination. The main() in C++ is defined as follows.

```
Int main()
{
        ……..
        ……..
        ……
        Retuns 0;
}
```

4.3 Function prototyping

Function prototyping is one of the major improvements added to C++ functions. The declaration of a function before it is used is called as **function prototype.** The prototype describes the function interface to the compiler by giving details such as the number and type of arguments and type of return values. With function prototyping, a template is always used when declaring and defining a function. When the function is called, the compiler uses the template to ensure that proper arguments are passed and the return value is treated correctly.

Function prototype is a declaration statement in the calling program and has the following form.

Type function_name (argument list);

The argument list contains the types and names of the arguments that must be passed to the function.

For example, **float volume(int x, float y, float z);**

In a function declaration, the names of the arguments are dummy variables and therefore they are optional. i.e **float volume(int , float , float);** is acceptable at the place of declaration. In general we can either include or exclude the variable names in the argument list of prototypes..

In the function definition, names are required because the arguments must be referenced inside the function. For example,

```
Float volume(int a, float b, float c)
{
        Flaot y=a*b*c;
        …
        …
}
```

4.4 Declaring and Defining Functions

Using functions in your program requires that you first declare the function and that you then define the function. The declaration tells the compiler the name, return type, and parameters of the function. The definition tells the compiler how the function works. No function can be called from any other function that hasn't first been declared. The declaration of a function is called its prototype.

The general form of the function is as follows

```
Return_type  function_name(type arg1, type arg2..)
{
        Local variable declarations;
        Statement1;
        Statement2;
        ….
        …..
        Statement n;
        Return expression;
}
```

The various components of the function are

1. **Return type:** the **return_type** specifies the type of value which will be sent back after the function has performed its task. The **return_type** can include the normal data types such as int, float or char etc. the data type void is used if the function does not return a value.
2. **Function name:** a single program may have any number of functions included in it. The function_name thus helps to uniquely identify and call a function.
3. **Argument list with declaration:** the argument list identifies a set of values which are to be passed to the function from either the main program or a subprogram. The argument list should be declared like other variables in the program.
4. **Body of the function:** this includes the identification of those components which will be used only within the function and not outside it. Thus those components are declared as local variables. A function contains a set of executable statements which will perform the required task for the user. The last executable statement of the function is **return** with an **expression** which contains the value that has to be sent back to calling program.

4.5 Call by value

The arguments passed in to the function are local to the function. Changes made to the arguments do not affect the values in the calling function. This is known as passing by value, which means a local copy of each argument is made in the function. These local copies are treated just like any other local variables. Listing 5.5 illustrates this point.

A demonstration of passing by value.

```
#include <iostream.h>
void swap(int x, int y);
int main()
{
        int x = 5, y = 10;
          cout << "Main. Before swap, x: " << x << " y: " << y << "\n";
        swap(x,y);
```

```
        cout << "Main. After swap, x: " << x << " y: " << y << "\n";
        return 0;
    }

    void swap (int x, int y)
    {
        int temp;
        cout << "Swap. Before swap, x: " << x << " y: " << y << "\n";
        temp = x;
        x = y;
        y = temp;
        cout << "Swap. After swap, x: " << x << " y: " << y << "\n";
    }
```

Output: Main. Before swap, x: 5 y: 10
Swap. Before swap, x: 5 y: 10
Swap. After swap, x: 10 y: 5
Main. After swap, x: 5 y: 10

4.6 Call by reference

Provision of the reference variables in C++ permits us to pass parameters to the functions by reference. When we pass arguments by reference, the formal parameters in the called function become aliases to the actual arguments in the calling functions. Consider the following function

```
Void swap (int &a, int &b)
{
    Int temp=a;
    A=b;
    B=temp;
}
```

Now if m and n are 2 integer variables then the function call swap(m,n) will exchange the values of m and n using their reference variables a and b.

```
#include <iostream.h>
void swap(int &x, int &y);
int main()
{
    int x = 5, y = 10;
    cout << "Main. Before swap, x: " << x << " y: " << y << "\n";
    swap(x,y);
    cout << "Main. After swap, x: " << x << " y: " << y << "\n";
    return 0;
```

```
    }

 void swap (int &x, int &y)
  {
            int temp;
             cout << "Swap. Before swap, x: " << x << " y: " << y << "\n";
            temp = x;
            x = y;
            y = temp;
            cout << "Swap. After swap, x: " << x << " y: " << y << "\n";
}
```

Output: Main. Before swap, x: 5 y: 10
Swap. Before swap, x: 5 y: 10
Swap. After swap, x: 10 y: 5
Main. After swap, x: 10 y: 5

4.7 Return Values

Functions return a value or return void. Void is a signal to the compiler that no value will be returned. To return a value from a function, write the keyword return followed by the value you want to return. The value might itself be an expression that returns a value. For example:

```
return 5;
return (x > 5);
return (MyFunction());
```

These are all legal return statements, assuming that the function MyFunction() itself returns a value. The value in the second statement, return (x > 5), will be zero if x is not greater than 5, or it will be 1. What is returned is the value of the expression, 0 (false) or 1 (true), not the value of x.

When the return keyword is encountered, the expression following return is returned as the value of the function. Program execution returns immediately to the calling function, and any statements following the return are not executed.

It is legal to have more than one `return` statement in a single function. Listing 5.6 illustrates this idea.

4.8 Default Parameters

C++ allows us to call a function without specifying all its arguments. In such cases, the function assigns a default value to a parameter which does not have a matching argument in the function call. Default values are specified when a function is declared.

For example, float amount(float p, int p, r=0.15);

The above prototype declares a default value of 0.15 to the argument rate.
A function call like **value=amount(5000,7);** passes the value 5000 to p and 7 to t and then lets the function use default value of 0.15 to r.

The function call **value=amount(5000,7, 0.12); passes** an explicit value of 0.12 to r.

Default arguments are useful in situations where some arguments always have the same value. For example, the bank interest may remain the same for all customers for a particular period of deposit. One important point to note that only the trailing arguments can have default values and therefore we must add defaults from **right to left**. We cannot provide a default value to a particular argument in the middle of an argument list. Some examples are

```
Int mul(int I, int j=5, int k=10);          //valid
Int mul(int i=5, int j)              //invalid
Int mul(int i=0, int j, int k=10);          //invalid
Int mul(int i=2, int j=5, int k=10); //valid
```

A demonstration of default parameter values.

```
#include <iostream.h>
int AreaCube(int length, int width = 25, int height = 1);
int main()
{
          int length = 100;
          int width = 50;
          int height = 2;
          int area;
          area = AreaCube(length, width, height);
          cout << "First area equals: " << area << "\n";
          area = AreaCube(length, width);
          cout << "Second time area equals: " << area << "\n";
          area = AreaCube(length);
          cout << "Third time area equals: " << area << "\n";
          return 0;
}

AreaCube(int length, int width, int height)
  {
          return (length * width * height);
  }
```

Output: First area equals: 10000
Second time area equals: 5000
Third time area equals: 2500

4.9 Const arguments

In C++, an argument to a function can be declared as const as shown below.

Int strlen(const chat *p);
Int length(const string &s);

The qualifier const tells the compiler that the function should not modify the argument. The compiler ill generate an error when this condition is violated. This type of declaration is significant only when we pass arguments by reference or pointers.

4.10 Inline Functions

When you define a function, normally the compiler creates just one set of instructions in memory. When you call the function, execution of the program jumps to those instructions, and when the function returns, execution jumps back to the next line in the calling function. If you call the function 10 times, your program jumps to the same set of instructions each time. This means there is only one copy of the function, not 10.

There is some performance overhead in jumping in and out of functions. It turns out that some functions are very small, just a line or two of code, and some efficiency can be gained if the program can avoid making these jumps just to execute one or two instructions. When programmers speak of efficiency, they usually mean speed: the program runs faster if the function call can be avoided.

If a function is declared with the keyword inline, the compiler does not create a real function: it copies the code from the inline function directly into the calling function. No jump is made; it is just as if you had written the statements of the function right into the calling function.

The inline functions are defined as follows

```
Inline function_header
{
        Function body
}
```

Example:

```
Inline double cube(double a)
```

```
{
        Return (a*a*a);
}
```

The above inline function can be invoked by statements like

```
C=cube(3.0);
D=cube(2.5 + 1.5);
```

On the execution of these statements, the values of c and d
Note that inline functions can bring a heavy cost. If the function is called 10 times, the inline code is copied into the calling functions each of those 10 times. This increases the size of the executable code.

If you have a small function, one or two statements, it is good to use as inline. Listing 5.9 demonstrates an inline function.

Demonstration of an inline function.

```
#include <iostream.h>
 inline int Double(int);
 int main()
{
         int target;
        cout << "Enter a number to work with: ";
        cin >> target;
        cout << "\n";
        target = Double(target);
        cout << "Target: " << target << endl;
        target = Double(target);
        cout << "Target: " << target << endl;
        target = Double(target);
        cout << "Target: " << target << endl;
        return 0;
}

int Double(int target)
 {
        return 2*target;
 }
```

Output: Enter a number to work with: 20

Target: 40
Target: 80

Target: 160

4.11 Overloading Functions

C++ enables you to create more than one function with the same name. This is called function overloading. This means that we can se the same function name to cerate functions that perform a variety of different tasks. This is known as function polymorphism in OOP. The functions must differ in their parameter list, with a different type of parameter, a different number of parameters, or both. Here's an example:

```
int myFunction (int, int);
int myFunction (long, long);
int myFunction (long);
```

myFunction() is overloaded with three different parameter lists. The first and second versions differ in the types of the parameters, and the third differs in the number of parameters.

The return types can be the same or different on overloaded functions. You should note that two functions with the same name and parameter list, but different return types, generate a compiler error.

Function overloading is also called function *polymorphism*. Poly means many, and morph means form: a polymorphic function is many-formed.

Function polymorphism refers to the ability to "overload" a function with more than one meaning. By changing the number or type of the parameters, you can give two or more functions the same function name, and the right one will be called by matching the parameters used.

Suppose you write a function that doubles whatever input you give it. You would like to be able to pass in an int, a long, a float, or a double. Without function overloading, you would have to create four function names:

```
int DoubleInt(int);
long DoubleLong(long);
float DoubleFloat(float);
double DoubleDouble(double);
```

With function overloading, you make this declaration:

```
int Double(int);
long Double(long);
float Double(float);
double Double(double);
```

Example: function overloading

```
#include <iostream.h>
#include<conio.h>
 int twice(int);
 long twice(long);
 float twice(float);

int main()
{
          clrscr();
        int     myInt = 6500;
        long    myLong = 65000;
        float   myFloat = 6.5F;

         int     dInt;
         long    dLong;
         float   dFloat;

        cout << "myInt: " << myInt << "\n";
         cout << "myLong: " << myLong << "\n";
        cout << "myFloat: " << myFloat << "\n";

         dInt = twice(myInt);
        dLong = twice(myLong);
        dFloat = twice(myFloat);

        cout << "doubledInt: " << dInt << "\n";
        cout << "doubledLong: " << dLong << "\n";
         cout << "doubledFloat: " << dFloat << "\n";

        return 0;
}

int twice(int myInt)
{
        return 2 * myInt;
}

long twice(long dLong)
{
        return 2 * dLong;
}
```

```
float twice(float dFloat)
{
        return 2 * dFloat;
}
```

Output:
myInt: 6500
myLong: 65000
myFloat: 6.5

DoubledInt: 13000
DoubledLong: 130000
DoubledFloat: 13

4.12 Mathematical functions

The standard C++ supports many math functions that can be used for performing certain commonly used calculations. Most frequently used math functions are given the table below.

Ceil(x)	Rounds x to smallest integer not less than x. for example, ceil(8.1)=9.0 and ceil(-8.8)=-8.0
Cos(x)	Trigonometric cosine of x
Exp(x)	Exponential function e^x
Fabs(x)	Absolute value of x. if x>0 then abs(x)=x If x=0 then abs(x)=0.0 If x<0 then abs(x)=-x
Floor(x)	Rounds x to the largest integer not greater than x. floor(8.2)=8.0 and floor(-8.8)=-9.0
Log(x)	natural logarithm of x
Log 10(x)	Logarithm of x(base 10)
Pow(x,y)	X raised to power y(i.e x^y)
Sin(x)	Trigonometric sine of x
Sqrt(x)	Square root of x
Tan(x)	Trigonometric tangent of x

4.13 Assignment 4

1. What is function prototype ? Give example.
2. What is inline function? Explain with an example
3. Give two situations in which inline function may not work.
4. What is default argument? When it is needed ?
5. What do you mean by pass by reference? Explain with example
6. What do you mean by pass by value? Explain with example
7. What is constant argument? Give example
8. Explain the concept of overloading a function in C++ with example. 4
9. Explain concept of default arguments with Example. Also mention the rules to be followed while assigning default values.

Chapter 5
Classes and objects

5.1 Classes and Members

You make a new type by declaring a class. A class is just a collection of variables--often of different types--combined with a set of related functions.

One way to think about a car is as a collection of wheels, doors, seats, windows, and so forth. Another way is to think about what a car can do: It can move, speed up, slow down, stop, park, and so on. A class enables you to encapsulate, or bundle, these various parts and various functions into one collection, which is called an object.

