

Question bank C++ answers 2023

Unit 1

Q1. Explain benefits/advantages of OOP.

Ans :

- **Reusability** : Functions and modules that are written by the user can be reused by other users without any modification
- **Inheritance** : we can eliminate redundant code and extend the use of existing classes.
- **Data hiding** : The programmer can hide the data and functions in a class from other classes.
- **Reduce complexity** : The given problem can be viewed as a collection of different objects. Each object is responsible for a specific task. The program is solved by interfacing the object. The technique reduces the complexity of the program.
- **Easy to maintain and upgrade** : OOP makes it easy to maintain and modifying existing code as a new object can be created with small differences to existing ones.

Q2. Explain the characteristics of OOP.

Ans :

- **Classes : write on your own**
- **Objects : write on your own**
- **Data encapsulation** : The class can contain variables for storing data and functions to specify various operations that can perform on data. The process of binding up data and functions into a single unit called as **encapsulation**.
When using data encapsulation data is not accessed directly. It is only accessible through the functions present inside the class.
- **Data abstraction** : Abstraction refers to the act of representing essential features without including the background details or explanations. It represents a functionality of a program without knowing

implementations details It is an approach that speaks about hiding of the complexity and consume only functionality.

- **Inheritance** : creating a new class from existing class or base class is called inheritance. The parent class is also known as parent class or super class. The new class is called as **derived class**. Instead of rewriting the code, you can create the class from the existing class and extending functionality of existing class.
- **Polymorphism** : Polymorphism comes from Greek word Poly and Morphism. Poly means many and morphism means form that is many forms.

Polymorphism means the ability to take more than one form. Advantage of this is you can make an object behave differently in different situations, so that no need to create different objects for different situation.

Q3. Explain the difference between procedures oriented and object oriented programming language.

Ans :

Procedure Oriented Prog.	Object Oriented Prog.
1. Program is divided into small parts called <u>functions</u> .	1. Program is divided into parts called <u>objects</u> .
2. It follows <u>Top Down</u> approach.	2. It follows <u>Bottom Up</u> approach.
3. It does not have any access specifier.	3. OOP has access specifier named Public, Private, Protected
4. Data can move freely from function to function in the system.	4. Objects can move and communicate with each other through member functions.
5. To add new data and function in POP is not so easy.	5. OOP provides an easy way to add new data and function.
6. It is less secure.	6. It is more secure.
7. Ex. C, VB, PASCAL, ALGOL	7. Ex. C++, JAVA, PYTHON

Q4. Explain class and object with example.

Ans : Class : Class is extension of 'c' structure class is a user defined data type which. Consist of both data and a function into a single unit. a class contains variables for storing data and functions to specify various operations. Class is only logical representation of data. Class is blueprint of an object. Class provides the concept of encapsulation. Class provides the concept of data hiding with private declarations.

Example :

```
#include<iostream>
using namespace std;
class temp{
    int a ;
    public:
    int read(){
        a = 100;
        cout << a;
    }
};
```

```
int main(){
    temp obj;
    obj.read();
    return 0;
}
```

Objects : A class is a only representation of data. To work with data, you must have to create a variable for class called an **object**. Object is a variable of type class. Object is a basic unit of object oriented programming. No memory is located when a class is created. Memory is allocated only when a object is created.

Example for class :

```
#include<iostream>
using namespace std;
class student{
    int id;
    public:
    int getdata(){
        cout << "Enter the id " << endl;
        cin >> id;
        return 0;
    }

    int putdata(){
        cout << "The id is : " << id << endl;
    }
};
int main(){
    student obj;
    obj.getdata();
    obj.putdata();
    return 0;
}
```

Q5. Explain access specifiers.

Ans :

Access specifiers are of three types public, private and protected.

Public : public members are accessible from outside the class. The keyword public is used to allow objects to access the member variables of a class directly.

Example :

```
#include<iostream>
using namespace std;
class item{
    public :
        int codeno;
        float price;
        int qty;
};
int main(){
    item one;
    one.codeno = 123;
    one.price = 123.3;
    one.qty = 50;
    cout << "Code no : " << one.codeno << endl;
    cout << "Code price : " << one.price << endl;
    cout << "Code quantity : " << one.qty << endl;
    return 0;
}
```

Private : a private member within a class denotes that only members of the same class have accessibility. The private data member is inaccessible from outside the class. To prevent member variables and functions from direct access. The private keyword is used.

```
#include<iostream>
using namespace std;
class item{
    private :
        int codeno;
        float price;
        int qty;
};
int main(){
    item one;
    one.codeno = 123;
    one.price = 123.3;
    one.qty = 50;
    cout << "Code no : " << one.codeno << endl;
    cout << "Code price : " << one.price << endl;
```

```

    cout << "Code quantity : " << one.qty << endl;
return 0;
}

```

When we run the program in compiler will display error message. Code number, price, quantity are private in the scope.

Protected :

Protected access specified is a stage between private and public access. If member function defined in a class are protected. They cannot be accessed from outside the class, but can be accessed from the derived class.

Q6. Explain default and copy constructor with the help of example.

Ans :

Default constructor :

Constructed without arguments are called **default constructor**.

Example :

```

#include <iostream>
using namespace std;

class A
{
    int a;
public:
    A() // constructor definition
    {
        a = 100;
        cout << "a is = " << a << endl;
    }
};

int main()
{
    A obj;
    return 0;
}

o/p
a is = 100

```

Copy constructor :

The process of copying one object data into another object is called as **copy constructor**. Copy constructor is used to create another copy, or xerox of an object. Copy construction receives another object to initialize current object.

