GOVERNMENT ARTS AND SCIENCE COLLEGE
PERAVURANI-614804

16SCCCS2- PROGRAMMING IN C++

DESCRIPTIVE QUESTIONS & ANSWERS

DEPARTMENT OF COMPUTER SCIENCE

Class: I B.Sc., Computer Science

COMPLIED BY,

Mrs. S.Jamuna M.Sc., M.Phil, B.Ed.,
Guest Lecturer, Dept. of Comp. Sci.
GASC, Peravurani-614804.

PROGRAMMING IN C++

DESCRIPTIVE QUESTIONS & ANSWERS

1. BASIC CONCEPTS OF OBJECT-ORIENTED PROGRAMMING

These include:

- Objects
- Classes
- Data abstraction and encapsulation
- Inheritance
- Polymorphism
- Dynamic binding
- Message passing

Objects:

- Objects are the basic run-time entities in an object oriented programming.
- It is an instance of class.
- Each instance of an object can hold its own relevant data.
- They may represent a person, a place, a bank account or any item that the program has to handle.
- Each object contains data (data members) and code (member functions) to manipulatethe data.

Object: STUDENT
DATA: Name Roll No Mark 1, Mark 2
FUNCTIONS Total Average Display

Classes:

It is a representation (or) blueprint of an object.

- The entire set of data and code of an object can made a user defined data type with the help of a class.
- Classes are user-defined data types and behave like built-in data types of
- Programming language and they are known as Abstract Data Type.
- For Example: If Fruit has been declared as a class, then the statement
  Fruit mango; will create an object **mango** belonging to the class **Fruit**

Data Abstraction and Encapsulation:

Encapsulation:

The wrapping up of data and functions into a single unit is known as encapsulation**.**

- The data is not accessible to the outside world and only those functions which are wrapped in the class can access it.
- These functions provide interface b/n the object's data and the program. This insulation of the data from direct access by the program is called *data hiding or information hiding*.

Abstraction:

It represents the act of representing the essential features without including the background details or explanations.

- Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and cost and functions to operate on these attributes.
- The attributes are called data members and functions are called member functions or methods.

Since the classes use the concept of data abstraction, they are known as **Abstract Data Types (ADT).**

Inheritance:

It is the process by which objects of one class acquire the properties of objects of another class.

- It supports the concept of hierarchical classification. For example, the 'student' and the 'employee' is a part of the class 'person'.

This concept provides the idea of *reusability*. This means that we can add additional features to an existing class without modifying it.

Types of Inheritance

- Single inheritance
- Multiple inheritance
- Multilevel inheritance
- Hierarchical inheritance
- Hybrid inheritance

Advantages

- It offers code reusability.
- It is a technique of design and code sharing.

Polymorphism:

It means the ability to take more than one form. An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation.

- For example the operation addition will generate sum if the operands are numbers whereas if the operands are strings then the operation would produce a third string by concatenation. The process of making an operator to exhibit different behaviors in different instances is known as *operator overloading.*
- A single function name can be used to handle different types of tasks based on the number and types of arguments. This is known as *function overloading.*

Dynamic Binding:

Binding refers to the linking of a procedure call to the code to be executed in response to the call. *Dynamic Binding (late binding)* means the code associated with the given procedure call is not known until the time of the call at run-time

Message Passing:

The process of programming in OOP involves the following basic steps:

- Creating classes that define objects and their behavior
- Creating objects from class definitions

 Establishing communication among objects

A *message* for an object is request for execution of a procedure and therefore will invoke a function (procedure) in the receiving object that generates the desired result. *Message passing* nvolves specifying the name of the object, the name of the function (message) and the information to be sent. E.g.: employee.salary(name); **Object :** employee, **Message**: salary

2. FEATURES OF OOP

- Emphasis is on data rather than procedure.
- Programs are divided into as objects.
- Data structures are designed such that they character the objects.
- Functions that operate on the data of an object are tied together in the data structure.
- Data In hidden and cannot be accessed by the external function.
- Object may communicate with each other through function.
- New data and functions can be easily added whenever necessary.
- Follows bottom up approach in program design.

3. STRUCTURE OF C++ PROGRAM

Include Files
Class Definition
Member Function Definitions
Main Function

Include file

- In C++ number of functions, classes and variables are defined and stored in a special file called **header file**.
- There are so many header files available.
- Each header file has related functions, classes and data.
    - If we want to use the predefined functions, classes and variables and data the appropriate header file should be included at the beginning of the program. E.g.: iostream.h, string.h, conio.h. #include<header_file_name.>

Class Definition

It is a user defined data type having defined functions and data.

```
Class test
    {
    }
```

Member function definition

Member function can be defined inside or outside the class.

```
Void add ()
{ }
```

Main function

The execution of the program begins from void main() function. The parentheses following the word main are the distinguishing feature of the function, without the parentheses the compiler would think that main referred to a variable (or) some other program element. Where void proceeding the function main () indicates that this particular function does not have a return value.

```
void main()
{
}
```

4. C++ OPERATORS

a. Arithmetic operator

b. Relation operator

c. logical operator

d. Assignment operator

e. Conditional operator

f. Increment and Decrement Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators. Generally, there are six type of operators: Arithmetical operators, Relational operators, Logical operators, Assignment operators, Conditional operators, Comma operator

Arithmetical operators

Arithmetical operators +, -, *, /, and % are used to performs an arithmetic (numeric) operation.