Member variables: also known as **data members** , are the variables in your class. Member variables are part of your class, just like the wheels and engine are part of your car.

Member functions: The functions in the class typically manipulate the member variables. They are referred to as **member functions** or **methods** of the class. Methods of the Car class might include Start() and Brake(). A Cat class might have data members that represent age and weight; its methods might include Sleep(), Meow(), and ChaseMice().

5.2 Declaring a Class

To declare a class, use the class keyword followed by an opening brace, and then list the data members and methods of that class. End the declaration with a closing brace and a semicolon. Here's the declaration of a class called Cat:

```
class Cat
{
        int  Age;
        int  Weight;
        Meow();
};
```

Declaring this class doesn't allocate memory for a Cat. It just tells the compiler what a Cat is, what data it contains (Age and Weight), and what it can do (Meow()). It also tells the compiler how big a Cat is--that is, how much room the compiler must set aside for each Cat that you create. In this example, if an integer is two bytes, a Cat is only four bytes big: **Age** is two bytes, and **Weight** is another two bytes. Meow() takes up no room, because no storage space is set aside for member functions (methods).

5.3 Defining an Object

An object is an individual instance of a class. You define an object of your new type just as you define an integer variable:

Cat Frisky; // define a Cat

The above statement defines Frisky, which is an object whose class (or type) is Cat.

5.4 Accessing Class Members

Once you define an actual Cat object, for example, Frisky, you use the dot operator (.) to access the members of that object. Therefore, to assign 50 to Frisky's Weight member variable, you would write **Frisky.Weight = 50;**

In the same way, to call the Meow() function, you would write Frisky.Meow();

When you use a class method, you call the method. In this example, you are calling Meow() on Frisky.

5.5 Private Versus Public

Other keywords are used in the declaration of a class. Two of the most important are **public** and **private.**

All members of a class--data and methods--are private by default. Private members can be accessed only within methods of the class itself. Public members can be accessed through any object of the class. consider an example.

```
class Cat
{
        int  Age;
        int  Weight;
        Meow();
};
```

In this declaration, **Age**, **Weight**, and **Meow()** are all private, because all members of a class are private by default. This means that unless you specify otherwise, they are private.
However, if you write

```
Cat  Frisky;
Frisky.itsAge=5;      // error! can't access private data!
```

The way to use Cat so that you can access the data members is

```
class Cat
{
public:
        int  Age;
        int  Weight;
        Meow();
};
```

Now Age, Weight, and Meow() are all public. Frisky. Age=5 compiles without problems.

Example: Accessing the public members of a simple class.

```
 #include <iostream.h>        // for cout
 class Cat                   // declare the class object
{
 public:                     // members which follow are public
            int Age;
            int Weight;
};

 void main()
{
            Cat Frisky;
            Frisky. Age = 5;       // assign to the member variable
            cout << "Frisky is a cat who is " ;
            cout << Frisky.Age << " years old.\n";
```

Output: Frisky is a cat who is 5 years old.

5.6 Defining member functions

Member functions can be defined in 2 places:

1. Outside the class definition
2. Inside the class definition

5.6.1 Outside the class definition

Member functions that are declared inside a class have to be defined separately outside the class. They should have a function body and a function body.

The general form of a member function is

```
Return type classname :: functionname(argument declaration)
{
        Function body
}
```

The **classname ::** tells the compiler that the function **functionname** belongs to the class **classname.** The scope of the function is restricted to the **classname** specified in the header line. The symbol :: is called scope resolution operator.

5.6.2 Inside the class definition

Another method of defining a member function is to replace the function declaration by the actual function definition inside the class.

For example

```
Class cat
{
        Int age;
        Int weight;

        Public:
                Void getdata()
                Void display()
                {
                        Cout << " the age is" << age;
                        Cout << "the weight is" << weight;
                }
};
```

When the function is defined inside a function it is treated as an inline function. Normally, only small functions are defined inside the class definition.

5.7 Make Member Data Private

As a general rule of design, you should keep the member data of a class private. Therefore, you must create public functions known as **accessor methods** to set and get the private member variables. These **accessor methods** are the member functions that other parts of your program call to get and set your **private** member variables.

A *public **accessor** method* is a class member function used either to read the value of a private class member variable or to set its value.

5.8 Making an outside function inline

We can define a member unction outside the class definition and still make it inline by just using the qualifier **inline in the header line of function definition. For example**

```
Class item
{
        ….
        …….
Public:
        Void getdata (int a, float b);
};
Inline void item :: getdata (int a,f loat b)
{
        Number=a;
        Cost=b;
}
```

5.9 Nesting of member functions

A member function of a class can be called only by an object of that class using the dot operator. A member function can be called by using its name inside another member function of the same class. This is known as nesting of member functions.

```
Class set
{
        Int m,n;
Public:
        Void input();
Void display();
Int largest();
};

Void set :: input()
{
        Cout << " enter two values" << endl;
        Cin >> m >>n;
}

Int set :: largest()
{
        If (m>=n)
                Return (m);
        Else
```

```
                Return (n);
}

Void set :: display()
{
        Cout << “largest value is” << largest() << endl;
}

Main()
{
        Set s;
        s.input();
        s.display();
        ();
}
```

output

```
Input 2 values
25
12

Largest value is 25
```

5.10 Private member functions

Although it is a normal practice to place all data items in a private section and all the functions in public section of a class, some situations may require certain functions to be hidden from the outside calls. A private member function can only be called by another function that is a member of its class. Even an object can not invoke a private function using the dot operator. Consider the following example;

```
Class sample
{
        Int m;
        Void read();            //private member function
Public:
        Void update();
        Void write();
};
```

If S1 is an object of the class sample, then

S1.read(); will not work because objects can not access private members. However the function read() can be called by the function update() to update the value of m.

```
Void sample::update()
{
        Read();
}
```

Will work and is legal.

5.11 Arrays within a class

The arrays can be used as member variables in a class. The following class definition is valid.

```
Class cat
{
        Int a[10];
        Public:
                Void getdata();
                Void display();
};
```

The variable a[10] declared as a private member of the class Cat. This can be used in the member functions like any other array variables.

5.12 Memory allocation for objects

The memory space for objects is allocated when they are declared and not when the class is specified. The member functions are created and placed in the memory space only when they are defined as a part of a class specification. Since all the objects belonging to that class use the same member functions, no separate space is allocated for member functions when the objects are created. Only space for member variables are allocated separately for each object.

5.13 Static data members

A data member of a class can be qualified as static. A static member variable has certain special characteristics. They are

1. It is initialized to zero when the first object of its class is created. No other initialization is permitted.
2. 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.
3. 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. Consider the following example:

```
#include<iostream.h>
#include<conio.h>
class item
{
        static int count;
        int number;
        public:
                void getdata(int a);
                void getcount();
};

int item::count;

void item::getdata(int a)
{
        number=a;
        count++;
}

void item::getcount()
{
        cout << "the count is" << count <<endl;
}


main()
{
        item a,b,c;
        a.getcount();
        b.getcount();
        c.getcount();

        a.getdata(100);
        b.getdata(200);
        c.getdata(300);

        cout <<"After reading data" << endl;

        a.getcount();
        b.getcount();
```

```
        c.getcount();
getch();
}
```

The output is:

```
Count 0
Count 0
Count 0
```

After reading data

```
The count is 3
The count is 3
The count is 3
```

Note that the type and scope of each static member variable must be defined outside the class definition. This is necessary because the static data members are stored separately rather than as a part of an object. Since they are associated with the class itself, they are also known as class variables.

5.14 Static member functions

Like static member variables we can also have static member functions. A member function that is declared as static has the following properties.

- A static function can have access to only other static members declared in the same class.
- A static member function can be called using the class name(instead of its objects) as follows

  Class name::function name;

In the program given below, the static function showcount() displays the number of objects created till that moment. A count of number of objects created is maintained by the static variable **count**.

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

class test
{
        int code;               //static member variable
        static int count;
```

```
        public:
        void setcode();
        void showcode();
        static void showcount();        //static member function
};

int test::count;

void test::setcode()
{
        code=++count;
}

void test::showcode()
{
        cout << "object number" << code << endl;
}

void test :: showcount()
{
        cout << "the count is" << count <<endl;
}

main()
{
        test t1,t2;
        t1.setcode();
        t2.setcode();
        test::showcount();

        test t3;
        t3.setcode();
        test::showcount(); //accessing static function

        t1.showcode();
        t2.showcode();
        t3.showcode();

        getch();
}
```

5.15 Array of objects

We can have arrays of variables of the type **class.** Such variables are called **arrays of objects.** Consider the following class definition.

```
Class employee
{
        Char name[10];
        Float age;
Public:
        Void getdata();
        Void display();
};
```

The identifier **employee** of a user defined data type and can be used to create objects that relate to different categories of employees.

```
Employee manager[3];
Employee worker[7];
```

In the above example, the array manager contains 3 objects namely, manager[0], manager[1], manager[2] of type **employee.** Similarly, the worker array contains 7 objects.

The statement manager[i]. display() will display the data of ith element of the array **manager.**

5.16 Objects as function arguments

Like any other data type, an object may be used as a function argument. This can be done in 2 ways:

1. A copy of the entire object is passed to the function
2. Only the address of the object is transferred to the function

The first method is called **pass by value.** Since a copy of the object is passed to the function, any changes made to the object inside the function do not affect the object used to call the function.

The second method is called pass by reference. When the address of the object is passed, the called function works directly on the actual object used in the call. This means that any changes made to the object inside the function will reflect in the actual object.

Example: to illustrate objects as function arguments

```
#include<iostream.h>
#include<conio.h>
class number
{
        public:
                int num1, num2, ans;
                void display();
                void sum(number, number);
        };

void number:: display()
{
        cout << " the sum is" << ans;
}

void number:: sum(number N1, number N2)
{

        ans=N1.num1+ N2.num2;
}

main()
{
        number N1, N2, N3;
        clrscr();
        N1.num1=2;
        N2.num2=3;
        N3.sum(N1, N2);
/*      cout << "the sum is" << N3.ans;*/
        N3.display();
}
```

5.17 Returning objects

A function can not only receive objects as arguments but also can return them. The following example illustrates how an object can be created and returned to another function.

Example : without using friend function

```
// returning ojects without using friend function
#include<iostream.h>
class Number
{
        public:
```

```
        int n1,n2,n3;
        Number sum(Number,Number); // passing object of a class
        void display();
};

Number Number:: sum(Number N1, Number N2)
{
        Number T;               // a new object T is created
        T.n3=N1.n1+N2.n2;
        return T;               //returning object T
}

void Number :: display()
{

    cout << "the sum is" << n3 << "\n";
}

main()
{
Number N1, N2, N3;
N1.n1=2;
N2.n2=3;
N3=N3.sum(N1,N2);
N3.display();
}
```

Note: the first N3 is used to return the value. The object T has the sum, it is received by N3. the second N3 is used because sum() is a member function. so to use that we have to use an object. it may not be N3. it can be N2 or N1

Example: using friend function

```
#include<iostream.h>
class add
{
        int num1, num2, num3;
        public:
        void getdata1(int n1);
        void getdata2(int n2);
        friend add sum(add, add); // passing 2 objects of a class
        void display();
};
```

```
void add::getdata1( int n1)
{
        num1=n1;
}
 void add:: getdata2(int n2)
 {
        num2=n2;
 }

add sum(add a1, add a2)
{
        add a3;                         //a3 object is created
        a3.num3=a1.num1+a2.num2;
        return a3;              //returning object a3
}

void add :: display()
{
    cout << "the sum is" << num3 << "\n";
}

main()
{
        add A, B, C;
        A. getdata1(2);
        B.getdata2(3);
        C=sum(A,B);
        C.display();
}
```

5.18 Friendly functions

It is known that the private members cannot be accessed from outside the class. i.e a non member function cannot have an access to the private data of a class. However there could be situation where we would like two classes to share a particular function. For ex, consider 2 classes **manager** and **scientist** have been defined. We would like to use a function **incometax ()** to operate on the objects of both these classes. In such situations, C++ allows the common function to be made friendly with both the classes thereby allowing the function to have access to the private data of these classes. Such a function need not be the member of any of these classes.

To make an outside function friendly to a class, we have to simply declare this function as a friend of the class as shown below.