Example :

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

class A
{
    private:
        int a,b;

    public:
        A (int x, int y)
        {
            a=x;
            b=y;
        }

        A (A &obj)
        {
            a=obj.a;
            b=obj.b;
        }

        int show()
        {
            cout<<"a is "<<a<<endl;
            cout<<"b is "<<b<<endl;
            return 0;
        }
};

int main()
{
    A A1(10,20);
    A A2(A1);
    cout<<"Parameterized constructor"<<endl;
    A1.show();
    cout<<"copy constructor"<<endl;
    A2.show();
    return 0;
}
```

Q7. What is inline function?

Ans : Inline is a request to the compiler to make a function as an inline function to reduce the overhead of function calling. If compiler treat a function as an inline function, it substitute the code of function in a single line.

Example:

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

inline product(int a, int b)
{
    return a * b;
}

int main()
{
    int a, b;
    cout << "Enter the value of a and b" << endl;
    cin >> a >> b;
    cout << "The product of a and b is " <<
        product(a, b) << endl;
    return 0;
}
```

Q8. What is friend function?

Ans : Friend function is a function which is not a member of class. Instead of that, it can access private and protected member of class. A friend function is a function which is declared within a class and is defined outside the class. It does not require any scope resolution operator. It can access private members of class. It cannot be called using object of that class.


```

#include <iostream>
using namespace std;

class A
{
    private:
        int a, b;

    public:
        int input()
        {
            cout << "Enter the value of a & b" << endl;
            cin >> a >> b;
        }

        friend int add(A obj); // friend funcn declaration
};

int add(A obj) // friend funcn definition
{
    int c;
    c = obj.a + obj.b;
    cout << "sum is" << c;
}

int main()
{
    A kk;
    kk.input();
    add(kk); // friend funcn calling
    return 0;
}

```

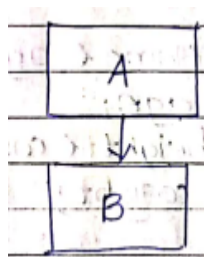
Unit 2

Q1. Explain the types of inheritance.

Ans :

Mechanism of deriving a new class from old class is called as **inheritance**

Single inheritance: if a class is derived from a single base class then is called as single inheritance. In single inheritance there is only one base class and one derived class.



Example:

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

class person
{
protected:
    char name[20];
    int age;
};

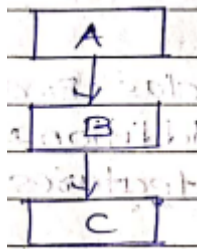
class phydata : public person
{
public:
    float height;
    float weight;
    int getdata()
    {
        cout << "Enter Name < age: ";
        cin >> name >> age;
        cout << "Enter height < weight: ";
        cin >> height >> weight;
        return 0;
    }

    int putdata()
    {
        cout << "In Name is: " << name;
        cout << "In Age is: " << age;
        cout << "In Height is: " << height << "Feet";
        cout << "In Weight is: " << weight << "kg";
        return 0;
    }
};
```

```
int main()
{
    phydata obj;
    obj.getdata();
    obj.putdata();
    return 0;
}
```

Multilevel inheritance :

A class is derived from a class which is derived from another class. It called as multi level inheritance.



Syntax:

```
class A
```

```
{
```

```
}
```

```
class B : public A
```

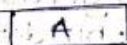
```
{
```

```
}
```

```
class C : public B
```

```
{
```

```
}
```



```

class ONE
{
protected:
    int n1;
};

class TWO: public ONE
{
protected:
    int n2;
};

class THREE: public TWO
{
public:
    int input()
    {
        cout << "Enter two integer No: ";
        cin >> n1 >> n2;
        return 0;
    }

    int sum()
    {
        int sum;
        sum = n1 + n2;
        cout << "sum of is: " << sum;
        return 0;
    }
};

int main()
{
    THREE obj;
    obj.input();
    obj.sum();
}

```

Multiple inheritance :



If a class is derived from More than one class, then it is called multiple inheritance There is only one derive class and several base classes.

Syntax:

```
class base-class1
{
}

class base-class2
{
}

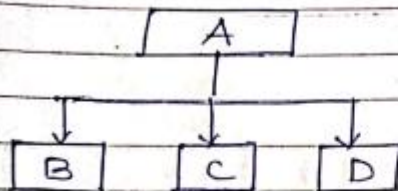
class derived-class:
{
}
```

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

class student
{
protected:
int id;
char name[20];
public:
int getstudent()
{
cout << "Enter student id & name:";
cin >> id >> name;
return 0;
}
int putstudent()
{
cout << "ID = " << id << endl;
cout << "Name = " << name << endl;
}
};
```

Hierarchical Inheritance

If one or more classes are derived from one class then it is called hierarchical inheritance.



class B, class C and class D are derived from class A.

- In this type of inheritance more than one sub class is inherited from a single class.
- more than one derived class is created from single base class.