Operator Meaning

+ Addition

- Subtraction

* Multiplication

/ Division

% Modulus

You can use the operators +, -, *, and / with both integral and floating-point data types. Modulus or remainder % operator is used only with the integral data type.

Relational operators

The relational operators are used to test the relation between two values. All relational operators are binary operators and therefore require two operands. A relational expression returns zero when the relation is false and a non-zero when it is true. The following table shows the relational operators.

Relational Operator

< Less than

<= Less than or equal to

== Equal to

> Greater than

>= Greater than or equal to

! = Not equal to

Logical operators

The logical operators are used to combine one or more relational expression. The logical operators are

Operators Meaning

|| OR

&& AND

! NOT

Assignment operator

The assignment operator '=' is used for assigning a variable to a value. This operator takes the

expression on its right-hand-side and places it into the variable on its left-hand-side. For example:

m = 5;

The operator takes the expression on the right, 5, and stores it in the variable on the left, m.

x = y = z = 32;

This code stores the value 32 in each of the three variables x, y, and z. In addition to standard assignment operator shown above, C++ also support compound assignment operators.

Compound Assignment Operators

Operator Example Equivalent to

+ = A + = 2 A = A + 2

- = A - = 2 A = A – 2

% = A % = 2 A = A % 2

/= A/ = 2 A = A / 2

***= A * = 2 A = A * 2**

Increment and Decrement Operators

C++ provides two special operators viz '++' and '--' for incrementing and decrementing the value of avariable by 1. The increment/decrement operator can be used with any type of variable but it cannot be used with any constant. Increment and decrement operators each have two forms, pre and post.

Conditional operator

The conditional operator ?: is called ternary operator as it requires three operands. The format of the conditional operator is :

Conditional_ expression ? expression1 : expression2;

If the value of conditional expression is true then the expression1 is evaluated, otherwise expression2 is evaluated. int a = 5, b = 6; big = (a > b) ? a : b; The condition evaluates to false, therefore big gets the value from b and it becomes 6.

5. DATA TYPES

Data type defines size and type of values that a variable can store along with the set of operations that can be performed on that variable. C++ provides built-in data types that correspond to integers, characters, floating-point values, and Boolean values. There are the seven basic data types in C++ as shown below:

Names	Description	Size	Range
char	Single text character or small integer. Indicated with single quotes (’a’, ’3’).	1 byte	signed: -128 to 127 unsigned: 0 to 255
int	Larger integer.	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean (true/false). Indicated with the keywords true and false.	1 byte	Just true (1)or false (0).
double	“Doubly” precise floating point number.	8 bytes	+/-1.7e +/-308 (15 digits

Type Meaning

a) char(character) holds 8-bit ASCII characters

b) int (Integer) represent integer numbers having no fractional part

c) float (floating point) stores real numbers in the range of about 3.4x10–38to 3.4x10 38,with a precision of seven digits.

d) Double (Double floating point) Stores real numbers in the range from 1.7x10 –308 to1.7x10308 with a precision of 15 digits.

e) bool(Boolean) can have only two possible values: true and false.

f) Void Valueless

C++ allows certain of the basic types to have modifiers preceding them. A modifier alters the meaning of the base type so that it more precisely fits the needs of various situations. The data type modifiers are: signed, unsigned, long and short This figure shows all combinations of the basic data types and modifiers along with their size and range for a 16-bit word machine.

6. CONTROL STRUCTURES (STATEMENTS)

(a) Conditional statements

(b) Loop statements

Conditional statements

This statements check the condition first and then execute the block statements. They have two statements:

If Statement

Switch Statements

Simple If Statement.

This statements check the condition first, if the condition is ture , then execute the block statements. If false, exit from if statement.

Syntax: If(condition)

{ statement(s);

}

if.. Else Statement.

This statements check the condition , if the condition is ture , then execute the one block statements. If false, another block of statement.

Syntax: If(condition)

{ statement(s)-I;

}

else

{ statement(s)-II;

}

Switch Statements

Select a block of statements to execute from the several block of statements base on the condition

Syntax:

switch(Exp) {

case 1:

statements-I;

break; }

7. LOOP STATEMENT

The loop statement executes the set of statements repeatedly until the condition is true. They are three statements.

• For statement

• While statement

• Do … while statement

For statement

Syntax

For(initialization; condition; increment;)

{

Statements;

}

While statement

while(condition)

{ statements;

}

Do … while statement

Syntax

do

{

```
Statements;
} while(condition);
```

8. C++ FUNCTIONS

A function is subprogram that contain set of statements. It perform a specific task.. It can be invoked from other parts of the program. Functions help to reduce the program size when same set of instructions are to be executed again and again.

Function declaration — prototype:

A function has to be declared before using it, in a manner similar to variables and constants. A function declaration tells the compiler about a function's name, return type, and parameters and how to call the function.

The general form : return_type function_name(parameter list);

Function definition

The function definition is the actual body of the function. The function definition consists of two parts namely, function header and function body.

The general form of a C++ function definition is as follows:

```
return_type function_name( parameter list )
{
body of the function
}
```

Here, **Return Type**: A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.

Function Name: This is the actual name of the function.

Parameters: A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

Function Body: The function body contains a collection of statements that define what the function does.

Calling a Function

To use a function, you will have to call or invoke that function. To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value.