```
Class cat
{
        ….
        …
        Public:
                ..
                …
        Friend void Mouse(void);
};
```

The function declaration should be preceded by the keyword **friend.** The function definition does not use either the keyword friend or the scope resolution operator :: . A friend function, although not a member function, has full access rights to the private members of the class.

A friend function has the following characteristics

1. It is not on the scope of the class to which it is declared as friend.
2. Since it is not in the scope of the class, it can not be called using the object of the class.
3. It can be invoked like a normal function without the help of any object.
4. Unlike member functions, it can not access the member names directly and has to use an object name and the dot membership operator with each name.
5. It can be declared either in the public or in the private part of a class.

Example : to calculate mean of 2 values

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

class sample
{
        int a, b;
        public:
                void setvalue();
                friend float mean(sample s); //indicates mean() is a friend of class sample
};

void sample::setvalue()
{
        a=25;
        b=40;
}

float mean(sample s)//here need not use sample::mean(). because mean is not
```

```
{                       //a member function of Sample class. it is just a function who is friend with
sample
        return float(s.a +s.b)/2.0;
}

main()
{
        sample x; //object x is the type class sample
        x.setvalue();
        cout << "mean value" << mean(x);
        getch();
}
```

Example : To find maximum of 2 numbers

```
#include<iostream.h>
#include<conio.h>
class XYZ;
class ABC
{

        int i;
        public:
        void setvalue();
        friend void max(ABC a,XYZ b);
};

class XYZ
{
        int j;
        public:
        void setvalue();
        friend void max(ABC a,XYZ b);
};

void ABC::setvalue()
{
        i=30;
}

void XYZ::setvalue()
{
        j=20;
}
```

```
void max(ABC a, XYZ b)
{
        if  (a.i > b.j)
                cout << a.i;
        else
                cout << b.j;
}
main()
{
clrscr();
ABC abc;
abc.setvalue();
XYZ xyz;
xyz.setvalue();
max(abc, xyz);
getch();
}
```

5.19 Return by reference

A function can also return a reference. Consider the following function.

```
Int & max (int &x, int &y)
{
        If(x>y)
                Return x;
        Else
                Return y;
}
```

Since the return type of max() is **int &,** the function return reference to x or y (not the value), then a function call such as **max(a,b)** will yield a reference to either a or b depending on their values. This means that this function call can appear on the left hand side of an assignment statement. i.e the statement

Max(a,b)=-1; is legal and assigns -1 to a if it is larger, otherwise -1 to b

5.20 Const member functions

If a member unction does not alter any data in the class, then we may declare it as a const member function as follows

```
Void mul(int, int) const;
Double getbalance() const;
```

The qualifier const is appended to the function prototypes (in both declaration and definition). The compiler generates an error message if such functions try to alter the data values.

5.21 Assignemnt5

1. Explain the usage of static data members with example.
2. What is a class? What is an object?
3. Write a note on nesting of classes.
4. Write a note on array of objects.
5. Write a note on objects as functional arguments.
6. How do you define member function outside the class? Give example.
7. Differentiate static data member and non static data member in a class.
8. When you use static data members. Given an example.
9. How do you access private member function of a class?
10. What is friend function? Why it is required.
11. Give any two properties of friend function.
12. What are constant member functions
13. Give the general form of class definition
14. Differentiate private and public members of class.

UNIT-III

Chapter 6
Constructors and Destructors

6.1 Introduction

There are two ways to define an integer variable. You can define the variable and then assign a value to it later in the program. For example,

```
int Weight;                 // define a variable
...                         // other code here
Weight = 7;                  // assign it a value
```

Or you can define the integer and immediately initialize it. For example,

```
int Weight = 7;             // define and initialize to 7
```

Initialization combines the definition of the variable with its initial assignment. How do you initialize the member data of a class? Classes have a special member function called a constructor. The constructor can take parameters as needed, but it cannot have a return value--not even void. The constructor is a class method with the same name as the class itself.

Whenever you declare a constructor, you'll also want to declare a destructor. Just as constructors create and initialize objects of your class, destructors clean up after your object and free any memory you might have allocated. A destructor always has the name of the class, preceded by a tilde (~). Destructors take **no arguments and have no return value**. Therefore, the Cat declaration includes **~Cat();**

6.2 Default Constructors and Destructors

If you don't declare a constructor or a destructor, the compiler makes one for you. The default constructor and destructor take no arguments and do nothing.
To declare an object without passing in parameters, such as **Cat Frisky**; you must have a constructor in the form **Cat();**

When you define an object of a class, the constructor is called. If the Cat constructor took two parameters, you might define a Cat object by writing **Cat Frisky (5,7);**

If the constructor took one parameter, you would write **Cat Frisky (3);**

In the event that the constructor takes no parameters at all, you leave off the parentheses and write **Cat Frisky ;**

This is an exception to the rule that states all functions require parentheses, even if they take no parameters. This is why you are able to write **Cat Frisky;** which is a call to the default constructor. It provides no parameters, and it leaves off the parentheses. You don't have to use the compiler-provided default constructor. You are always free to write your own constructor with no parameters. Even constructors with no parameters can have a function body in which they initialize their objects or do other work.
As a matter of form, if you declare a constructor, be sure to declare a destructor, even if your destructor does nothing.

The constructor functions have some special characteristics. They are

1. They should be declared in the public section.
2. They are invoked automatically when the objects are created.
3. They do not have return types, not even void and therefore they can not return values.
4. They can not be inherited, though the derived class can call the base class constructor.
5. They can have default arguments.

Example: Using constructors and destructors.

```
 #include<iostream.h>
#include<conio.h>
class cat
{
        public:
                cat();
                ~cat();
                int age;
                int weight;
                char petname[20];
                void getdata();
                void display();
};
void cat :: cat()
{
        age=0;
        weight=0;
}

void cat::getdata()
{
        cout << "enter Frisky's age" << endl;
        cin >> age;
        cout << "enter Frisky's weight" << endl;
        cin >> weight;
```

```
        cout << "enter frisky's pet name" << endl;
        cin >> petname;
}

void cat ::display()
{
        cout << " the age is " << age << endl;
        cout << " the weight is " << weight;
        cout << " the pet name is" << petname;
}

main()
{
        clrscr();
        cat frisky;
        frisky.getdata();
        frisky.display();
        getch();

}
```

6.3 Parameterized constructors

It may be necessary to initialize the various data elements of different objects with different values when they are created. C++ allows us to achieve this by passing arguments to the constructor function when the objects are created. The constructors that can take arguments are called **parameterized constructors.**

When a constructor has been parameterized, the object declaration statement such as **cat frisky;** does not work. We must pass the initial values as arguments to the constructor function when the object is declared. This can be done in 2 ways.

1. By calling the constructor explicitly
2. By calling the constructor implicitly.

The following declaration illustrates the first method.

Cat frisky =cat(10); //explicit call

This statement creates a cat object frisky and passes the value 10 to it. The second is implemented as follows

Cat frisky(10) // implicit call.

This method is used very often and easy to implement.

Example:

```
#include <conio.H>
#include<iostream.h>
#include<string.h>
class address
{
        private:
          int houseno;
          char street[20];
          char city[20];
          char state[20];
        public:
          address(int hno, char st[20], char ct[20], char stt[20]);
          void display();
};

address::address (int hno, char st[20], char ct[20], char stt[20])
{
           houseno=hno;
           strcpy(street,st);
           strcpy(city,ct);
           strcpy(state,stt);
}

void address::display()
{
        cout << "Hose Number: " << houseno << endl;
        cout << "Street: " << street <<endl;

        cout << "City: " << city << endl;
        cout << "State: " << state << endl;
}

void main()
{

          int hno;
          char st[20];
          char ct[20];
          char stt[20];
        clrscr();
         cout << "enter the House number" << endl;
         cin >> hno;
         cout << "enter the street" << endl;
```

```
        cin >> st;
        cout << "enter the city" << endl;
        cin >> ct;
        cout << "enter the state" << endl;
        cin >> stt;
        address add1( hno, st, ct, stt);
        add1.display();
        getch();
}
```

6.4 Overloading Constructors

The point of a constructor is to establish the object; for example, the point of a Rectangle constructor is to make a rectangle. Before the constructor runs, there is no rectangle, just an area of memory. After the constructor finishes, there is a complete, ready-to-use rectangle object.

Constructors, like all member functions, can be overloaded. The ability to overload constructors is very powerful and very flexible.

For example, you might have a **cat** object that has two constructors: The first takes a name, age and weigh. The second takes no values and prints the default value. The following example implements this idea.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class cat
{
        private:
                int age;
                int weight;
                char cname[20];
        public:
                cat(int a, int w, char name[20]);
                cat();
                void getdata();
                void display();
        }

           cat::cat (int a, int w, char name[20]) //parametrised constructor
                {
                        age=a;
                        weight=w;
```

```
                        strcpy(cname,name);
                }

        cat::cat()

        {
                age=10;
                weight=30;
                strcpy(cname,"Frisky");
        }

void cat :: display()
{
        cout << "name is" << cname;
        cout << "age is " << age;
        cout << "weight is" << weight;
}

main()
        clrscr();
        cat frisky;
        frisky.display();
        cat billy(12, 90, "Billy");
        billy.display();
        getch();
}
```

Output

```
 Name is Frisky
Age is 10
Weight is 30

Name is Silly
Age is 12
Weight is 90
```

6.5 Constructors with default arguments

It is possible to define constructors with default arguments. For example, the constructor cat() is declared as follows.

```
Cat( int age, char name[10]=”Billy”);
```

The default value of the argument name is **Billy**. Then the statement cat(10); assigns the value to the age variable and Billy to name variable (by default). However the statement cat(10, Silly); assigns the value 10 to age and Silly to name variable. The actual parameter, when specified, overrides the default value.

Example: to illustrate constructors with default values

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class cat
{
        private:
                int age;
                int weight;
                char cname[20];
        public:
                cat(int a, int w, char name[20]);
                void display();
        }

           cat::cat (int a, int w, char name[20]="Billy") //parametrised constructor with defalut
value
                {
                        age=a;
                        weight=w;
                        strcpy(cname,name);
                }

void cat :: display()
{
        cout << "the age is " << age << endl;
        cout << "the weight is" << weight;
        cout << "the name is" << cname << endl;

}

main()
{
clrscr();
cat Billy(5, 20);
Billy.display();
getch();
}
```

6.6 Dynamic initialization of objects

Class objects can be initialized dynamically too. The initial value of an object may be provided during runtime. One advantage of dynamic initialization is that we can provide various initialization formats, using overloaded constructors.

6.7 The Copy Constructor

In addition to providing a default constructor and destructor, the compiler provides a default copy constructor. The copy constructor is called every time a copy of an object is made.

When you pass an object by value, either into a function or as a function's return value, a temporary copy of that object is made. If the object is a user-defined object, the class's copy constructor is called. All copy constructors take one parameter, a reference to an object of the same class. It is a good idea to make it a constant reference, because the constructor will not have to alter the object passed in. For example:

CAT(const CAT & theCat);

Here the CAT constructor takes a constant reference to an existing CAT object. The goal of the copy constructor is to make a copy of theCAT.

The default copy constructor simply copies each member variable from the object passed as a parameter to the member variables of the new object. This is called a member-wise (or shallow) copy, and although this is fine for most member variables, it breaks pretty quickly for member variables that are pointers to objects on the free store.

A *shallow* or *member-wise* copy copies the exact values of one object's member variables into another object. Pointers in both objects end up pointing to the same memory. A deep copy copies the values allocated on the heap to newly allocated memory.

If the CAT class includes a member variable, itsAge, that points to an integer on the free store, the default copy constructor will copy the passed-in CAT's itsAge member variable to the new CAT's itsAge member variable. The two objects will now point to the same memory.

Write a program to illustrate Copy constructor

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

class cat
{
```

```
        private:
                int age;
                int weight;
                char cname[20];
        public:
                cat(int a, int w, char name[20]);
                cat(cat &);//prototype
                void display();
        }

           cat::cat (int a, int w, char name[20]="Billy") //parametrised constructor with defalut
value
                {
                        age=a;
                        weight=w;
                        strcpy(cname,name);
                }

        cat::cat(cat &c)
        {
                age=c.age;
                weight=c.weight;
                strcpy(cname,c.cname);
        }
void cat :: display()
{
        cout << "the name is" << cname << endl;
        cout << "the age is " << age << endl;
        cout << "the weight is" << weight << endl;;
}

main()
{
clrscr();
cat Billy(5, 20);
Billy.display();
cat Silly(Billy);
Silly.display();
getch();
}
```

6.8 Dynamic constructors

The constructors are also used to create memory while creating objects. This will enable the system to allocate right amount of memory for each object when the objects are not of the same size, thus resulting in saving memory. Allocation of memory to objects at the time of their construction is known as dynamic construction of objects. The memory is allocated with the help of the new operator.

6.9 Const objects

We may create and use constant objects using **const** keyword before object declaration. For example, we may create x as a constant object of the class **matrix** as follows.