Syntax

```
class base-class name
```

```
{
```

```
    // body;
```

```
}
```

```
class derived class 1 : visibility mode base-class name
```

```
{
```

```
    // body;
```

```
}
```

```
class derived class 2 : visibility mode base-class name
```

```
{
```

```
}
```

Example:

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

```
class A
```

```
{
```

```
    public:
```

```
    int x, y;
```

```
    int getdata()
```

```
{
```

```
        cout << "Enter the value of x and y:";
```

```
        cin >> x >> y;
```

```
        return 0;
```

```
}
```

```
};
```

```
class B : public A
```

```
{
```

```
    public:
```

```
    int product()
```

```
{
```

```
        cout << "In Product is : " << x * y;
```

```
}
```

```
};
```

```
class C : public A
```

```
{
```

```
    public:
```

```
    int sum()
```

```
{
```



```
    cout << "\n sum is : " << x+y;  
  }  
};
```

```
int main()  
{
```

```
    B obj1;
```

```
    C obj2;
```

```
    obj1.getData();
```

```
    obj1.product();
```

```
    obj2.getData();
```

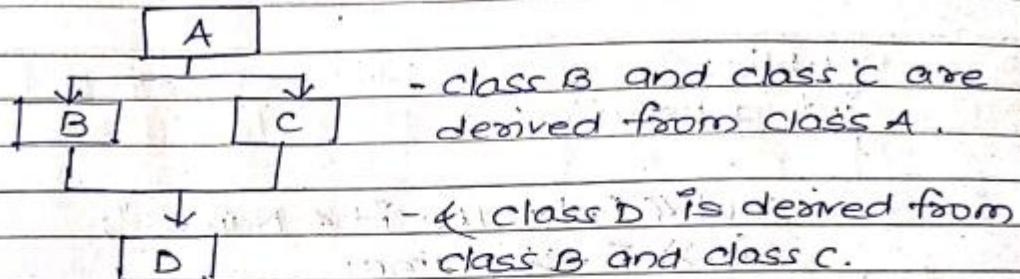
```
    obj2.sum();
```

```
    return 0;
```

```
}
```

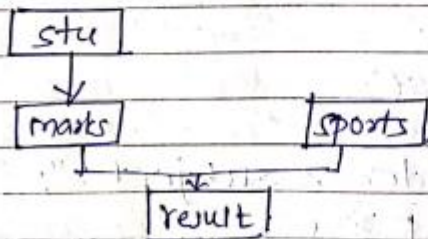
Hybrid Inheritance

The combination of one or more types of inheritance is known as hybrid inheritance.



- class A, class B & class C are example of Hierarchical Inheritance.
- class B, class C & class D is example of multiple Inheritance.

Example.



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

```
class stu
```

```
{
```

```
protected:
```

```
int id;
```

```
char name[20];
```

```
public:
```

```
int getstu()
```

```
{
```

```
cout << "Enter the id & Name:";
```

```
cin >> id >> name;
```

```
return 0;
```

```
}
```

```
};
```

```
class marks: public stu
```

```
{
```

```
protected:
```

```
int m, p, c;
```

```
public:
```

```
int getmarks()
```

```
{
```

```
cout << "Enter 3 subject:";
```

```
cin >> m >> p >> c;
```

```
return 0;
```

```
}
```

```
};
```



```

class sports
{
    protected:
        int spmarks;

    public:
        int getsports()
        {
            cout << "Enter sports marks:";
            cin >> spmarks;
        }
};

class result: public marks, public sports
{
    public:
        int tot;
        float avg;

    public:
        int show()
        {
            tot = m + p + c;
            avg = tot / 3;
            cout << "Total = " << tot << endl;
            cout << "Average = " << avg << endl;
            cout << "Avg + sportsmarks = " << avg + spmarks << endl;
        }
};

```

<pre> int main() { result r; r.getstu(); r.getmarks(); </pre>	<pre> r.getsports(); r.show(); return 0; } </pre>
---	---

Q2. Explain the constructor and destructor order of execution in inheritance.

Ans :

Constructors:

Constructors are special member functions that are automatically called when an object of a class is created. They are responsible for initializing the object's data members and performing any necessary setup.

The constructor order of execution in inheritance follows a top-to-bottom approach in the class hierarchy.

The constructors of the base classes are executed before the constructor of the derived class.

The constructor of the most base class is executed first, and then the constructors of each derived class are executed in the order they appear in the inheritance hierarchy.

Destructors:

Destructors are also special member functions that are automatically called when an object goes out of scope or is explicitly deleted. They are responsible for freeing any resources allocated by the object and performing necessary cleanup.

The destructor order of execution in inheritance follows a bottom-to-top approach in the class hierarchy.

The destructor of the derived class is executed before the destructor of the base class.

The destructor of the most derived class is executed first, and then the destructors of each base class are executed in the reverse order of their construction.

Q3. Explain ambiguity problem in multiple inheritance.

Ans :

In C++, multiple inheritance allows a class to inherit from more than one base class. It can also lead to a problem known as the "**ambiguity problem**" when certain situations arise in the class hierarchy.

The ambiguity problem occurs when there is a naming conflict in the derived class due to multiple inheritance. This means that the derived class inherits the same member (variable or function) from two or more base classes, and the compiler is not able to determine which version of the member to use. This ambiguity can result in compilation errors and makes the program difficult to understand and maintain.

Example :

```
#include <iostream>

using namespace std;

class A {
public:
    void display() {
        cout << "Class A" << endl;
    }
};

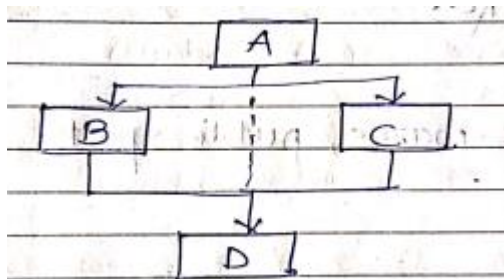
class B {
public:
    void display() {
        cout << "Class B" << endl;
    }
};

class C : public A, public B {
};
```

```
int main() {  
    C obj;  
    obj.display();           // Ambiguity here.  
    return 0;  
}
```

Q4. Explain virtual base class.