Example program

A c++ program calculating factorial of a number using functions

```
#include<iostream.h>
#include<conio.h>
int factorial(int n); //function declaration
int main(){
int no, f;
cout<<"enter the positive number:-"; OOPS with C++ 21 cin>>no;
f=factorial(no); //function call
cout<<"\nThe factorial of a number"<<no<<"is"<<f;
return 0;
}
int factorial(int n) //function definition
{ int i , fact=1;
for(i=1;i<=n;i++)
{ fact=fact*i;
}
return fact; }
```

9. INLINE FUNCTIONS

An inline function is a function that is expanded inline at the point at which it is invoked, instead of actually being called. when a function is expanded inline, none of those operations occur. Although expanding function calls in line can produce faster run times, it can also result in larger code size because of duplicated code.

A function can be defined as an inline function by prefixing the keyword inline to the function header as given below:

inline function header

```
{
function body
}
```

Example:

```
// A program illustrating inline function
#include<iostream.h>
#include<conio.h>
inline int max(int x, int y){
if(x>y)
return x;
else
return y;
}
int main( ) {
int a,b;
cout<<”enter two numbers”;
cin>>a>>b;
cout << "The max is: " <<max(a,,b) << endl;
return 0;
}
```

10. FUNCTION OVERLOADING

Functions having the same name but with different signature is known as function overloading.

- A signature is a combination of a function name and its parameters.
- Parameter is differentiated by

1. Number of arguments

Sum(int, int);

Sum(int, int, int);

2. Type of arguments

Square(int);

Square(double);

3. Order of arguments

Sum(int, float);

Sum(float, int);

 It is commonly used to perform similar task, but on different data types.

 For example many functions in math library are overloaded for different numeric data types.

11. DIFFERENCE BETWEEN C & C++ LANGUAGE

Sl. No	C Programming	C++ Programming
1	C Language is not object oriented	C++ is object oriented
2	C program may be executed in C++ Compiler	C++ program cannot be executed in C Compiler.
3.	When function takes no parameters Void is given.	When function takes no parameter Void is optional.
4.	Function prototype is optional in C	Functional prototype is must in C++
5.	Function return type is optional in C	Function return type is must in C++
6.	Local variables can be declared only at the start of a program.	Local variables can be declared anywhere before they are used.
7.	C is not supporting Boolean data type.	C++ supports Boolean data type.
8.	C is not supporting Class data type.	C++ supports Class data type.
9.	Data hiding is not possible in C	Data hiding is possible in C++
10.	Data are not secure.	Data are more secure.
11.	There is a limitation on the length of identifiers	C++ does not put any limit on the identifiers.
12.	C does not support reference variables	C++ support reference variables
13.	C does not support Scope resolution operator.	C++ Supports scope resolution operator.
14.	Printf() in C requires type of variables to be printed.	Cout in C++ does not require type of variables to be printed.
15.	C does not support passing default value to function parameter,	C++ has a facility of passing default values to function parameter.

12. CLASS AND OBJECT

a) Class

A class is a user defined data type. It is a template of an object. A class contain data member and member function. When you define a class with declare the data and function. the data and function of a class are called members of the class.

The general form of class s:

class class-name {

access-specifier:

Declaration data ;

Declaration of functions

// ...

} object-list;

The object-list is optional. If present, it declares objects of the class. Here, access-specifier is one of these three C++ keywords:

public ,private, protected

Private member functions and data can not be accessed out side of class and only accessed within a class. The public access_specifier allows functions or data to be accessible to other parts of your program. The protected access_specifier is needed only when

inheritance is involved.

Example:

#include<iostream.h>

#include<conio.h>

Class myclass

{// class declaration

// private members to myclass

int a;

public:

// public members to myclass

void set_a(intnum);

int get_a( );

};

b) Object

An object is a runtime entity in oops. An object is said to be an instance of a class. Defining an object is similar to defining a variable of any data type: Defining objects in this way means creating them. This is also called instantiating them. Once a Class has been declared, we can create objects of that Class by using the class Name like any other built-in type variable as shown:

className objectName

Example

```
void main( ) {
myclass ob1, ob2; //these are object of type myclass
// ... program code
}
```

Accessing Class Members

The main() cannot contain statements that access class members directly. Class members can be accessed only by an object of that class. To access class members, use the dot (.) operator. The dot operator links the name of an object with the name of a member. The general form of the dot operator is shown here:

object. member

Example

ob1.set_a(10);

The private members of a class cannot be accessed directly using the dot operator, but through the member functions of that class providing data hiding. A member function can call another member unction directly, without using the dot operator.

Example: C++ program to find sum of two numbers using classes

```
#include<iostream.h>
#include<conio.h>
class A{
int a,b,c;
public:
void sum(){
cout<<"enter two numbers";
cin>>a>>b;
c=a+b;
cout<<"sum="<<c;
```

```
}
};
int main(){
A u;
u.sum();
getch();
return(0); }
```

13. FRIEND FUNCTION

In a class, private member variables can be accessed only by the member functions in that class. Friend function can be used to access all private and protected members of the class for which it is a friend. Friend function should be a non-member class.

General form: friend return_type function_name(args);

Characteristics:

- The keyword friend can only be used inside the class.
- More than one friend function can be declared in a class.
- A function can be friend to more than one class.
- Friend function definition should not contain the scope operator.