```
Const matrix x();                    //object x is constant
```

Any attempt to modify the values of m and n will generate compile time error. A constant object can call only **const** member functions. Whenever **const** objects try to invoke **non-const** member functions, the compiler generate errors.

6.10 Destructors

A destructor is used to destroy the objects that have been created by the constructor. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde. For example, the destructor of the class **cat** cab ne defined as ~cat(){ }

A destructor never takes any argument nor does it return any value. It will be invoked by the compiler after exiting from the program to clean up storage that is no longer accessible. It is a good practice to declare destructors in a program since it releases memory space for future use.

Whenever new is used to allocate memory in the constructors, we should use delete to free that memory.

6.11 Assignment6

1) How constructor is invoked? Give example.
2) Differentiate default constructor and constructor with default arguments.
3) How do you define parameterised constructor?
4) How to define constructor function inline? Give example
5) Give any two features of constructors.
6) How do you define copy constructor?
7) Why copy constructor takes reference variable as argument?
8) What is destructor? How do you define it?
9) What are the characteristics of constructor?
10) What are the different ways of calling constructor? Explain with example.
11) What do you mean by dynamic initialisation of objects? Explain
12) Write a note on copy constructors. \
13) What is dynamic constructor? Explain.
14) Define class string having data member for holding string data and its length. Include all types of constructors to initialise objects. Write a program to test your class to do the following
    - To copy one object to another
    - two join two strings using overloaded + operator
15) Explain how multiple constructors are defined in a class with example.
16) What is meant by constructor overloading? Explain with code example.

Chapter 7
Operator Overloading

7.1 Introduction

Operator loading is one of the many exciting features of C++ language. C++ has the ability to provide the operators with a special meaning for a data type. The mechanism of giving special meaning to an operator is known as operator overloading.

We can overload i.e give additional meaning to all the C++ operators except the following

1. Class member access operators(. and *)
2. Scope resolution operator (::)
3. Sizeof operator (sizeof)
4. Conditional operator(:?)

7.2 Defining operator overloading

To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied. This is done with the help of a special function called **operator** function, which describes the task.

The general form of an operator function is

```
Return type class_name :: operator op(argument list)
{
        Function body
}
```

Where return type is the type of value returned by the specified operation and op is the operator being overloaded. The op is preceded by the keyword **operator. Operator op** is the function name.

Operator function must be either member functions or friend functions. A basic difference between them is that a friend function will have only one argument for unary operator and two for binary operators. While a member function has no arguments for unary and only one for binary operators.

7.3 Overloading unary operators

Consider a unary minus operator. A minus operator when used as a unary, takes just one operand. This operator changes the sign of an operand when applied to a data item. We will see how to overload this operator so that it can be applied to an object in mush the same way

as is applied to an int or float variables. The unary minus when applied to an object should change the sign of each of its data items.

Example 1: overloading ++ operator and –- operator.

```
#include<iostream.h>
#include<conio.h>
class Number
{
        int a,b,c;
        public:
        void getdata();
        void display();
        void operator-();
};

void Number::getdata()
{
        cout << "enter 3 numbers\n";
        cin >> a>> b >> c;
}

void Number:: display()
{

        cout << a << " " ;
        cout << b << " " ;
        cout << c << "\n";
}

void Number :: operator -()
{
        a=-a;
        b=-b;
        c=-c;
}

main()
{
        clrscr();
        Number N;
        N.getdata();
        cout << " the original Numbers are \n";
        N.display();
```

```
        -N;
        cout << "the numbers after applying Unary Minus are\n";
        N.display();
        getch();
}
```

Example 2: overloading ++ operator and –- operator.

```
#include<iostream.h>
#include<conio.h>
class Number
{
        public:
        int num1;
        Number();

        void getdata();

        Number operator++();
        Number operator--();
};

Number::Number()
{
        num1=0;
}

Number Number :: operator++()
{
        num1++;
}

Number Number :: operator--()
{
        num1--;
}

void Number::getdata()
{
        cout << "enter the number";
        cin >> num1;
}

main()
```

```
{
        int choice=-1;
        Number n;
        clrscr();
        while(choice<5)
        {
                cout << "{enter 1,2,3,4 for getdata, increment, decrement, exit}" << endl;
                cin >> choice;
                switch(choice)
                {
                        case 1: n.getdata();
                                break;
                        case 2: n++;
                                cout << "the incremented value is" << n.num1;
                                break;
                        case 3: n--;
                                cout << "the decremented value is" << n.num1;
                                break;
                }
        }
        getch();
}
```

7.4 Overloading Binary operator

The same mechanism can be used to overload a binary operator. We can overload a + operator using an operator+() function.

Program to add 2 numbers using operator overloading

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

class Number
{
        public:
        int num;
        Number operator+(Number n);
};

Number Number::operator+(Number n)
{
        Number c;
```

```
        c.num = num + n.num;
        return c;
}

int main()
{
        Number n1, n2, n3;
        clrscr();
        cout << "enter the first number";
        cin >> n1.num;
        cout << "enter the 2 number";
        cin >> n2.num;
        n3=n1+n2;
        cout << n3.num << endl;
        getch();
        return 0;
}
```

7.5 Overloading Binary operator using friends

As stated earlier, **friend** functions may be used in the place of member functions for overloading a binary operator, the only difference being that a friend function requires two arguments to be passed to it, while a member function requires only one.

Write a program to illustrate binary operators using friends

7.6 Overloading arithmetic assignment operators

We can overload arithmetic assignment operators too. For example the +=operator. This operator combines assignment and addition into one step. We will use this operator to add 2 numbers a and leaving the result in the first.

```
//overloading arithmetic assignment operators
#include <iostream.h>
#include <conio.h>

class Number
{
        public:
        int num;
        void operator+=(Number n);
};
void Number::operator+=(Number n)
```

```
{

        num+=n.num;

}

int main()
{
        Number n1, n2, n3;
        clrscr();
        cout << "enter the first number";
        cin >> n1.num;
        cout << "enter the 2 number";
        cin >> n2.num;
        n1+=n2;
        cout << n1.num << endl;
        getch();
        return 0;
}
```

7.7 Manipulation of strings using operators

Concatenating strings

The + operator cannot be used to concatenate C strings. That's why we can write S3=S1+S2; where S1, S2 and S3 are string variables. However if we use our own String class, then we can overload the + operator to perform such concatenation.

The == operator can be used to compare 2 string objects. We will use the == to compare 2 string objects, returning true if they are the same and false if they are different

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

class MyString {
        public:
                getdata();
                display();
                char str[50];

                MyString operator + (MyString a);
                int operator == (MyString a);
                int operator > (MyString a);
```

```
                int operator < (MyString a);
};

MyString::getdata()
{
        cout<<"Enter the string";
        cin>>str;
}

MyString::display()
{
        cout<<"String is "<<str<<endl;
}

MyString MyString::operator +(MyString x)
{
        MyString c;
        strcpy(c.str, str);
        strcat(c.str, x.str);

        return c;
}

int MyString::operator ==(MyString x)
{
        if (strcmp(str, x.str) == 0)
                return 1;
        else
                return 0;
}

int MyString::operator <(MyString x)
{
        if (strcmp(str, x.str) < 0)
                return 1;
        else
                return 0;
}

int MyString::operator > (MyString x)
{
        if (strcmp(str, x.str) > 0)
                return 1;
        else
```

```
                return 0;
}

int main()
{
 MyString a, b, c;
 int choice=-1;

 clrscr();

 a.getdata();
 b.getdata();

 while (choice < 2) {
        cout<<"Enter {1,2} for {cat,equal}"<<endl;
        cin>>choice;
        switch(choice) {
        case 1: // cat
                c = (a + b);
                c.display();
                break;
        case 2: // compare
                if (a == b)
                        cout << "Equal"<<endl;
                else if (a < b)
                        cout <<"Less Than\n";
                else
                        cout << "Greater than";
                break;
        }
 }
 getch();
}

/* Note :
 * This is to teach the '+' operator to handle classes created by us
 * By default, those operators understand only int, float, char, long
 */
```

7.8 Comparison operators

Let us see how to overload a different kind of C++ operator: comparison operator.

```
//overloading comparison operator(<)
```

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

class Number
{
        public:
        int num;
        int operator<(Number n);
/*      void getdata1();
        void getdata2();*/
};

int Number::operator<(Number n)
{
  if (num < n.num)
        return 1;
  else
        return 0;

}

int main()
{
        Number n1, n2, n3;

        clrscr();

        cout << "enter the first number";
        cin >> n1.num;
        cout << "enter the 2 number";
        cin >> n2.num;
if (n1<n2)
cout << n1.num << "n1 is small";
else
cout << n2.num << "is small";
        getch();

        return 0;
}
```

7.9 Rules for overloading operators

1. Only existing operators can be overloaded. New operators cannot be created.
2. The overload operator must have at least one operand that is of user defined type.

3. We cannot change the basic meaning of an operator. That is to say, we cannot redefine the plus (+)operator to subtract one value form other.
4. Overloaded operators follow the syntax rules of the original operators. They can not to overridden.
5. There are some operators that cannot be overloaded.
6. We cannot use friend functions to overload certain operators. They are =, (), [] and →.
7. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but, those overloaded by means of a friend function, take one reference argument.
8. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
9. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
10. Binary arithmetic operators such as +, -, * and / must explicitly return a value. They must not attempt to change their own arguments.

7.10 Pitfalls of operator overloading and conversion

1. **Use similar meanings:** use overload operators to perform operations that are as similar as possible to those performed on basic data types. Overloading an operator assumes that t makes sense to perform a particular operation on objects of a certain class. If we are going to overload the + operator in class X, then the result of adding two objects of class X should have a meaning at least somewhat similar to addition.

2. **Use similar syntax:** use overloaded operators in the same way you use basic types. For example a and b are basic types, the assignment operator in the statement operator in the statement a+=b; sets a to the sum of a and b. any overloaded version of those operator should do something analogues. It should probably do the same thing as a=a+b; where the + is overloaded. Some syntactical characteristics of operators can't be changed even if you want them to**.** You can't overload a binary operator to be a unary operator or vive versa.

3. **Show restraint:** if the number of overloaded operators grows too large and if they are not used in a proper way, then the whole point of using them is lost. Use overloaded operators only when it is a must to use.

4. **Not all operators can be overloaded:** the following operators cannot be overloaded: the member access or dot operator(.), the scope resolution operator(:☺, and the conditional operator(?:). Also the pointer- to- member operator(→) cannot be overloaded.

7.11 Assignment 7

1. What do you mean by overloading a operator?
2. Give the general form of operator function.
3. Give any two rules of overloading operator.
4. List any four operators that cannot be overloaded.
5. List the operators that cannot be overloaded by friend function.
6. What is the use of keyword ***operator*** in C++? Give example.
7. What are the rules for overloading a operator?
8. How do you overload a unary operator using member function ? Explain with example.
9. How do you overload a unary operator using friend function ? Explain with example
10. Define a class String. Using overloaded operator = = check whether two strings are equal or not .

UNIT-IV

Chapter 8
Inheritance

8.1 Introduction

The mechanism of deriving a new class from an old one is called inheritance. The old class is referred to as the base class and the new one is called the derived class or sub class.

The derived class inherits some or all the properties from the base class. A derived class with only one base class is called single inheritance and with several base classes is called multiple inheritance.

On the other hand, the properties 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.

8.2 Defining derived classes

A derived class can be defined by specifying its relationship with the base class in addition to its own details. The general from of defining a derived class is

```
Class derived_class_name : visibility mode basecalss_name
{
        ....
        ....            // members of derived class
        ....
} ;
```

The colon indicates that the derived class name is derived from the baseclass_name. the visibility mode is optional and if present, may be either private or public. The default is private.

For example

```
Class ABC: private XYZ
{
        Members of ABC;
};

Class ABC : public XYZ
{
        members of ABC;
```

};

When a base is privately inherited by a derived class, the 'public members' of the base class become the private members of the derived class and therefore the public members of the base class can only be accesses by the member functions of the derived class. They cannot be accessed by the objects of the derived class.

When the base is publicly inherited, the 'public members' of the base class become the 'public members of the derived class and therefore they are accessible to the members functions and the objects of the derived class. In both the cases, the private members are not inherited and therefore the private members of the base class will never become the members of the derived class.

```
class A
{
……….
……….
……….
};

class B public or private A
{
……..
……….
……..
};
```

In the example above, **class B** is a *"derived class"* derived from **class A**. **class A** is called the *"base class"* of **class B**.

The class B can derive from class A in two ways, **Public** or **Private**. This is called the **type of inheritance**.

1. The **type of inheritance** decides how the **public members** of class A are treated when they are inherited into class B.

- If the type of inheritance is **public**, then the **public** members of A are **public** members of B.
- If the type of inheritance is **private**, then the **public** members of A are **private** members of B.

- If a variable is a private member of a class, we know that **only member functions** can access it.
- If a variable is a public member of a class, we know that **both member functions** and **objects of the class** can access it.
- But a protected function in the base class, is also protected in the derived class always.

Example:

```
#include <iostream>

class A
 {
        public :
                void f();
};

class B : private A
 {
        public:
                void g();
};

class C : public A
{
        public:
                void h();
};

void A::f()
{
        cout << "F called" << endl;
}

void B::g()
{
        F();                            // Can call f() from here, always
        cout << "G called" << endl;
}

void C::h()
{
        F();                             // Can call f() from here, always
        cout << "H called" << endl;
}

int main()
{
        A a;
        B b;
        C c;
```

```
        a.f();                  // a.f() can be called because f is public in A

                                // b.f() can not be called because B has inherited A privately

        b.g();                  // But B.g can call f() as B.g is a member function

        c.f();                  // c.f() can be called because B has inherited A publicly

        c.h();                  // C.h can call f() has C.h is a member function

        return 0;
}
```

8.3 Making a private data member inheritable

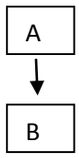
We have seen that a private member of a base class cannot be inherited and therefore it is not available for the derived class directly. The private data can be inherited by a derived class by modifying the visibility limit of the private member by making it public. But if the private data made public, it is accessible to all other functions in the program.