Ans : It is used to prevent the duplication. Child class inherits, the grandparent via, two separate paths. It is also called as indirect parent class. All the public and protected member of grandparent are inherited twice into child. We can stop this duplication by making base class virtual.



```

#include <iostream>
using namespace std;
class A
{
protected:
    int a;
};
class B : public virtual A
{
protected:
    int b;
};
class C : public virtual A
{
protected:
    int c;
};
class D : public B, public C
{
    int d;
public:
    public:

```

Scanned with CamScanner

```

int getdata()
{
    cout << "Enter values of a, b & c." << endl;
    cin >> a >> b >> c >> d;
    return 0;
}

int putdata()
{
    cout << "a is = " << a;
    cout << "b is = " << b;
    cout << "c is = " << c;
    cout << "d is = " << d;
}

int main()
{
    D obj;
    obj.getdata();
    obj.putdata();
    return 0;
}

```

Q6. Explain Nested class.

Ans : Nested class is a class that is declared in another class. The outside class is called enclosing class and inside class is called **nested class**. The nested class is also member variable of enclosing class and has some access rights as the other members.

Syntax

```

class class-name {
    // ...
    class class-name-nested {
        // ...
    }
}

```

Scanned with CamScanner

Example

```

#include <iostream>
using namespace std;

class A
{
public:
    class B
    {
private:
        int a, b;

public:
        int getdata()
        {
            cout << "Enter the values of a & b";
            cin >> a >> b;
        }
        int putdata()
        {
            cout << "a = " << a;
            cout << "b = " << b;
        }
    };
};

int main()
{
    cout << "Nested classes in C++" << endl;
    A::B obj;
    obj.getdata();
    obj.putdata();
    return 0;
}

```

Q7. CPP program using class and object with inheritance.

Ans : Solve at paper

Unit 3

Q1. Explain pointers and need of pointers in C++.

Ans : Pointer is a variable that holds address of another variable. The pointer variable and normal variable must be of same data type. Pointer is denoted by * symbol.

Need of pointers :

Pointers save memory space. execution time with pointer is faster because data manipulated with the address that is direct access to memory. Pointer are used with data structure They are useful for representing two dimensional and multi dimensional array. In C++ pointer declared to a place class could access the object of a derived class. However, a pointer to derive class cannot access the object of base class.

Example :

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

int main()
{
    int a = 10;
    int *ptr;
    ptr = &a;

    cout << "Address of a is: " << &a << endl;
    cout << "Value of a is: " << a << endl;
    cout << "Address of pointer is: " << ptr << endl;
    cout << "Value of pointer is: " << *ptr << endl;

    return 0;
}
```

Q2. Explain void pointer.

Ans : void pointer :

The type of pointer which can store the address of all types of data types is called **void pointer**.

Void pointers are pointers that points to a value that has no type. It is declared by the keyword void before the name of pointer. void pointer cannot be directly reference.

The void pointer is useful when you want to work with data of unknown or arbitrary types, such as when dealing with low-level memory operations or when passing pointers to functions that need to accept various data types.

To use a void pointer, you must explicitly cast it to the appropriate data type before dereferencing it. This is done to inform the compiler about the actual data type of the object being pointed to. Casting a void pointer back to its original data type is essential for accessing the correct data when you need to read or modify the pointed-to data.

Ex. void *ptr

Q3. Explain invalid pointer

Ans : An invalid pointer in C++ is a pointer that does not point to a valid memory location. In other words, it is a pointer that contains an address that is not allocated or has been deallocated. Attempting to dereference or use an invalid pointer leads to undefined behavior, which means that the behavior of the program is unpredictable and can result in crashes, data corruption, or other unexpected outcomes.

Q4. Explain null pointer.

Ans : A pointer variable that is initialized with the null value at the time of pointer declaration is called **null pointer**. Null pointer is a type of pointer of any data type, and generally takes the value zero in all pointer doesn't point to any memory location.

```
int main ()
{
    base B;
    base *bptr;
    derived D;
    bptr = &B;
    cout << "Base class object ";
    bptr->b = 100;
    bptr->show();

    bptr = &D;
    bptr->b = 200;
    cout << "Derived class object ";
    bptr->show();

    derived *dptr;
    dptr = &D;
    cout << "\n derived class object ";
    dptr->d = 300;
    dptr->show();
    return 0;
}
```

Q5. Explain the call by value and call by address in C++.

Ans : Call by value : Call by value the actual value that is pass is not changed after performing some operations on it.

Call by address : In call by address, the function receives the memory address of the original argument. This means that any changes made to the parameter inside the function directly affect the original argument outside the function.

To pass by address, you use pointers as function parameters. The pointer points to the memory location of the original argument, allowing you to modify its value inside the function.

Example :

```
#include <iostream>
using namespace std;
void modifyValueByAddress(int* ptr) {
    *ptr = 10;
}
int main() {
    int num = 5;
    modifyValueByAddress(&num);
    cout << "Modified value: " << num << endl;
    return 0;
}
```

Q6. Explain this pointer.

Ans : This pointer (→) :

In C++, the this pointer is a special pointer that is implicitly available within the member functions of a class. It points to the object on which the member function is called. The this pointer allows member functions to access the data members and other member functions of the current object.

The this pointer is especially useful when a member function's parameter name is the same as the name of a data member of the class. In such cases, the this pointer is used to differentiate between the local parameter and the data member.