Example:

```
#include<iostream.h>
#include<conio.h>
class sample
{
private:
int a,b,c;
public:
void getdata();
friend void display(sample); //friend function declaration
};
void sample::getdata()
{
cin>>a>>b>>c;
}
```

```
void display(sample s)
{
cout<<s.a<<s.b<<s.c;
}
void main()
{
sample s; s.getdata(); display(s); getch(); }
```

14. CONSTRUCTORS

Constructor is a special member function with the same name as its class name. Constructor functions are invoked automatically when an object for a class is created.

Uses:

- It is used to initialize the object.
- It usually provides initial value for the data member of the object.

Characteristics of Constructor:

- A constructor has the same name as the class itself.
- It is invoked automatically when the objects are created.
- It should be defined public.
- They can have default argument.
- It can be overload.
- It has no return type even void.
- We cannot refer to their address.
- They cannot be inherited.
- They cannot be virtual.

Types of Constructor

- Default Constructor / Constructor with no argument
- Parameterized Constructor
- Multiple Constructor / Overloading Constructor
- Copy Constructor

Default Constructor (or) Constructor with no argument

A Constructor that accepts no parameter is called Default Constructor.

```
Constructor-name()
{ }
```

Parameterized Constructor

The constructor that can take argument is called Parameterized Constructor. Initial values must be passed at the time of object creation. It can be done by two ways.

Constructor-name(para-list)

{ }

- By calling the constructor implicitly
- By calling the constructor explicitly

Copy Constructor

It is used to declare and initialize an object with the values from another object.

Syntax:

copy constructor definition

classname(classname &reference_object)

{

statement;

}

15. DESTRUCTORS

Destructor is a member function used to deallocate the memory space allocated by the constructor. It is used to destroy the object that has been created by a constructor.

General form:

~destructorname()

{

Statement;

}

where ~ (tilde) destructor operator

Characteristics:

- Destructor has the same name as the classname.
- No argument can be provided to a destructor.
- It won‟t return any value.
- They cannot be inherited.
- It may not be static.

16. OPERATOR OVERLOADING

The mechanism of giving special meaning to an operator is known as **Operator overloading.**

All the operators in C++ can be overloaded except the following

1.Class member access operator (., .*)

2.Scope resolution operator(::)

3.Size operator (sizeof)

4.Conditional operator(?:)

Rules for overloading operators

1. Only existing operators can be overloaded.

2. Must have at least one operand that is of user defined data-type.

3. Not able to change basic meaning of an operator

4. Some operators that cannot be overloaded

5. Some operators cannot be overloaded by friend functions. However ,can be overloaded by member functions

Operator function is a function used to perform operator overloading. The operator function could be

1. Member Function

2. Friend Function

The general form of operator function is defined as

Return-type class-name ::operator OP(argument-List)

{

Function Body;

}

The Member Operator function should have

- No argument for Unary Operators
- Only one argument for Binary Operators
- The Friend Operator Function should have
- Only one argument for Unary operators
- Two argument for Binary Operators

The process of overloading involves the following steps

1. Create a class that defines the data type that is to be used in the overloading function

2. Declare the operator overloading function. It may be either **a member function** or **a friend function**

3. Define the operator overloading function

In c++ operators are of two types

A.Unary operator

B.Binary operator

Unary operators using Member function

The Unary Operator – (Negative) when apply to an object, it changes the sign of each of its members. Unary operator acts on only one operand. The unary operators are

- (Negation)

++ (Increment)

-- (Decrement)

For unary operators:

- Member function takes Zero arguments
- Friend function takes one argument

Operators act on two operands are called Binary Operators.

For example, + (Addition), -(Subtraction) , * Multiplication, / Division

For Binary operators:

- Member function takes one argument
- Friend function takes two arguments

Example:

- Binary operator overloading= o op o1; Where op is an operator.

17. INHERITANCE

Inheritance is one of the most important concepts in object-oriented. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

Inheritance is the process of deriving a new class called derived class from an old class called base class. The derived class inherits some or all features of base class.

The idea of inheritance implements the is a relationship. For example, Mango is-a Fruit. Here Fruit is the base class of the derived class Mango. Likewise Mango, Orange, Banana, etc are the derived classes of the base class Fruit.

Features/Advantages of Inheritance

- Reusability
- Extensibility
- Saves time and effort
- Reliability
- Overriding

TYPES OF INHERITANCE

i) Single Inheritance

ii) Multiple Inheritance

iii) Multilevel Inheritance

iv) Hierarchical Inheritance

v) Hybrid Inheritance

vi) Multipath Inheritance

Single Inheritance

A class is derived from only one base class.

Multiple Inheritance

A class is derived from more than one base class.

Multilevel Inheritance

A class is derived from another derived class.

Hierarchical Inheritance

Derivation of several classes from a single base class.

Hybrid Inheritance

Derivation of a class involving more than one form of inheritance is known as Hybrid Inheritance

18. POINTER AND POINTER TO OBJECT

A pointer is a variable that contains a memory address. Very often this address is the location of another object, such as a variable. For example, if x contains the address of y, then x is said to "point to" y. Pointer variables must be declared as such. The general form of a pointer variable declaration is

type *var-name;

Here, type is the pointer's base type. The base type determines what type of data the pointer will be pointing to. var-name is the name of the pointer variable.

To use pointer:

- We define a pointer variable.
- Assign the address of a variable to a pointer.
- Finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand.

19. VIRTUAL FUNCTIONS (POLYMORPHISM)

The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Virtual function

- When same function name used in both base and derived classes, the function in base class is declared as virtual using the keyword virtual at its beginning of declaration.
- Virtual function support late binding (or) dynamic binding. When a function is made virtual, C++ determines which function is use at run time based on the type of object pointed by base pointer.
- By making the base pointer to point to different objects, we can execute different version of virtual function. The derived class inherits the virtual function from the base class.