C++ provides a third visibility modifier called protected. A member declared as protected is accessible by the member functions within the class and any class immediately derived from it. It can not be accessed by the functions outside these 2 classes.

```
Class alpha
{
        Private:
        ……..            //visible to the member functions of its class
        …….
        Protected:
        ……..            //visible to member functions of its own and derived class
        ……..
        Public:
        ……..
        ……..            //visible to all functions in the program

};
```

When a protected member is inherited in public mode, it becomes protected in the derived class too and therefore is accessible by the member function of the derived class. A protected member inherited in the private mode, becomes private in the derived class. It is available to the member functions of the derived class.

8.4 Single inheritance

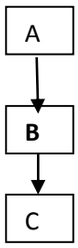


In the above diagram A is the base class and B is the derived class. i.e Class B is derived from Class A. the syntax is

```
Class A
{
        …….
        …….
};

Class B: public A
{
        ….
        ….
};
```

8.5 Multilevel inheritance

A class can be derived another derived class as shown in the figure. The class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. the class B is known as the intermediate base class since it provides a link for the inheritance between A and C. the chain ABC is known as inheritance path.



Base class

Intermediate base class

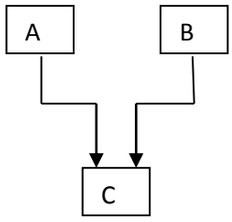
Derived class

A derived class with multilevel inheritance is declared as followed.

```
Class A { ……};              //base class
Class B : public A {…….};    //B derived from A
Class C: public B {…….};     //C derived from B
```

8.6 Multiple inheritance

A class can inherit the attributes of two or more classes as shown in the figure. This is known as multiple inheritance. Multiple inheritance allows us to combine the features of several existing classes as a starting point for defining new classes. It is like a child inheriting the physical features of one parent and intelligence from of another.

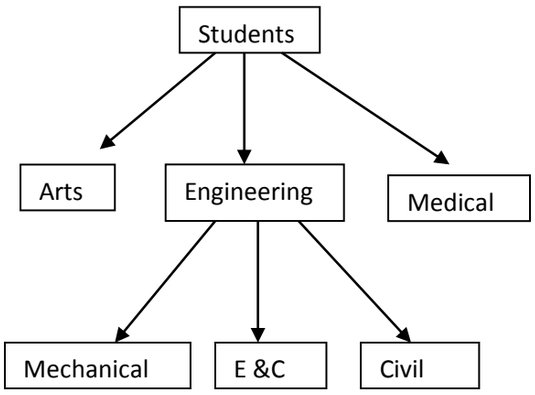


The syntax of a derived class with multiple base classes is as follows:

```
Class C: public B, public A
{
        Body of C
        ……
};
```

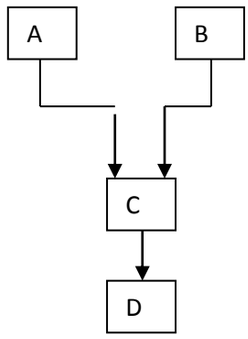
8.7 Hierarchical inheritance

In this the base class will include all the features that are common to the subclasses. A subclass can be constructed by inheriting the properties of the base class. A subclass can serve as a base class for the lower level classes s so on. In the diagram below, all the students have certain things in common.



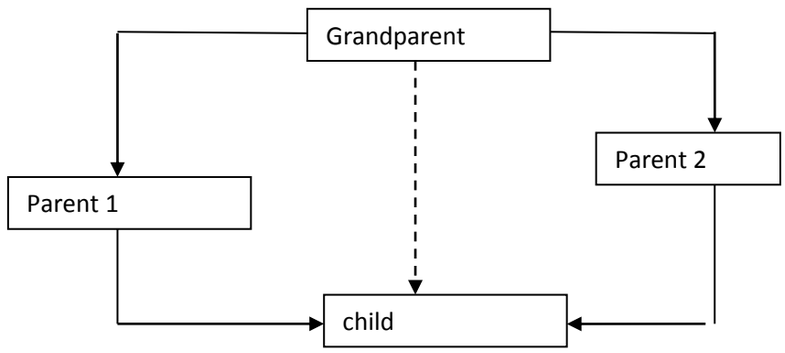
8.8 Hybrid inheritance

This is a mixture of 2 or more types of inheritance.



8.9 Virtual base classes

Consider a situation where all the three kinds of inheritance namely, multilevel, multiple and hierarchical inheritance are involved. This is illustrated in the following figure. The child has two direct base classes 'parent1' and 'parent2' which themselves have a common base class 'grandparent'. The 'child' inherits the features of 'grandparent' via two separate paths. It can also inherit directly as shown by the broken line. The grandparent is sometimes referred to as indirect base class.



Inheritance by the child might pose some problems. All the public and protected members of grandparent are inherited into child twice via parent1 and via parent2. This means child may have duplicate sets of members inherited from grandparent. This introduces ambiguity and should be avoided.

The duplication of inherited members due to these multiple paths can be avoided by making the common base class as virtual base class while declaring the direct or intermediate base classes as shown below.

```
Class A {……};                          //grandparent
Class B1: Virtual public A {……};        //parent1
Class B2: virtual public A {……..};      //parent2
Class c: public B1, public B2{……..};    //child. Only one copy of A will be inherited.
```

When a class is made a virtual base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exists between the virtual base class and a derived class.

8.10 Ambiguity resolution in inheritance

We may face a problem in using multiple inheritance, when a function with the same name appears more than one base class. Consider the following example,

```
Class M
{
        Public:
        Void display();
}

Class M::dispalay()
{
        Cout <<"Hi"
}

Class N
{
        Public:
        Void display();
};

Class N::display()
{
        Cout <<"Bye"";
}

Class P: public M, public N
{
        ……..
        ……..
};                              //assume that display() is not there in the derived class P.
Main()
{
        P p1
```

```
        P1.display();
}
```

Which display() is to be called by the derived class when both the base classes are having the same display() function? This is called ambiguity. This can be resolved as shown below. Assume that the intention of the user is to call display() of the base class.

```
Class P: public M, public N
{
        Public:
                Void  display()
                {
                        M::display();
                }
};
```

Ambiguity may arise in single inheritance applications. For example, consider the following

```
Class A
{
        Public:
                Void display()
                {
                        Cout << “A” <<endl;
                }
};


Class B:public A
{
        Public:
                Void display()
                {
                        Cout << “B\n”;
                }
};
```

In this case if a simple call to display() by B type object will invoke function defined in B only. We may invoke function defined in A using scope resolution operator to specify the class.

For example,

```
Main()
{
        B b;                            //b is derived class object
        b.display()                     //calls display in class B
        b.A::display()                  //calls display in A
```

```
        b.B::display()                  //class display in B
}
```

8.11 Abstract classes

An abstract class is one that is not used to create objects. An abstract class is designed only to act as a base class i.e to be inherited by other classes.

8.12 Constructors in derived classes

- Constructors are used for initializing objects. As long as no base class constructor takes any arguments, the derived class need not have a constructor function.
- However, if any base class contains a constructor with one or more arguments, then it is a must for the derived class to have a constructor and pass the arguments to the base class constructor.
- When both the derived and base classes contain constructors, the base class constructor is executed first and then derived class constructor.
- In case of multiple inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class.
- In a multilevel inheritance, the constructors will be executed in the order of inheritance.
- The base constructors are called and executed before executing the statements in the body of the derived class.

The general form of defining a derived constructor is:

Derivedconstructor(arglist1, arglist2….arglistn): base1(arglist1), base2(arglist2),……..basen(arglistn)
{
Body of the derived class
}

The header line of **derivedconstructor** function contains two parts separated by colon. The first part provides the declaration of arguments that are passed to the derived constructor and the second part gives the function calls to the base constructors.

```
#include<stdio.h>
#include<conio.h>
#include<iostream.h>
class alpha
{
        int x;
        public:
        alpha(int i);
```

```
        void show_x();

};

class beta
{
        float y;
        public:
                beta(float j);
                void show_y();
};

class gamma: public beta, public alpha
{
        int m,n;
        public:
                gamma(int a, float b, int c, int d): alpha(a), beta(b)
                {
                        m=c; n=d;
                        cout << "gamma initialized\n";
                }
        void show_mm()
        {
                cout << "the value of m is" << m;
                cout << "the value of n is " << n;
        }
};

void alpha::alpha(int i)
{
        x=i;
        cout <<"alpha initialized\n";
}
void alpha::show_x()
{
        cout << "the value of x is" << x << endl;
}

void beta::beta(float j)

{
        y=j;
        cout <<"beta initialized\n";
}
```

```
void beta::show_y()
{
        cout << "the value of y is" << y << endl;
}

main()
{
        gamma g(5,10.3, 20,30);
        g.show_x();
        g.show_y();
        g.show_mm();
        getch();
}
```

The output is:

```
Beta initialized
Alpha initialized
Gamma initialized
The value of x is 5
The value of y is 10.3
The value of m is 20
The value of n is 30
```

8.13 Member classes: nesting of classes

C++ supports another way of inheriting properties of one class into another. This approach takes a view that an object can be a collection of many other objects. That is the class can contain objects of other classes as its members as shown below.

```
Class alpha
{
        ……..
        ……
};
Class beta
{
        ……..
        ……..
};

Class gamma
{
```

```
        Alpha a;
        Beta b;
}
```

All objects of gamma class will contain the objects a and b. This kind of relationship is called containership or nesting.

8.14 Assignment8

1. Define base class and Derived class.
2. List four types of inheritance.
3. What is the difference between private and protected access specifies ?
4. When do we use protected access specifier in C++ ?
5. What is the difference between multiple inheritance and multilevel inheritance?
6. What is the difference between overloading and overriding?
7. Give the general form of derived class declaration.
8. What is virtual base class?
9. When do we need virtual functions?
10. What is abstract class?
11. How to define a derived class constructor?
12. What is container class?
13. What are the different types of inheritance supported by C++ ?Explain
14. Explain single inheritance with an example.
15. How ambiguity is resolved in multiple inheritance using virtual base class ? Explain
16. Explain visibility of private , protected and public members in different modes of inheritance.
17. Explain multi level inheritance with an example.
18. Explain multiple inheritance with an example.
19. Explain hybrid inheritance with an example.
20. Explain hybrid inheritance with an example.
21. Explain protected mode of inheritance with example.
22. Explain private mode of inheritance with example
23. Explain public mode of inheritance with example.

Chapter 9
Pointers, Virtual function and polymorphism

9.1 Polymorphism

Polymorphism simply means 'one name, multiple forms'. Polymorphism is implemented using the overloaded functions and operators. 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 or static binding or static linking. Also known as compile polymorphism, early binding simply means that an object is bound to its function call at compile time.

9.2 Pointers

We know that a pointer is a derived data type that holds the address of a variable.

Declaring and initializing pointers: the declaration of a pointer variable has the following form:

Datatype *pointer_variable;

Here pointer variable is the name of the pointer and the data type refers to one of the valid data type such as int, char, float and so on. The data type is followed by an * symbol which distinguishes a pointer variable from other variables to the compiler.

For example: int *p;

Where p is a pointer variable and points to an integer data type. A variable must be initialized before using it in C++ program, we can initialize a pointer variable as follows:

```
Int *p, a;          //declaration
P=&a                //initialization
```

The pointer variable p contains the address of variable **a**. the address of operator **&** is used to retrieve the address of a variable. The initialization statement assigns the address of the variable **a** to the pointer variable **p.**

9.3 Manipulation of pointers

We can manipulate a pointer with the indirection operator * which is also known as dereference operator. With this operator we can directly access the content of data variable. It takes the following form.

*pointer_variable;

We know that, dereferencing a pointer allows us to get the content of a memory location that the pointer points to. Using the dereference operator, we can change the content of the memory location.

9.4 Pointer expressions and pointer arithmetic

C++ allows pointers to perform the following arithmetic operations:

- A pointer can be incremented or decremented
- Any integer can be added to or subtracted from a pointer
- One pointer can be subtracted from another.

Examples:

```
Int a[6];
Int *p;
P=&a[0];
```

The pointer variable p refers to the base address of the variable a. we can increment the pointer variable as shown as follows.

P++ or ++p; this statement moves the pointer to the next memory address. Similarly we can decrement the pointer variable as follows:

p—or –p;. this statement moves the pointer to the previous memory address.

9.5 Using pointers with arrays and strings

We can declare the pointers to arrays as follows:

```
Int *p;
P=a[0];
```

Or

```
P=a;
```

Here p is the pointer to the first element of the integer array, a[0]. Also consider the following example:

```
Float *p;
P=a[0];
```

Or
P=a;

Here p points t the first element of the array of a.

9.6 Array of pointers

Similar to others we can create an array of pointers in C++. The array of pointers represents a collection of addresses. An array of pointers point to an array of data items. Each element of pointer array points to an item of the data array. Data items can be accessed either directly or by dereferencing the elements of pointer array.

We can declare an array of pointer as follows:

Int *a[10];

This statement declares an array of 10 pointers, each of which points to an integer. The address of the first pointer is a[0], the second pointer is a[1], and final pointer points to a[9].