```
#include <iostream>
using namespace std;
class MyClass {
```

```

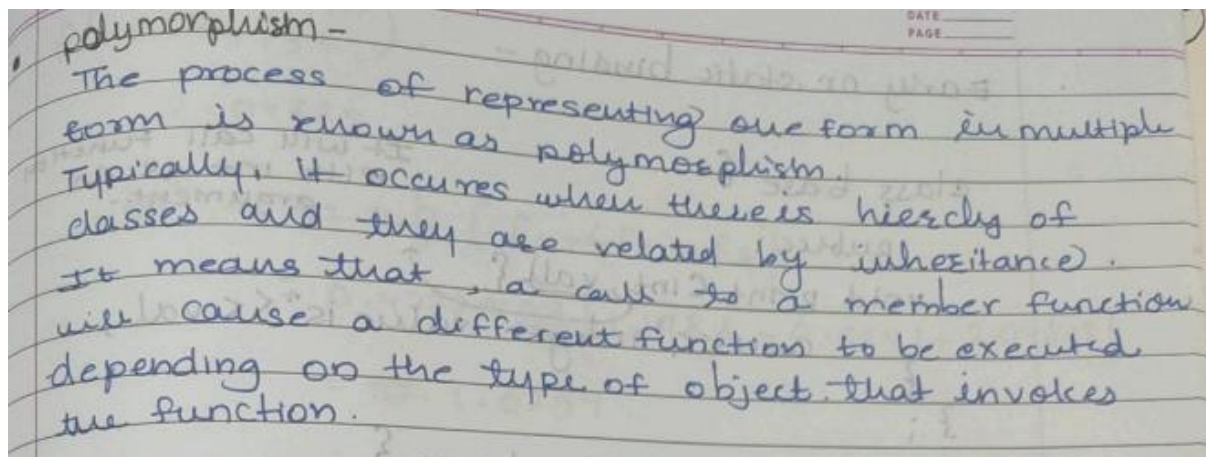
private:
    int x;
public:
    void setValue(int x) {
        this->x = x;
    }
    int getValue() {
        return this->x;
    }
};

int main() {
    MyClass obj;
    obj.setValue(42);
    cout << "Value of x: " << obj.getValue() << endl;
    return 0;
}

```

Q7. Explain Polymorphism.

Ans :



polymorphism -
 The process of representing one form in multiple form is known as polymorphism. Typically, it occurs when there is hierarchy of classes and they are related by inheritance. It 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.

Q8. Differentiate between early binding and late binding.

Ans :

Early binding	Late binding.
<ul style="list-style-type: none">- This is done at compile time.- is said to happen when a function invocation (call) binds to a function definition based on the static type of object.- Early binding is used when all the info is needed to call a function is known at compile time.- Ex - normal funⁿ calls, overloaded funⁿ calls & overloaded operators.	<ul style="list-style-type: none">- This is done at run-time.- When a function invocation binds to the function defⁿ based on the dynamic type of objects.- Decision is made based upon which address is stored in base class's pointer.- <u>Example</u> - function pointers, virtual functions.

base() f()

 / \

 +()

derived()

ptr -> f().

Q9. Differentiate between run time and compile time polymorphism

Compile Time poly ^m	Runtime Time polym ^m
1) The Function to be invoked at the compile time	1) The Function to be invoked at the run-time.
2) It is also known as overloading, early binding & static binding.	2) It is also known as overriding, Dynamic binding and late binding.
3) Overloading is a compile time polymorphism where more than one method is having same name but with different number of parameters or the type of parameters.	3) Overriding is run-time polymorphism where more than one method is having same name, number of parameters & type of parameters.
4) It is achieved by function overloading & operator overloading.	4) It is achieved by virtual functions and pointers.
5) It provides fast execution as it is known at the compile time.	5) It provides slow execution as it is known at the run-time.
6) It is less flexible as mainly all the things execute at the compile time.	6) It is more flexible as mainly all the things execute at the run-time.

Ans :

	Compile-Time Polymorphism	Runtime Polymorphism
1	This is resolved by the compiler .	This is not resolved by the compiler .
2	This is also known as static/early binding or overloading .	This is also known as dynamic/late binding or overriding .
3	The method name is same with different parameters and return type .	The method name is same with the same parameters and same return type .
4	Provides fast execution since the method to be executed is known at compile-time.	Provides slow execution since the method to be executed is known at runtime.
5	Less flexible since all things execute at compile-time.	More flexible since all things execute at runtime.

Q10. Differentiate between function overloading and function overriding.

Ans :

Function Overloading	Function Overriding
<u>Definition:</u> It occurs when two or more methods in one class same method name but different parameters.	It means having two methods with the same method name and parameter but one of the methods is in the parent class and other is in child class.
<u>Polymorphism :</u> Call to an overloaded method is resolved at compile time.	Call to an overridden method is resolved at runtime depending upon the type of object.
<u>Behavior :</u> To add / extend more to method's behavior.	To change existing behavior of method.
<u>Meaning :</u> More than one method shares the same name in class but having different signature.	Method of base class is redefined in derived class having same signature.
<u>Method signature :</u> Must have different signature.	Must have same signature.
<u>Inheritance :</u> Not required.	 always required.
<u>Method relationship:</u> Relationship is b/w methods of same class	 • Relationship is between methods of super class and sub class.
<u>Number of classes</u> Does not require more than one class for overloading.	Require at least two classes for overriding.

Q11. Differentiate between function overloading and operator overloading.

Ans :

Aspect	Function Overloading	Operator Overloading
Purpose	Allows defining multiple functions with the same name but different parameter lists.	Allows defining how operators work with user-defined data types.
Syntax	Functions have the same name but different parameter lists.	Operators are overloaded by defining functions with the <code>operator</code> keyword followed by the operator symbol. For example, <code>operator+</code> , <code>operator==</code> , etc.
Applicability	Applicable to both user-defined and standard library functions.	Applicable only to user-defined classes, enabling custom behavior for operators with objects of those classes.
Argument Types	Functions can have different parameter types, different numbers of parameters, or parameters in a different order.	Operator overloads are specific to the operator and the data types of the operands.
Invocation	Functions are called explicitly using their names.	Operators are invoked implicitly when used with objects of the class or explicitly using the operator syntax.
Return Type	Functions can have different return types.	The return type is defined by the operator overload function, just like a regular function.

Q12. Explain virtual functions and its significance in C++.

Ans : A virtual function is a special form of member function that is declared within a base class and redefined by a derived class.

The keyword `virtual` is used to create a virtual function, precede the function's declaration in the base class.

A virtual function is a function which gets override in the derived class and instructs the C++ compiler for executing late binding on that function.

A function call is resolved at runtime in late binding and so compiler determines the type of object at runtime

Significance :

Virtual functions are a fundamental feature of object-oriented programming in C++. They play a crucial role in achieving polymorphism and enabling dynamic dispatch. The significance of virtual functions lies in their ability to provide runtime polymorphism, which allows a derived class to override the behavior of a function defined in its base class

Polymorphism: Virtual functions enable polymorphism, which means the ability of objects to take on multiple forms. This is achieved through function overriding. When a virtual function is defined in a base class and overridden in a derived class, the function call is resolved at runtime based on the actual type of the object, not the reference or pointer type. This enables dynamic binding and ensures that the correct function is called at runtime based on the type of the object.

Q13. Explain pure virtual functions in C++.

Ans : A pure virtual function means “do nothing” function

We can say empty function. A pure virtual function has no definition relative to the base class

Programmers have to redefine pure virtual function in derived class, because it has no definition in base class

A class containing pure virtual function cannot be used to create any direct objects of its own

This type of class is also called as abstract class