Rules

- The virtual functions must be members of some class.
- They cannot be static members.

- They are accessed by using object pointers.
- A virtual function can be a friend of another class.
- Virtual function in a base class must be defined even though it may not be used.
- The prototype of base class version of a virtual function and all the derived class
- versions must be identical.
- We cannot have virtual constructors, but we can have virtual destructors.
- Base pointer can point to any type of derived object. But we cannot use a pointer of
- a derived class to access an object of base type.
- Incrementing and decrementing of the base pointer will not point to next object of
- derived class.
- It will get incremented or decremented only relative to its base type.
- If a virtual function is defined in the base class, it not be redefined in the derived
- class, then calls will invoke the base function.

20. ABSTRACT BASE CLASS

Abstract class is a class which contains at least one pure virtual function in it. Abstract classes are used to provide an interface for its sub classes. Classes inheriting an Abstract class must provide definition to the pure virtual function; otherwise they will also become abstract class.

Characteristics

- It cannot be instantiated, but pointers and references of Abstract class type can be
- created.
- It can have normal functions and variables along with a pure virtual function.
- They are mainly used for Up casting, so that its derived classes can use its interface.
- Classes inheriting an Abstract class must implement all pure virtual functions, or else they will become abstract too.

21. C++ STREAMS

The C++ I/O system operates through streams. A stream is logical device that either producesor consumes information. A stream is linked to a physical device by the C++ I/O system. All streams behave in the same manner, even if the actual physical devices they are

linked to differ. Because all streams act the same, the I/O system presents the programmer with a consistent interface.

Two types of streams:

Output stream: a stream that takes data from the program and sends (writes) it to destination.

Input stream: a stream that extracts (reads) data from the source and sends it to the program.

C++ provides both the formatted and unformatted IO functions. In formatted or highlevel IO, bytes are grouped and converted to types such as int, double, string or user-defined types. In unformatted or low-level IO, bytes are treated as raw bytes and unconverted. Formatted IO operations are supported via overloading the stream insertion (<<) and stream extraction (>>)operators, which presents a consistent public IO interface.

C++ provides supports for its I/O system in the header file< iostream>. Just as there are different kinds of I/O (for example, input, output, and file access), there are different classes depending on the type of I/O. The following are the most important stream classes:

- **Class istream** :- Defines input streams that can be used to carry out formatted and unformatted input operations. It contains the overloaded extraction (>>) operator functions. Declares input functions such **get(), getline() and read().**
- **Class ostream** :- Defines output streams that can be used to write data. Declares output functions put and **write().**The ostream class contains the overloaded insertion (<<) operator function

When a C++ program begins, these four streams are automatically opened:

Stream Meaning Default Device

cin Standard input Keyboard

cout Standard output Screen

cerr Standard error Screen

clog Buffer version of cerr Screen

22. UNFORMATTED INPUT/ OUTPUT FUNCTIONS

Functions get() and put()

The get function receives one character at a time. There are two prototypes available in C++ forget as given below:

get (char *)

get ()

Their usage will be clear from the example below:

```
char ch ;
cin.get (ch);
```

In the above, a single character typed on the keyboard will be received and stored in the character variable ch. Let us now implement the get function using the other prototype:

```
char ch ;
ch = cin.get();
```

This is the difference in usage of the two prototypes of get functions.

The complement of get function for output is the put function of the ostream class. It also has two forms as given below:

```
cout.put (var);
```

Here the value of the variable var will be displayed in the console monitor. We can also display a specific character directly as given below:

```
cout.put ('a');
```

getline() and write() functions

C++ supports functions to read and write a line at one go. The getline() function will read one line at atime. The end of the line is recognized by a new line character, which is generated by pressing the Enter key. We can also specify the size of the line.

The prototype of the getline function is given below:

```
cin.getline (var, size);
```

When we invoke the above statement, the system will read a line of characters contained in variable var one at a time. The reading will stop when it encounters a new line character or when the required number (size-1) of characters have been read, whichever occurs earlier. The new line character will be received when we enter a line of size less than specified and press the Enter key. The Enter key or Return key generates a new line character. This character will be read by the function but converted into a NULL character and appended to the line of characters. Similarly, the write function displays a line of given size. The prototype of the write function is given below: write (var, size) ; where var is the name of the string and size is an integer.

Unformatted Binary I/O:

- The binary data format will be known as unformed binary data stream if we store unformatted

binary data on disk it consumer very less disk memory.

- If we want to write records in Binary format we have to open the file in binary mode by using theI/O operations read and write.

Read():

This function is used to read record in a binary form from a file.

Syntax :

Ifstream read(char * buf, streamsize num);

The read() reads num characters from the stream and put them in the buffer pointed to by buf.

Write( ):

This function is used to write the Binary form data on file.

Syntax :

Ofstream write((char*buf, stream size num).

Formatted I/O via manipulators

The C++ I/O system allows you to format I/O operations. For example, you can set a field width,specify a number base, or determine how many digits after the decimal point will be displayed. I/O manipulators are special I/O format functions that can occur within an I/O statement.

23. FILE CONCEPTS AND RELATED FUNCTIONS IN C++

A file is a bunch of bytes stored on some storage devices like hard disk, floppy disk etc. File I/O and console I/O are closely related. In fact, the same class hierarchy that supports console I/O also supports the file I/O. To perform file I/O, you must include <fstream> in your program.