9.7 Pointers and strings

We know that a string is one dimensional array of characters, which start with the index 0 and ends with a null character '\0' in C++. There are 2 ways to assign a value to a string . we can use the character array or variable of type char *. Consider the following string declaration:

Char num[]="one";

Const char *p="one";

The first declaration creates an array of 4 characters which contains the characters, 'o', 'n' 'e', '\0' where as the second declaration generates a pointer variable, which points to the first character i.e 'o' of the string.

9.8 Pointers to functions

Using function pointers we can allow a C++ program to select a function dynamically at run time. We can also pass a function as an argument to another function. Here the function is passed as a pointer. C++ allows us to compare two function pointers.

C++ provides two types of function pointers: function pointers that point to static member functions and function pointers that point to non static member functions.

Like other variables, we can declare a function pointer in C++. It takes the following form

Datatype(*function name)();

Example: **int(*function1(int x));**

Example:

```
Typedef void(*Funptr)(int, int);
Void add(int I, int j)
{
        Cout << I << "+"" << j << "=" << i+j;
}

Void subtract(int I, int j)
{
        Cout << I << "-"" << j << "=" << i-j;

}
Main()
{
        Funptr p;
        P=&Add;
        P(1,2);
        Cout <<endl;
        P=&subtract;
        P(3,2);
}
```

Output is:

1+2=3
3-2=1

9.9 Pointers to objects

Consider the following statement

Item x;

Where item is a class and x is an object of type item. Simil;arly we can define a pointer p of type item as follows.

Item *p;

Object pointers are useful in creating objects at runtime. We can also use an object pointer to access the public members of an object. Consider the following

```
Class item
{
        Int code;
        Float price;
        Public:
                Void getdata(int a, float b);
                Void show();
};

Void item::getdata(int a, float b)
{
        Code=a;
        Price=b;
}

Void item::show()
{
        Cout << “code is” << code <<endl;
        Cout <<”price is”<<price <<endl;
}
```

Let us declare an item variable x and a pointer p to x as follows.

```
Item x;
Item *p=&x;
```

The pointer p is initialized with the address of x.

We can refer to the member functions of item in 2 ways, one by using the dot operator and the object and aother by using the arrow operator and the object pointer. The statements

```
x.getdata(100,34.5);
x.show();
```

are equivalent to

```
p→getdata(100,34.5);
p→show();
```

we can also create the objects using pointers and new operator as follows:

item *p=new item; //this is called creating object at runtime.

This statement allocates enough memory for the data members in the object structure and assigns the address of the memory space to p. then p can be used to refer to the members as shown below

p→show();

9.10 This pointer

C++ uses a unique keyword called **this** to represent an object that invokes a member function. **This** is a pointer that points to the object for which **this** function was called. For example, the function call A.max() will set the pointer **this** to the address of the object A. consider the following example

```
Class A
{
        Int a;
        …….
        ……..
};
```

The private variable 'a' can be used directly inside a member function like a=123;

We can also use the following statement to do the same job.

This→a=123;

Since C++ permits the use of the form a=123, we have not been using the pointer **this** so for. Recall that when a binary operator is overloaded using a member function, we pass only one argument to the function. The other argument is implicitly passed using the pointer **this.** One important application of the pointer **this** is to return the object it points to. For example, the statement

Return *this;

Inside a member function will return the object that invoked the function.

For example

```
Person & person ::greater(person x)
{
        If x.age >age
                Return x;
```

```
        Else
                Return *this;
}
```

Suppose we invoke this function by using

Max=A.greater(B);

This function will return the object B if the age of person B is greater than that of A. otherwise it will return the object A using the pointer this.

9.11 Virtual functions

Polymorphism refers to the property by which objects belonging to different classes are able to respond to the same message, but in different forms. An essential requirement of the polymorphism is therefore the ability to refer to objects without any regard to their classes. This necessitates the use of a single pointer variable to refer to the objects of different classes. We use the pointer to base class to refer to all the derived objects. But the base pointer even when it is made to contain the address of a derived class, always executes the function in the base class. The compiler simply ignores the content of the pointer and chooses the member function that matches the type of the pointer. In this case, we can achieve polymorphism using virtual functions.

When we use the same function name in both the base and derived classes, the function in base class is declares as **virtual** using the keyword virtual. When a function is made virtual, C++determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer. Thus, by making the base pointer to point to different objects, we can execute different versions of **virtual** function.

One important point to remember that, we must access virtual functions through the use of a pointer declared as a pointer to the base class.

Rules for virtual functions

1. The virtual functions must be the members of some class.
2. They can't be static members
3. They are accessed by using object pointers.
4. A virtual function can be a friend of another class.
5. A virtual function in the base class must be defined, even though it may not be used.
6. The prototypes of the base class version of a virtual function and all the derived class versions must be identical. If two functions with the same name have different prototypes, C++ considers them as overloaded functions and the virtual function mechanism is ignored.

7. We cannot have virtual constructors, but we can have virtual destructors.
8. While a base pointer can point to any type of derived object, the reverse is not true. That is, we cannot use a pointer to a derived class to access an object of the base type.

9.12 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 used for performing nay task. For example, we have not defines any object of class 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 in a base class that has no definition relative to the base class. In such cases compiler requires each derived class to either define the function or redeclare it as a pure virtual function. A class containing pure virtual function cannot be used to declare any objects of its own. Such classes are called abstract base classes.

9.13 Assignment 9

1. What is polymorphism? What are its types?
2. What is compile time polymorphism?
3. What is this pointer?
4. What is virtual function?
5. What is pure virtual function?
6. What are the implications of making a function pure virtual ?
7. Write a note on compile time and run time polymorphism.
8. Explain how pointer to objects are used in a C++ program.
9. What are the basic rules for function to be virtual ?
10. Write a note on virtual constructors and destructors

Data Abstraction in C++

Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

Let's take one real life example of a TV, which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, BUT you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.

Thus, we can say a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having zero knowledge of its internals.

Now, if we talk in terms of C++ Programming, C++ classes provides great level of **data abstraction**. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally.
For example, your program can make a call to the **sort()** function without knowing what algorithm the function actually uses to sort the given values. In fact, the underlying implementation of the sorting functionality could change between releases of the library, and as long as the interface stays the same, your function call will still work.
In C++, we use **classes** to define our own abstract data types (ADT). You can use the **cout** object of class **ostream** to stream data to standard output like this:

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

int main( )
{
        cout << "Hello C++" <<endl;
        return 0;
}
```

Access Labels Enforce Abstraction

In C++, we use access labels to define the abstract interface to the class. A class may contain zero or more access labels:

- Members defined with a public label are accessible to all parts of the program. The data-abstraction view of a type is defined by its public members.

- Members defined with a private label are not accessible to code that uses the class. The private sections hide the implementation from code that uses the type.

There are no restrictions on how often an access label may appear. Each access label specifies the access level of the succeeding member definitions. The specified access level remains in effect until the next access label is encountered or the closing right brace of the class body is seen.

Benefits of Data Abstraction:

Data abstraction provides two important advantages:

- Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.

- The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

By defining data members only in the private section of the class, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what affect the change may have. If data are public, then any function that directly accesses the data members of the old representation might be broken.

Data Abstraction Example

Any C++ program where you implement a class with public and private members is an example of data abstraction. Consider the following example:

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

class Adder{
  public:
    // constructor
    Adder(int i = 0)
    {
          total = i;
    }
    // interface to outside world
    void addNum(int number)
    {
          total += number;
    }
    // interface to outside world
    int getTotal()
```

```
    {
          return total;
    };
  private:
    // hidden data from outside world
    int total;
};
int main( )
{
          Adder a;

          a.addNum(10);
           a.addNum(20);
          a.addNum(30);

         cout << "Total " << a.getTotal() <<endl;
          return 0;
}
```

When the above code is compiled and executed, it produces the following result

```
Total 60
```

Data Encapsulation in C++

All C++ programs are composed of the following two fundamental elements:

- **Program statements (code):** This is the part of a program that performs actions and they are called functions.
- **Program data:** The data is the information of the program which affected by the program functions.

Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of **data hiding**.

Data encapsulation is a mechanism of bundling the data, and the functions that use them and **data abstraction** is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called **classes**. We already have studied that a class can contain **private, protected** and **public** members. By default, all items defined in a class are private. For example:

```
class Box
{
```

```
  public:
    double getVolume(void)
    {
      return length * breadth * height;
    }
  private:
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box
};
```

The variables length, breadth, and height are **private**. This means that they can be accessed only by other members of the Box class, and not by any other part of your program. This is one way encapsulation is achieved.

To make parts of a class **public** (i.e., accessible to other parts of your program), you must declare them after the **public** keyword. All variables or functions defined after the public specifies are accessible by all other functions in your program.

Making one class a friend of another exposes the implementation details and reduces encapsulation. The ideal is to keep as many of the details of each class hidden from all other classes as possible.

Data Encapsulation Example:

Any C++ program where you implement a class with public and private members is an example of data encapsulation and data abstraction. Consider the following example:

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

class Adder{
  public:
    // constructor
    Adder(int i = 0)
    {
          total = i;
    }
    // interface to outside world
    void addNum(int number)
    {
          total += number;
    }
    // interface to outside world
    int getTotal()
```

```
    {
          return total;
    };
  private:
    // hidden data from outside world
    int total;
};
int main( )
{
          Adder a;

           a.addNum(10);
           a.addNum(20);
           a.addNum(30);

           cout << "Total " << a.getTotal() <<endl;
          return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Total 60
```

Interfaces in C++ (Abstract Classes)

An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class.

The C++ interfaces are implemented using **abstract classes** and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.

A class is made abstract by declaring at least one of its functions as **pure virtual** function. A pure virtual function is specified by placing "= 0" in its declaration as follows:

```
class Box
{
  public:
    // pure virtaul function
    virtual double getVolume() = 0;
  private:
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box
```

```
};
```

The purpose of an **abstract class** (often referred to as an ABC) is to provide an appropriate base class from which other classes can inherit. Abstract classes cannot be used to instantiate objects and serves only as an **interface**. Attempting to instantiate an object of an abstract class causes a compilation error.

Thus, if a subclass of an ABC needs to be instantiated, it has to implement each of the virtual functions, which means that it supports the interface declared by the ABC. Failure to override a pure virtual function in a derived class, then attempting to instantiate objects of that class, is a compilation error.

Classes that can be used to instantiate objects are called **concrete classes**.

Abstract Class Example:

Consider the following example where parent class provides an interface to the base class to implement a function called **getArea()**:

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

// Base class
class Shape
{
public:
  // pure virtual function providing interface framework.
          virtual int getArea() = 0;
  void setWidth(int w)
  {
         width = w;
  }
  void setHeight(int h)
  {
         height = h;
  }
protected:
          int width;
          int height;
};

// Derived classes
class Rectangle: public Shape
{
```

```
        public:
         int getArea()
        {
                return (width * height);
        }
};

class Triangle: public Shape
{
        public:
         int getArea()
        {
                 return (width * height)/2;
         }
};

int main(void)
{
        Rectangle Rect;
        Triangle  Tri;

          Rect.setWidth(5);
         Rect.setHeight(7);
         // Print the area of the object.
        cout << "Total Rectangle area: " << Rect.getArea() << endl;

         Tri.setWidth(5);
           Tri.setHeight(7);
         // Print the area of the object.
         cout << "Total Triangle area: " << Tri.getArea() << endl;

         return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Total Rectangle area: 35
Total Triangle area: 17
```

C++ Files and Streams

So far, we have been using the **iostream** standard library, which provides **cin** and **cout** methods for reading from standard input and writing to standard output respectively.
This tutorial will teach you how to read and write from a file. This requires another standard C++ library called **fstream**, which defines three new data types:

Data Type	Description
ofstream	This data type represents the output file stream and is used to create files and to write information to files.
ifstream	This data type represents the input file stream and is used to read information from files.
fstream	This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files.

To perform file processing in C++, header files <iostream> and <fstream> must be included in your C++ source file.

Opening a File:

A file must be opened before you can read from it or write to it. Either the **ofstream** or **fstream** object may be used to open a file for writing and ifstream object is used to open a file for reading purpose only.

Following is the standard syntax for open() function, which is a member of fstream, ifstream, and ofstream objects.

```
void open(const char *filename, ios::openmode mode);
```

Here, the first argument specifies the name and location of the file to be opened and the second argument of the **open()** member function defines the mode in which the file should be opened.

Mode Flag	Description
ios::app	Append mode. All output to that file to be appended to the end.
ios::ate	Open a file for output and move the read/write control to the end of the file.

ios::in	Open a file for reading.
ios::out	Open a file for writing.
ios::trunc	If the file already exists, its contents will be truncated before opening the file.

You can combine two or more of these values by **OR**ing them together. For example if you want to open a file in write mode and want to truncate it in case it already exists, following will be the syntax:

```
ofstream outfile;
outfile.open("file.dat", ios::out | ios::trunc );
```

Similar way, you can open a file for reading and writing purpose as follows:

```
fstream  afile;
afile.open("file.dat", ios::out | ios::in );
```

Closing a File

When a C++ program terminates it automatically closes flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination.