```
#include <iostream>
using namespace std;
class Shape {
public:
    virtual void draw() const = 0;
    void displayInfo() const {
        cout << "This is a shape." << endl;
    }
};
class Circle : public Shape {
public:
    void draw() const override {
        cout << "Drawing a circle." << endl;
    }
};
```

```

class Square : public Shape {
public:
    void draw() const override {
        cout << "Drawing a square." << endl;
    }
};

int main() {
    Circle circle;
    Square square;
    circle.draw();
    square.draw();
    circle.displayInfo();
    square.displayInfo();
    return 0;
}

```

Q14. Explain abstract base class.

Ans : A class that contains at least one pure virtual function is called abstract class

You cannot create objects of an abstract class, you can create pointers and references to an abstract class

An abstract base class in C++ is a class that is designed to be used as a blueprint for other classes but cannot be instantiated directly. It serves as an interface or a contract that defines a set of pure virtual functions that must be implemented by its derived classes. Abstract base classes are also known as abstract classes, and they are used to achieve abstraction, polymorphism, and dynamic binding.

```

#include <iostream>
using namespace std;
class Shape {
public:
    virtual void draw() const = 0;
    void displayInfo() const {
        std::cout << "This is a shape." << std::endl;
    }
    virtual double area() const = 0;
}

```

```

};
class Circle : public Shape {
private:
    double radius;
public:
    Circle(double r) : radius(r) {}

    void draw() const override {
        cout << "Drawing a circle." << endl;
    }

    double area() const override {
        return 3.14159 * radius * radius;
    }
};

class Square : public Shape {
private:
    double side;
public:
    Square(double s) : side(s) {}
    void draw() const override {
        std::cout << "Drawing a square." << std::endl;
    }
    double area() const override {
        return side * side;
    }
};

int main() {
    Circle circle(5.0);
    Square square(4.0);
    circle.draw();
    square.draw();
    circle.displayInfo();
    square.displayInfo();
    cout << "Area of the circle: " << circle.area() << endl;
    cout << "Area of the square: " << square.area() << endl;

    return 0;
}

```

Q15. CPP program to overload the unary operator.

Ans : -----

Q16. CPP program to overload the binary operator.

Ans : -----

Unit 4

Q1. Explain why we need templates in C++?

Ans : The template is one of the most useful characteristics of c++

A template is a technique that allows a single function or class to work with different data types.

Using templates, we can create a single function that can process any type of data. They can accept data from any type, such as int float and long.

Q2. Explain function template.

Ans : The templates declare for functions are called function templates. The function template defines how an individual function can be constructed.

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

template <class T>

void show(T x)
{
    cout << "In x = " << x;
}

int main()
{
    int i = 65;
    char c = 'A';
    float b = 15.97;
    double d = 65.254;

    show(i);
    show(c);
    show(b);
    show(d);
    return 0;
}
```

Q3. Distinguish between function overloading and function templates.

Ans :

Aspect	Function Overloading	Function Templates
Definition	Defining multiple functions with the same name but different parameter lists.	Defining a single generic function that works with multiple data types.
Syntax	Functions have different names or the same name with different parameters.	A single function is defined with template parameters enclosed in angle brackets ('<>').
Applicability	Applicable to functions with different parameter types, numbers, or order.	Applicable to functions that perform similar tasks but with different data types.
Static Polymorphism	Overloading allows static (compile-time) polymorphism based on the function signature.	Templates provide static polymorphism based on the data type used during function invocation.
Code Duplication	Overloaded functions require separate implementations for each data type.	Function templates eliminate the need for code duplication by providing a single generic implementation.
Flexibility	Overloaded functions provide more control over individual implementations for specific data types.	Function templates offer more flexibility by supporting multiple data types with a single definition.
Explicit Function Calls	Overloaded functions are explicitly called using the function name.	Function templates are implicitly called by the compiler based on the data type used.
Example	<code>`cpp int add(int a, int b); double add(double a, double b);`</code>	<code>`cpp template <typename T> T add(T a, T b) { return a + b; }`</code>
Usage	Useful when the function implementations for different data types are distinct.	Useful when the same functionality can be achieved for different data types.
Function Name Clarity	Function names should reflect the different data types or behaviors they support.	Function templates should have generic names that indicate their purpose and flexibility.