It defines several classes, including **ifstream, ofstream and fstream**. In C++, a file is opened by linking it to a stream. There are three types of streams: input, output and input/output. Before you can open a file, you must first obtain a stream.

To create an input stream, declare an object of type ifstream.

To create an output stream, declare an object of type ofstream.

To create an input/output stream, declare an object of type fstream.

For example, this fragment creates one input stream, one output stream and one stream capable

of both input and output:

ifstream in; // input;

fstream out; // output;

fstream io; // input and output

24. SEQUENTIAL INPUT AND OUTPUT OPERATIONS

The file stream classes support a number of member functions for performing the input and output operations on files.One pairs of functions,**put()** and **get()** are designed for handling a single character at a time.Another pair of functions, **write(),read()** are designed to write and read blocks of binary data.

put() and get() Functions:

The function **put()** writes a single character to the associated stream.Similarly,the function **get()** reads a single character from the associated stream.The following program illustrates how the functions work on a file.The program requests for a string.On receiving the string,the program writes it character by character, to the file using the **put()** function in a for loop.The length of string is used to terminate the for loop.

The program then displays the contents of file on the screen.It uses the function **get()** to fetch a character from the file and continues to do so until the end –of –file condition is reached. The character read from the files is displayed on screen using the operator <<.

write() and read () functions:

The functions **write()** and **read(),**unlike the functions **put()** and **get()** ,handle the data in binary form. This means that the values are stored in the disk file in same format in which they are stored in the internal memory. An int character takes two bytes to store its value in the binary form, irrespective of its size. But a 4 digit int will take four bytes to store it in the character form.The binary input and output functions takes the following form:

infile.read ((char *) & V,sizeof (V));

outfile.write ((char *) & V ,sizeof (V));

These functions take two arguments.The first is the address of the variable V, and the second is the length of that variable in bytes. The address of the variable must be cast to type char*(i.e pointer to character type).The following program illustrates how these two functions are used to save an array of floats numbers and then recover them for display on the screen.

25. RANDOM ACCESS FILE

File pointers

C++ also supports file pointers. A file pointer points to a data element such as character in the file. The pointers are helpful in lower level operations in files.

There are two types of

pointers:

get pointer

put pointer

The get pointer is also called input pointer. When we open a file for reading, we can use the get pointer. The put pointer is also called output pointer. When we open a file for writing, we can use put pointer. These pointers are helpful in navigation through a file. When we open a file for reading, the get pointer will be at location zero and not 1. The bytes in the file are numbered from zero. Therefore, automatically when we assign an object to ifstream and then initialize the object with a file name, the get pointer will be ready to read the contents from 0th position. Similarly, when we want to write we will assign to an ofstream object a filename. Then, the put pointer will point to the 0th position of the given file name after it is created. When we open a file for appending, the put pointer will point to the 0th position. But, when we say write, then the pointer will advance to one position after the last character in the file.

File pointer functions

There are essentially four functions, which help us to navigate the file as given below Functions, Function Purpose

tellg() Returns the current position of the get pointer

seekg() Moves the get pointer to the specified location

tellp() Returns the current position of the put pointer

seekp() Moves the put pointer to the specified location

26. EXCEPTION HANDLING

Two common types of error in a program are:

1) **Syntax error** (arises due to missing semicolon, comma, and wrong prog. constructs etc)

2) **Logical error** (wrong understanding of the problem or wrong procedure to get the solution)

Exceptions

Exceptions are the errors occurred during a program execution. Exceptions are of two types:

- Synchronous (generated by software i.e. division by 0, array bound etc).
- Asynchronous (generated by hardware i.e. out of memory, keyboard etc).

Exception handling mechanism

- C++ exception handling mechanism is basically built upon three keywords namely, try, throw and catch.
- Try block hold a block of statements which may generate an exception.
- When an exception is detected, it is thrown using a throw statement in the try block.

The exception handling is built upon three keywords

a. Try

b. Catch

c. Throw

Try block

Detects and throws exception

Catch block

Catches and handles the exception

A try block can be followed by any number of catch blocks.

The general form of **try** and **catch** block is as follows:

Syntax:-

```
Try {
// try block
}
catch(type1 arg) {
// catch blcok
}
catch(type2 arg) {
// catch block
}
.catch(typen arg) {
// catch block }
```

27. TEMPLATES

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type. A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

There is a single definition of each container, such as **vector**, but we can define many different kinds of vectors for example, **vector <int>** or **vector <string>**. You can use templates to define functions as well as classes, let us see how they work:

Function Template

The general form of a template function definition is shown here:

```
template <class type> ret-type func-name(parameter list)
{
// body of function
}
```

Here, type is a placeholder name for a data type used by the function. This name can be used within the function definition.

Class Template

Just as we can define function templates, we can also define class templates. The general form of a generic class declaration is shown here:

```
template <class type> class class-name
{.
}
```

Here, **type** is the placeholder type name, which will be specified when a class is instantiated. You can define more than one generic data type by using a comma-separated list.

28. COMPONENTS OF SOFTWARE DEVELOPMENT

- Software engineers have been trying various tools, methods, and procedures to control the process of software development in order to build high quality software with improved productivity.
- The methods provide “how to s” for building the software while the tools provide automated or semi-automated support for the methods.
- They are used in all the stages of software development process, namely, planning,

analysis, design, development and maintenance. The software development procedures integrate the methods and tools together and enable rational and timely development of software systems. They provide guidelines as to apply the methods and tools, how to produce the deliverables at each stage, what controls to apply, and what milestones to use to assess the progress.