Following is the standard syntax for close() function, which is a member of fstream, ifstream, and ofstream objects.

```
void close();
```

Writing to a File

While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an **ofstream** or **fstream** object instead of the **cout** object.

Reading from a File

You read information from a file into your program using the stream extraction operator (>>) just as you use that operator to input information from the keyboard. The only difference is that you use an **ifstream** or **fstream** object instead of the **cin** object.

Read & Write Example:

Following is the C++ program which opens a file in reading and writing mode. After writing information inputted by the user to a file named afile.dat, the program reads information from the file and outputs it onto the screen:

```
#include <fstream>
#include <iostream>
using namespace std;

int main ()
{

   char data[100];

   // open a file in write mode.
   ofstream outfile;
   outfile.open("afile.dat");

   cout << "Writing to the file" << endl;
   cout << "Enter your name: ";
   cin.getline(data, 100);

   // write inputted data into the file.
   outfile << data << endl;

   cout << "Enter your age: ";
   cin >> data;
   cin.ignore();

   // again write inputted data into the file.
   outfile << data << endl;

   // close the opened file.
   outfile.close();

   // open a file in read mode.
   ifstream infile;
   infile.open("afile.dat");

   cout << "Reading from the file" << endl;
   infile >> data;
```

```
         // write the data at the screen.
        cout << data << endl;

        // again read the data from the file and display it.
        infile >> data;
        cout << data << endl;

        // close the opened file.
         infile.close();

        return 0;
}
```

When the above code is compiled and executed, it produces the following sample input and output:

```
$./a.out
Writing to the file
Enter your name: Zara
Enter your age: 9
Reading from the file
Zara
9
```

Above examples make use of additional functions from cin object, like getline() function to read the line from outside and ignore() function to ignore the extra characters left by previous read statement.

File Position Pointers

Both **istream** and **ostream** provide member functions for repositioning the file-position pointer. These member functions are **seekg** ("seek get") for istream and **seekp** ("seek put") for ostream.

The argument to seekg and seekp normally is a long integer. A second argument can be specified to indicate the seek direction. The seek direction can be **ios::beg** (the default) for positioning relative to the beginning of a stream, **ios::cur** for positioning relative to the current position in a stream or **ios::end** for positioning relative to the end of a stream.
The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location. Some examples of positioning the "get" file-position pointer are:

```
// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg( n );

// position n bytes forward in fileObject
```

```
fileObject.seekg( n, ios::cur );

// position n bytes back from end of fileObject
fileObject.seekg( n, ios::end );

// position at end of fileObject
fileObject.seekg( 0, ios::end );
```

C++ Exception Handling

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,** and **throw**.

- **throw:** A program throws an exception when a problem shows up. This is done using a **throw** keyword.

- **catch:** A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.

- **try:** A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following

```
try
{
  // protected code
}
catch( ExceptionName e1 )
{
  // catch block
}
catch( ExceptionName e2 )
{
  // catch block
```

```
}
catch( ExceptionName eN )
{
  // catch block
}
```

you can list down multiple **catch** statements to catch different type of exceptions in case your **try** block raises more than one exception in different situations.

Throwing Exceptions

Exceptions can be thrown anywhere within a code block using **throw** statements. The operand of the throw statements determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs:

```
double division(int a, int b)
{
  if( b == 0 )
  {
    throw "Division by zero condition!";
  }
  return (a/b);
}
```

Catching Exceptions

The **catch** block following the **try** block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try
{
  // protected code
}
catch( ExceptionName e )
{
 // code to handle ExceptionName exception
}
```

Above code will catch an exception of **ExceptionName** type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows:

```
try
{
  // protected code
}
catch(...)
{
 // code to handle any exception
}
```

The following is an example, which throws a division by zero exception and we catch it in catch block.

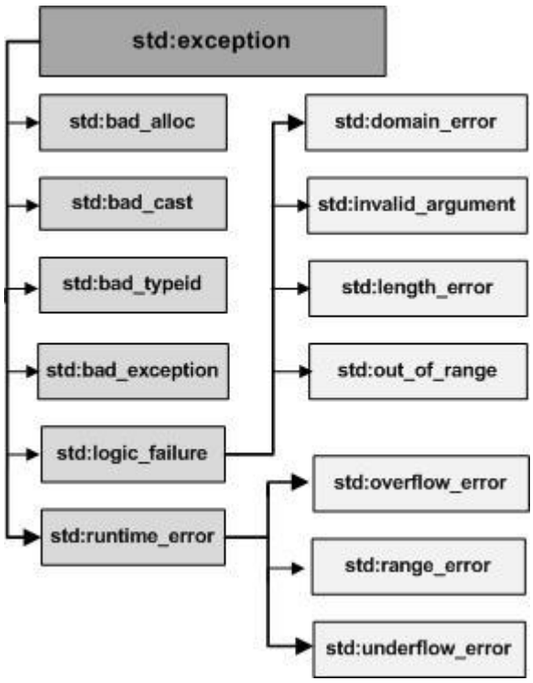
```
#include <iostream>
using namespace std;

double division(int a, int b)
{
  if( b == 0 )
  {
    throw "Division by zero condition!";
  }
  return (a/b);
}

int main ()
{
        int x = 50;
        int y = 0;
        double z = 0;

  try
 {
        z = division(x, y);
        cout << z << endl;
  }
catch (const char* msg)
{
    cerr << msg << endl;
  }

  return 0;
}
```

Because we are raising an exception of type **const char***, so while catching this exception, we have to use const char* in catch block. If we compile and run above code, this would produce the following result:

Division by zero condition!

C++ Standard Exceptions

C++ provides a list of standard exceptions defined in **<exception>** which we can use in our programs. These are arranged in a parent-child class hierarchy shown below:



Here is the small description of each exception mentioned in the above hierarchy:

Exception	Description
std::exception	An exception and parent class of all the standard C++ exceptions.
std::bad_alloc	This can be thrown by **new**.
std::bad_cast	This can be thrown by **dynamic_cast**.

std::bad_exception	This is useful device to handle unexpected exceptions in a C++ program
std::bad_typeid	This can be thrown by **typeid**.
std::logic_error	An exception that theoretically can be detected by reading the code.
std::domain_error	This is an exception thrown when a mathematically invalid domain is used
std::invalid_argument	This is thrown due to invalid arguments.
std::length_error	This is thrown when a too big std::string is created
std::out_of_range	This can be thrown by the at method from for example a std::vector and std::bitset<>::operator[]().
std::runtime_error	An exception that theoretically can not be detected by reading the code.
std::overflow_error	This is thrown if a mathematical overflow occurs.
std::range_error	This is occured when you try to store a value which is out of range.
std::underflow_error	This is thrown if a mathematical underflow occurs.

Define New Exceptions

You can define your own exceptions by inheriting and overriding **exception** class functionality. Following is the example, which shows how you can use std::exception class to implement your own exception in standard way

```
#include <iostream>
#include <exception>
using namespace std;
```

```
struct MyException : public exception
{
  const char * what () const throw ()
  {
        return "C++ Exception";
  }
};

int main()
{
        try
        {
                throw MyException();
        }
        catch(MyException& e)
        {
                std::cout << "MyException caught" << std::endl;
                std::cout << e.what() << std::endl;
        }
        catch(std::exception& e)
        {
                //Other errors
        }
}
```

This would produce the following result:

```
MyException caught
C++ Exception
```

Here, **what()** is a public method provided by exception class and it has been overridden by all the child exception classes. This returns the cause of an exception.

Namespaces in C++

Consider a situation, when we have two persons with the same name, Zara, in the same class. Whenever we need to differentiate them definitely we would have to use some additional information along with their name, like either the area if they live in different area or their mother or father name, etc.

Same situation can arise in your C++ applications. For example, you might be writing some code that has a function called xyz() and there is another library available which is also having same function xyz(). Now the compiler has no way of knowing which version of xyz() function you are referring to within your code.

A **namespace** is designed to overcome this difficulty and is used as additional information to differentiate similar functions, classes, variables etc. with the same name available in different libraries. Using namespace, you can define the context in which names are defined. In essence, a namespace defines a scope.

Defining a Namespace

A namespace definition begins with the keyword **namespace** followed by the namespace name as follows:

```
namespace namespace_name
{
  // code declarations
}
```

To call the namespace-enabled version of either function or variable, prepend the namespace name as follows

```
name::code;  // code could be variable or function.
```

Let us see how namespace scope the entities including variable and functions:

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

// first name space
namespace first_space{
  void func()
{
          cout << "Inside first_space" << endl;
  }
}

// second name space
namespace second_space
{
           void func()
          {
                    cout << "Inside second_space" << endl;
           }
}

int main ()
```

```
{

        // Calls function from first name space.
         first_space::func();

        // Calls function from second name space.
        second_space::func();

  return 0;
}
```

If we compile and run above code, this would produce the following result:

```
Inside first_space
Inside second_space
```

The using directive

You can also avoid prepending of namespaces with the **using namespace** directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace. The namespace is thus implied for the following code

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

// first name space
namespace first_space
{
        void func()
        {
                cout << "Inside first_space" << endl;
        }
}
// second name space
namespace second_space{
  void func()
{
    cout << "Inside second_space" << endl;
  }
}


using namespace first_space;
int main ()
```

```
{

  // This calls function from first name space.
  func();

  return 0;
}
```

The using directive

You can also avoid prepending of namespaces with the **using namespace** directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace. The namespace is thus implied for the following code

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

// first name space
namespace first_space{
  void func(){
    cout << "Inside first_space" << endl;
  }
}
// second name space
namespace second_space{
  void func(){
    cout << "Inside second_space" << endl;
  }
}
using namespace first_space;
int main ()
{

  // This calls function from first name space.
  func();

  return 0;
}
```

If we compile and run above code, this would produce the following result:

Inside first_space

The using directive can also be used to refer to a particular item within a namespace. For example, if the only part of the std namespace that you intend to use is cout, you can refer to it as follows:

```
using std::cout;
```

Subsequent code can refer to cout without prepending the namespace, but other items in the **std** namespace will still need to be explicit as follows

```
#include <iostream>
using std::cout;

int main ()
{

  cout << "std::endl is used with std!" << std::endl;

  return 0;
}
```

If we compile and run above code, this would produce the following result:

```
std::endl is used with std!
```

Names introduced in a **using** directive obey normal scope rules. The name is visible from the point of the **using** directive to the end of the scope in which the directive is found. Entities with the same name defined in an outer scope are hidden.

Discontiguous Namespaces

A namespace can be defined in several parts and so a namespace is made up of the sum of its separately defined parts. The separate parts of a namespace can be spread over multiple files.

So, if one part of the namespace requires a name defined in another file, that name must still be declared. Writing a following namespace definition either defines a new namespace or adds new elements to an existing one

```
namespace namespace_name
 {
  // code declarations
}
```

Nested Namespaces

Namespaces can be nested where you can define one namespace inside another name space as follows:

```
namespace namespace_name1 {
  // code declarations
  namespace namespace_name2 {
    // code declarations
  }
}
```

You can access members of nested namespace by using resolution operators as follows:

```
// to access members of namespace_name2
using namespace namespace_name1::namespace_name2;

// to access members of namespace:name1
using namespace namespace_name1;
```

In the above statements if you are using namespace_name1, then it will make elements of namespace_name2 available in the scope as follows:

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

// first name space
namespace first_space{
  void func()
{
    cout << "Inside first_space" << endl;
  }
  // second name space
  namespace second_space
{
    void func()
{
          cout << "Inside second_space" << endl;
    }
  }
}
using namespace first_space::second_space;
int main ()
{

  // This calls function from first name space.
  func();

  return 0;
}
```

If we compile and run above code, this would produce the following result:

```
Inside second_space
```

Question Bank – C++ using OOP concepts
BCACAC 157

UNIT I

Two Mark Questions

1. What is Object Oriented Programming ?
2. Give any two difference between object oriented programming and procedural programming.
3. List any four features of Object Oriented Programming.
4. What is data encapsulation and Data abstraction ?
5. How data hiding concept is implemented in C++ ?
6. What is Polymorphism ? Give example.
7. What do you mean by message passing ?
8. What is dynamic binding ?
9. Mention any four advantages of OOPs .
10. Mention any four application areas of OOPs .
11. Give syntax of cin and cout operators in C++.
12. Define i) Token ii) Identifier
13. What are the rules for declaring identifier ?
14. Differentiate Basic and Derived data types.
15. What is user defined data type ? Give two examples.
16. What are the uses of void datatype in C++ ?
17. Give two methods of declaring symbolic constants in C++.
18. What do you mean by pointer constant ? Give example.
19. What is pointer to the constant ? Give example.
20. What do you mean by dynamic initialisation of variables in C++? Give example
21. What is reference variable ?
22. What is scope resolution operator ?
23. List any four operators in C++ not found in C.
24. How to allocate memory to pointer using new operator ?
25. What are manipulators ? Give example.
26. Give syntax and usage of setw and endl manipulators.
27. What is implicit type conversion ?
28. What is operator precedence and associatively of operators.
29. Give the syntax of switch statement.
30. Compare while and do..while loops.
31. What is the output of the following statement:
    cout<<"value of expression is"<<((a!=b)?pow(a,2) : pow(b,2));
    (assume a=5, b=6)

32. What is the output of following code ?