Q4. Explain class template.

Ans : Class template :

Class template is a blueprint for creating a family of classes that share the same code structure but can work with different data types. It allows you to create generic classes that can be instantiated with various data types, providing flexibility and code reuse. Class templates are defined using the template keyword followed by one or more template parameters enclosed in angle brackets

Syntax :

```
template <typename T>
```

```
class ClassName {
```

```
    // Class members and functions that use the template parameter T
```

```
};
```

Example :

```
#include <iostream>
using namespace std;
template <typename T>
class MyClass {
public:
    T data;
};

int main() {
    MyClass<int> obj1;
    obj1.data = 42;

    MyClass<double> obj2;
    obj2.data = 3.14;

    cout << "obj1 data: " << obj1.data << endl;
    cout << "obj2 data: " << obj2.data << endl;

    return 0;
}
```

Q5. Explain class template using multiple parameters.

Ans : Class Template using multiple parameters :

A class template can accept multiple template parameters, allowing you to create a generic class that works with multiple data types simultaneously. This allows you to create a family of related classes that share the same code structure but can work with different combinations of data types

Syntax :

```
template <typename T1, typename T2, ...>

class ClassName {

    // Class members and functions that use the template parameters T1, T2,
    etc.

};
```

Example :

```
#include <iostream>
using namespace std;
template <typename T1, typename T2>
class Pair {
public:
    T1 first;
    T2 second;
    Pair(const T1& f, const T2& s) : first(f), second(s) {}
};

int main() {
    Pair<int, double> intDoublePair(42, 3.14);
    cout << "First: " << intDoublePair.first << ", Second: " <<
intDoublePair.second << endl;
    Pair<char, string> charStringPair('A', "Hello");
    cout << "First: " << charStringPair.first << ", Second: " <<
charStringPair.second << endl;

    return 0;
}
```

Q6. Explain generic programming.

Ans : Generic programming in C++ is a programming that aims to create reusable and flexible code by writing algorithms and data structures that work with multiple data types. It allows you to write generic functions and classes that can operate on various data types without duplicating the code for each type. The key concept that enables generic programming in C++ is the use of templates.

There are two main types of templates in C++:

Function Templates: These allow you to define a generic function that can work with different data types. The function template is defined using the template keyword followed by one or more template parameters. Inside the function, these parameters act as types, and you can use them as if they were regular data types.

Class Templates: These allow you to define a generic class that can handle different data types. Similar to function templates, class templates are defined using the template keyword followed by one or more template parameters. The template parameters can be used throughout the class definition.

Q7. Explain Exception handling mechanism.

Ans : What is an exception?

Exceptions are run time anomalies or unusual conditions that a program may encounter while executing

Exception handling Mechanism :

The exception handling mechanism is built upon three keywords

Try

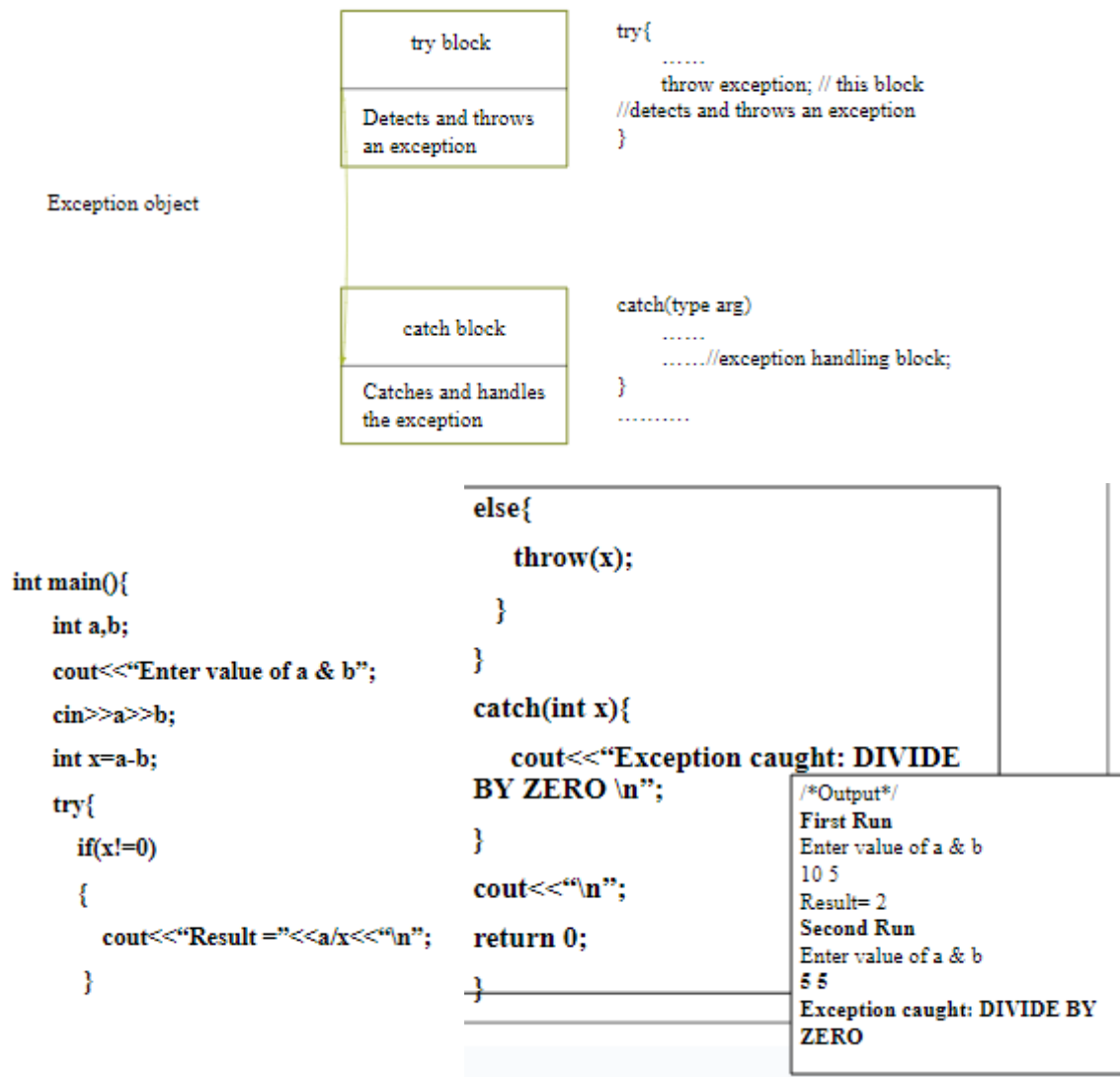
Is used to prefer a block of statements which may generate exceptions