- There exist a number of software development paradigms, each using a different set of methods and tools.
- The selection of particular paradigms depends on the nature of the application, the programming language used, and the controls and deliverables required.
- The development of a successful system depends not only on the use of the appropriate methods and techniques but also on the developer's commitment to the objectives of the systems.
- A successful system must:

1. satisfy the user requirements,
2. be easy to understand by the users and operators,
3. be easy to operate,
4. have a good user interface,
5. be easy to modify, Tools
6. be expandable,
7. have adequate security controls against misuse of data,
8. handle the errors and exceptions satisfactorily, and
9. Be delivered on schedule within the budget

29. PROCEDURE-ORIENTED PARADIGM.

- Software development is usually characterized by a series of stages depicting the various asks involved in the development process.
- The classic life cycle is based on an underlying model, commonly referred to as the "water fall" model.
- This model attempts to break up the identifiable activities into series of actions, each of which must be completed before the next begins.
- The activities include problem definition, requirement analysis, design, coding, testing and maintenance.

- Further refinements to this model include iteration back to the previous stages in order to incorporate any changes or missing links.

Problem Definition: This activity requires a precise definition of the problem in user terms. A clear statement of the problem is crucial to the success of the software. It helps not only the development but also the user to understand the problem better.

- **Analysis:** This covers a detailed study of the requirements of both the user and the software. The activity is basically concerned with what of the system such as

•What are the inputs to the systems?

•What are the processes required?

•What are the outputs expected?

•What are the constraints?

- **Design:** the design phase deals with various concepts of system design such as data structure, software architecture, and algorithms. This phase translates the requirements into a representation of the software. This stage answers the questions of how.
- **Coding:** coding refers to the translation of the design into machine-readable form. The more detailed the design, the easier is the coding, and better its reliability.
- **Testing:** once the code is written, it should be tested rigorously for correctness of the code and results. Testing may involve the individual units and the whole systems. It requires a detailed plan as to what, when and how to test.
- **Maintenance:** After the software has been installed, it may undergo some changes. This may occur due to a change in the user's requirement, a change in the operating environment, or an error in the software that has been fixed during the testing. Maintenance ensures that these changes are incorporated wherever necessary.

30. OBJECT-ORIENTED ANALYSIS

- Object-oriented analysis (OOA) refers to the methods of specifying requirements of the software in the terms of real-world objects, their behavior, and their interactions.
- Object- oriented design (OOD), on the other hand, turns the software requirements into specifications for objects and derives class hierarchies from which the objects can be created.
- Finally, object-oriented programming (OOP) refers to the implementation of the program using objects, in an object-oriented programming language such as C++.

- By developing specifications of the objects found in the problem space, a clear and well organized statement of the problem is actually built into application.
- These objects form a high-level layer of definitions that are written in terms of the problem space. During the refinement of the definitions and the implementation of the application objects, other objects and identified.
- All the phases in the object-oriented approach work more closely together because of the commonality of the object model. In one phase, the problem domain objects are identified, while in the next phase additional objects required for a particular solution are specified. The design process is repeated for these implementation-level objects.
- Object-oriented analysis provides us with simple, yet powerful, mechanism for identifying objects, the building block of the software to be developed. The analysis is basically concerned with the decomposition of a problem into its component parts and establishing a logical model to describe the system functions.
- The object-oriented analysis (OOA) approach consists of the following steps:

1.Understanding the problem.

2.Drawing the specification of requirement of the user and the software.

3.Identifying the objects and their attributes.

4.Identifying the services that each object is expected to provide (interface).

5.Establishing inter-connections (collaborations) between the objects in terms of services required and services rendered

31. CONTAINER

A container is a way to store data, whether the data consists of built-in types such as int and float, or of class objects.

- The STL makes seven basic kinds of containers available, as well as three more that are rived from the basic kinds.
- Containers in the STL fall into two main categories: ***sequence*** **and** ***associative*****.**
- The sequence containers are ***vector, list,*** **and** ***deque.***
- The associative containers are ***set, multiset, map,*** **and** ***multimap*****.** In addition, several specialized containers are derived from the sequence containers. These are *stack*, *queue*, and *priority queue*

Sequence Containers:

- A sequence container stores a set of elements in what you can visualize as a line, like houses on a street. Each element is related to the other elements by its position along the line. Each element is preceded by one specific element and followed by another.
- The STL provides the *vector* container to avoid these difficulties.This can be very timeconsuming.
- The STL provides the *list* container, which is based on the idea of a linked list.
- The third sequence container is the *deque*, which can be thought of as a combination of a stack and a queue. Both input and output take place on the top of the stack.
- A queue, on the other hand, uses a first-in-first-out arrangement: data goes in at the front and comes out at the back, like a line of customers in a bank.
- A deque combines these approaches so you can insert or delete data from either end. The word deque is derived from Double-Ended QUEue. It's a versatile mechanism that's not only useful in its own right, but can be used as the basis for stacks and queues.