```
void main()
{
int a = 20;
int &n = a;
n=a++;
a=n++;
cout<<a <<" ,"<<n<<endl;
}
```

33. What is output of following program ?

```
int a = 1;
void main()
{
        int a = 100;
                {
                        int a = 200;
                        {
                                int a = 300;
                                cout<<a<<",";
                        }
                        cout<<a<<",";
                }
                cout<<a<<",";
                cout<<::a<<",";
}
```

34. What is the output of the following code?

```
#include<iostream.h>
int main()
{
for(int ii=0;ii<3;++ii)
{
switch(ii)
        {
          case 0:cout<<"zero ";
          case 1:cout<<"one ";continue;
          case 2:cout<<"two ";break;
        }
  }
        return 0;
        }
```

Short answer questions **3-4 marks**

1. Explain any two concepts/features of OOP. 4
2. Write a note on application and benefits of OOP. 4
3. Explain basic data types supported by C++. 4
4. Explain any two types of operators in C++ 4
5. What is enumerated type ? Explain 4
6. Write a note on pointer declarations in C++. 3
7. Explain increment and decrement operators with example 4
8. Explain any two special operators in C++ 3
9. How concept of reference variable can be used in function call ? Explain 4
10. Write a note on type conversions in C++. 3
11. Explain precedence and associatively of the operators with reference following expression

    a+b*c/d-(e+f)/d 4
12. Explain for loop structure with example 4
13. Explain the use of break and continue statements in C++ 4

Long Answer questions

1. Explain any four features of OOP. 6
2. Explain applications and advantages of OOP. 5
3. Explain classification of data types in C++ 6
4. Explain arithmetic .relational and logical operators in C++ 6
5. Explain various bitwise operators available in C++ 6
6. What are constants? How they are classified? Give example for each. 6
7. Explain three methods of defining symbolic constants in C++. 6
8. Write a note on different types of expressions in C++ 5
9. Explain any two loop control structures with syntax and example. 6

UNIT II

Two mark Questions

10. What is function prototype ? Give example.
11. What is inline function ?
12. Give two situations in which inline function may not work.
13. What is default argument ? When it is needed ?
14. What do you mean by pass by reference ?
15. What do you mean by pass by value ?
16. What is function overloading ?
17. What is constant argument ? Give example

18. Give the general form of class definition.
19. Differentiate class and structure in C++.
20. What is object ? How it is declared in C++ ?
21. What is class ? How it accomplish data hiding ?
22. Differentiate private and public members of class.
23. Explain the concept of data hiding in C++ with an example.
24. How do you define member function outside the class ? Give example.
25. Differentiate static data member and non static data member in a class.
26. When you use static data members. Given an example.
27. Compare static member function with normal member function of class.
28. How do you access private member function of a class ?
29. What is friend function ? Why it is required.
30. Give any two properties of friend function.
31. What are constant member functions?

Short Answer Questions

1. Compare C and C++ structures. 3
2. Explain the concept of overloading a function in C++ with example. 4
3. Explain concept of default arguments with Example. Also mention the rules to be followed while assigning default values. 4
4. Explain how to pass arrays to functions with an suitable example. 4
5. Explain the usage of static data members with example. 4
6. What is friend function ? What are the merits and demerits of using friend function. 4
7. Write a note on nesting of classes. 4
8. Write a C++ code to illustrate pointers to data members of a class. 4
9. Write a note on objects as functional arguments. 4

Long Answer questions

1. Explain different ways of defining member functions of a class with an example. 5
2. What is friend function ? What are the merits and demerits of using friend function. 5
3. Write a program using friend functions to add , subtract and multiply two complex numbers. 6
4. With proper example, explain how to pass and return an object to/from a function.
5. Explain the concept of pointer to object and pointer to member functions. 5
6. Explain how objects are passed as argument to the function with an example. 5

UNIT III

Two mark Questions

11. What is constructor ?
12. How constructor is invoked ? Give example.
13. Differentiate default constructor and constructor with default arguements.
14. How do you define parameterised constructor ?
15. How to define constructor function inline ? Give example.
16. Distinguish between following two statements
    time T2(T1);
    time T2 =T1; Where T1 and T2 are objects of time class.
17. Give any two features of constructors.
18. How do you define copy constructor ?
19. Why copy constructor takes reference variable as argument ?
20. What is destructor ? How do you define it.?
21. What do you mean by overloading a operator ?
22. Give the general form of operator function.
23. Give any two rules of overloading operator.
24. List any four operators that can not be overloaded.
25. List the operators that can not be overloaded by friend function.
26. Why assignment operator can not be overloaded by friend function ?
27. What is the use of keyword ***operator*** in C++? Give example.
28. What is constant object ? How it is declared.
29. What is meant by casting operator and write the general form of overloaded casting operator?
30. What is class conversion function ?
31. How to define conversion function for class to basic type conversion ?
32. What is difference between overloading unary operator using member function & friend function ?

23. What is the output of the following code?

```
#include<iostream.h>
int count=0;
class obj
{
        public :
                obj(){count++;}
                ~obj(){count--;}
};
        int main()
                {
                        obj A,B,C,D,E;
                        obj F;
```

```
                                        {
                                        obj G;
                                        }
                                        cout<<count;
                                        return 0;
                                    }
```

24. What is the output of the following code?

```
#include<iostream.h>
class obj
{
public :
obj(){cout<<"in ";}
~obj(){cout<<"out ";}
};
int main()
{
obj A,B;
{
obj D;
}
obj E;
return 0;
}
```

Short Answer Questions.

1. What are the characteristics of constructor ? 4
2. What are the different ways of calling constructor ? Explain with example. 3
3. What do you mean by dynamic initialisation of objects ? Explain 4
4. Write a note on copy constructors . 4
5. What is dynamic constructor ? Explain. 3
6. What are the rules for overloading a operator ? 4
7. How do you overload a unary operator using member function ? Explain with example. 4
8. How do you overload a unary operator using friend function ? Explain with example. 4
9. Define a class String. Using overloaded operator = = check whether two strings are equal or not . 4

Long Answer Questions

1. Define class string having data member for holding string data and its length. Include all types of constructors to initialise objects. Write a program to test your class to do the following
   i) To copy one object to another
   ii) two join two strings using overloaded + operator 6
2. Explain how multiple constructors are defined in a class with example. 5
3. What is meant by constructor overloading? Explain with code example. 5
4. Explain with example how two dimensional array is constructed and destroyed using dynamic constructor and destructor. 6
4. Construct a class INTEGER containing a integer data member and write a program to overload four arithmetic operators so that they operate on the objects of INTEGER. 6
5. How do you overload a binary operator using member function ? Explain with example. 5
6. Write a program to generate fibonacci numbers by overloading ++ (post increment) 5
7. How do you overload a binary operator using member function ? Explain with example. 5
8. Explain basic to class type conversion with an example. 6
9. Explain class to basic type conversion with an example. 5
10. Explain one class another class conversion with an example. 6

UNIT IV

Two mark Questions

24. Define base class and Derived class.
25. List four types of inheritance.
26. What is difference between private and protected access specifies ?
27. When do we use protected access specifier in C++ ?
28. What is the difference between multiple inheritance and multilevel inheritance?
29. What is the difference between overloading and overriding?
30. Give the general form of derived class declaration.
31. What is virtual base class ?
32. When do we need virtual functions ?
33. What is abstract class ?
34. How to define a derived class constructor ?
35. What is container class ?
36. What is static binding or early binding?
37. What is polymorphism? What are its types?
38. What is compile time polymorphism ?
39. What is late binding ?

40. What is this pointer ?
41. What is virtual function ?
42. What is pure virtual function ?
43. What are the implications of making a function pure virtual ?
44. What is wrong in the following code?

```
#include<iostream.h>
    class Base
    {
    public :
    Base(){};
    virtual ~Base(){};
    };
    class Derived : protected Base
    {
    public:
    virtual ~Derived(){};
    };
    int main()
    {
    Base *pb = new Derived();
    return 0
}
```

22. What is the output of the following code?

```
#include<iostream.h>
class professor{public:professor(){cout<<"professor ";};};
class researcher{public: researcher(){cout<<"researcher ";};};
class teacher: public professor{public: teacher(){cout<<"teacher ";};};
class myprofessor: public teacher, public virtual researcher
{public:myprofessor(){cout<<"myprofessor ";};};
int main()
{
        myprofessor obj;
        return 0;
}
```

23. What is the output of the following code?

```
#include<iostream.h>
class Parent
{
public:
Parent(){Status();}
virtual ~Parent() { Status();}
virtual void Status(){cout<<"Parent ";}
};
class Child: public Parent
{
public:
```

```
        Child(){Status();}
        virtual ~Child() { Status();}
        virtual void Status(){cout<<"Child ";}
};
void main()
{
        Child c;
}
```

Short Answer Questions

1. What are the different types of inheritance supported by C++ ?Explain 4
2. Explain single inheritance with an example. 4
3. How ambiguity is resolved in multiple inheritance using virtual base class ? Explain 4
4. Explain order of execution of constructors in inheritance with an example 4
5. How containership differ from inheritance ? Explain. 3
6. How do the properties of following two derived classes differ ? 3
   i) class D1 : private B { ---------- }
   ii) class D2: public B { ---------- }
7. Write a note on compile time and run time polymorphism. 3
8. Explain how pointer to objects are used in a C++ program. 4
9. What is this pointer ? explain its importance in C++. 4
10. What are the basic rules for function to be virtual ? 4
11. Write a note on virtual constructors and destructors 4
12. Consider the following program

```
class Base
{
int static i;
        public:
        Base(){}
};
        class Sub1: public virtual Base{};
        class Sub2: public Base{};
        class Multi: public Sub1, public Sub2 {};
        void main()
        {
                Multi m;
        }
```

In the above program, how many times Base's constructor will be called? Explain 3

Long Answer Questions (5 to 7 marks)

1. Explain visibility of private , protected and public members in different modes of inheritance. 6
2. Explain multi level inheritance with an example. 5
3. Explain multiple inheritance with an example. 5
4. Explain hybrid inheritance with an example. 5
5. Explain hybrid inheritance with an example. 5
6. Explain protected mode of inheritance with example. 5
7. Explain private mode of inheritance with example
8. Explain public mode of inheritance with example. 5
9. Explain how pointers to derived class are used in a program with an example. 5
10. Explain how runtime polymorphism is achieved using virtual functions with an example. 6

Note:

All the practical list programs are considered to be part of question bank. PART -A programs can be considered for 5 marks , PART-B programs for 6 marks and PART-C programs for 7 marks.

BCACAC 157

Credit Based Second Semester B.C.A. Degree Examination, April/may 2015
(New Syllabus) (2012-13 Batch Onwards)
OBJECT ORIENTED PROGRAMMING USING C++

Time: 3 Hours **Max. Marks : 80**

Note: Answer **any ten** questions from **Part-A** and **one full** question from each Unit of **Part-B**

PART-A

1

a) What is dynamic binding?
b) What are manipulators? Give an example.
c) Give the syntax of switch statement.
d) Differentiate class and structure in C++.
e) Give any two properties of a friend function.
f) List any four operators that cannot be overloaded.
g) What is a constant object? How it is declared?
h) What is destructor? How do you define it in C++?
i) When do we use protected access specifier in C++?
j) Give the general form of a derived class declaration.
k) What is virtual function?

PART-B
Unit-I

2

a) What are the advantages of OOPs?
b) Explain three methods of defining symbolic constants in C++.
c) Explain the use of break and continue statement in C++. (5+6+4)

3

a) Explain the classification of data types in C++.
b) Explain precedence and associativity of the operators with reference to the following expression a+b*c/d-(e+f)/g.
c) Explain the for loop structure with a suitable example. (5+5+5)

Unit-II

4

a) Explain the different ways of defining member functions of a class with an example.
b) What is friend function? What are the merits and demerits of using friend function.

c) Write a note on function prototyping. (5+5+5)

5

a) Explain how to pass arrays to functions with a suitable example.
b) Write a note on objects as functional arguments.
c) With proper example, explain how to pass and return an object to/from a function. (5+4+6)

Unit-III

6

a) How do you overload a binary operator using member function? Explain with example.
b) Write a note on copy constructor.
c) Construct a class INTEGER containing an integer data member and write a program to overload four arithmetic operators so that they operate on the objects of INTEGER. (5+4+6)

7

a) What are the different ways of calling constructor? Explain with an example.
b) Explain class to basic type conversion with an example.
c) Define class string having data member for holding string data. Include all types of constructors to initialize objects. Write a program to test your class to do the following:
    i. To copy one object to another
    ii. Two join two strings using overloaded + operator. (4+5+6)

UNIT-IV

8

a) What is this pointer? Explain its importance in C++ with an example.
b) Explain how pointers to objects are used in a C++ program.
c) Explain private mode of inheritance with example. (5+5+5)

9

a) Explain multiple inheritance with an example.
b) Write a note on compile time and runtime polymorphism.
c) What are the basic rules for a function to be virtual? (5+4+6)

****