Associative Containers

- An associative container is not sequential; instead it uses *keys* to access data. The keys, typically numbers or stings, are used automatically by the container to arrange the stored elements in a specific order.
- It's like an ordinary English dictionary, in which you access data by looking up words arranged in alphabetical order and the container converts this key to the element's location in memory.
- There are two kinds of associative containers in the STL: ***sets* and *maps*.**
- These both store data in a structure called a ***tree***, which offers fast searching, insertion, and deletion.
- Sets and maps are thus very versatile general data structures suitable for a wide variety of applications. However, it is inefficient to sort them and perform other operations that require random access.
- Sets are simpler and more commonly used than maps. A set stores a number of items which contain *keys*. The keys are the attributes used to order the items.

- For example, a set might store objects of the person class, which are ordered alphabetically using their name attributes as keys. In this situation, you can quickly locate a desired person object by searching for the object with a specified name.
- If a set stores values of a basic type such as int, the key is the entire item stored.
- A map stores pairs of objects: a key object and a value object.

32. STRINGS AND BASIC STRING OPERATIONS IN C++

Putting aside any string-related facilities inherited from C, in C++, strings are not a built-in data type, but rather a Standard Library facility. Thus, whenever we want to use strings or string manipulation tools, we must provide the appropriate **#include** directive, as shown below:

```
#include <string>
using namespace std; // Or using std::string;
```

We now use **string** in a similar way as built-in data types, as shown in the example below, declaring a variable **name**:

```
string name;
```

Unlike built-in data types (**int**, **double**, etc.), when we declare a **string** variable without initialization (as in the example above), we *do have* the guarantee that the variable will be initialized to an empty string — a string containing zero characters.

C++ strings allow you to directly initialize, assign, compare, and reassign with the intuitive operators, as well as printing and reading (e.g., from the user), as shown in the example below:

```
string name;
cout << "Enter your name: " << flush;
cin >> name;
// read string until the next separator
// (space, newline, tab)
// Or, alternatively:
getline (cin, name);
// read a whole line into the string name
if (name == "")
{
cout << "You entered an empty string, "
```

```
<< "assigning default\n";
name = "John";
}
else
{
cout << "Thank you, " << name
<< "for running this simple program!"
<< endl;
}
```

OOPS with C++ 93

String Operations.:

C++ strings also provide many string manipulation facilities. The simplest string manipulation that we commonly use is concatenation, or addition of strings. In C++, we can use the + operator to concatenate (or "add") two strings, as shown below:

```
string result;
string s1 = "hello ";
string s2 = "world";
result = s1 + s2;
// result now contains "hello world"
```

Notice that both **s1** and **s2** remain unchanged! The operation reads the values and produces a result corresponding to the concatenated strings, but doesn't modify any of the two original strings. The **+=** operator can also be used. In that case, one string is appended to another one:

```
string result;
string s1 = "hello";
// without the extra space at the end
string s2 = "world";
result = s1;
result += ' '; // append a space at the end
result += s2;
```

After execution of the above fragment, **result** contains the string **"hello world"**.

You can also use two or more + operators to concatenate several (more than 2) strings. The example below shows how to create a string that contains the full name from first name and last name (e.g., **firstname = "John"**,**lastname = "Smith"**, **fullname = "Smith, John"**).

```
string firstname, lastname, fullname;
```

```
cout << "First name: ";
getline (cin, firstname);
cout << "Last name: ";
getline (cin, lastname);
fullname = lastname + ", " + firstname;
cout << "Fullname: " << fullname << endl;
```

Of course, we didn't need to do that; we could have printed it with several << operators to concatenate to the output. The example intends to illustrate the use of strings concatenation in situations where you need to store the result, as opposed to simply print it.

Now, let's review this example to have the full name in format **"SMITH, John"**. Since we can only convert characters to upper case, and not strings, we have to handle the string one character at a time. To do that, we use the square brackets, as if we were dealing with an array of characters, or a vector of characters.

For example, we could convert the first character of a string to upper case with the following code:

```
str[0] = toupper (str[0]);
```

The function **toupper** is a Standard Library facility related to character processing; this means that when using it, we have to include the **<cctype>** library header:

```
#include <cctype>
```

If we want to change all of them, we would need to know the length of the string. To this end, strings have a method**length**, that tells us the length of the string. we could use that method to control a loop that allows us to convert all the characters to

upper case:

```
for (string::size_type i = 0; i < str.length(); i++)
{
str[i] = toupper (str[i]);
}
```

C++ supports functions that manipulate null-terminated strings.

- strcpy(str1, str2): Copies string str2 into string str1.
- strcat(str1, str2): Concatenates string str2 onto the end of string str1.
- strlen(str1): Returns the length of string str1.
- strcmp(str1, str2): Returns 0 if str1 and str2 are the same; less than 0 if str1<str2;
- greater than 0 if str1>str2.

- strchr(str1, ch): Returns a pointer to the first occurrence of character ch in string str1.
- strstr(str1, str2): Returns a pointer to the first occurrence of string str2 in string str1.

Important functions supported by String Class

- append(): This function appends a part of a string to another string
- assign():This function assigns a partial string
- at(): This function obtains the character stored at a specified location
- begin(): This function returns a reference to the start of the string
- capacity(): This function gives the total element that can be stored
- compare(): This function compares a string against the invoking string
- empty(): This function returns true if the string is empty
- end(): This function returns a reference to the end of the string
- erase(): This function removes character as specified
- find(): This function searches for the occurrence of a specified substring
- length(): It gives the size of a string or the number of elements of a string
- swap(): This function swaps the given string with the invoking one
- *Important Constructors obtained by String Class*
- String(): This constructor is used for creating an empty string