Throw

When an exception is detected, it is thrown using a throw statement in the try block

Catch

A catch block defined by the keyword catch catches the exception by the throw statements in the try block and handles it appropriately



Q8. Explain single catch statements for multiple Exceptions

Ans : You can use a single catch statement to handle multiple exceptions by using a catch block that specifies a common base class for the exceptions you want to catch. This technique is known as catching exceptions by reference or catching exceptions polymorphically.

By using a single catch statement for multiple exceptions, you can consolidate the exception handling logic and make the code more concise and maintainable. The catch block with the most derived exception type will handle the exception first, followed by the catch block for the base exception.

When you catch exceptions by reference, you can handle multiple exception types in the same catch block. The catch block will catch any exception that is of the specified base class or derived from it.

Q9. Explain exception handling with multiple catch statements.

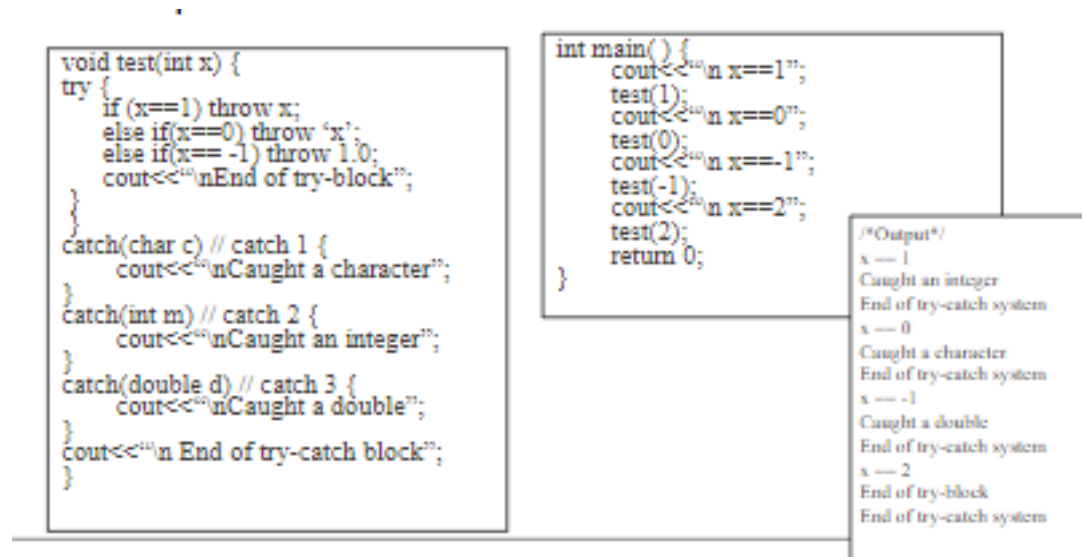
Ans : Multiple catch statements :

Multiple **catch** statements can be associated with a **try** block

When an exception is thrown, the exception handlers are searched for an appropriate match

The first handler that yields the match is executed

After executing the handler, the controls goes to the first statement after the last **catch** block for that **try**



Q10. Explain exception handling to handle "divide by zero".

Ans :

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

int main()
{
    int a, b;
    cout << "Enter the value of a & b" << endl;
    cin >> a >> b;

    try
    {
        if (b != 0)
            cout << a/b;
        else
            throw (b);
    }
    catch (int b)
    {
        cout << "Division by zero" << b;
    }
    return 0;
}
```

This C++ program takes two integer inputs a and b from the user and performs division of a by b. It utilizes a try block to handle the possibility of division by zero. If the divisor b is not zero, the program displays the result of the division. However, if b is zero, it throws an exception of type int representing the division by zero error. The catch block then catches this exception and displays the message "Division by zero." The program ensures that division by zero is handled gracefully, preventing any runtime errors and providing appropriate feedback to the user.

Q11. Explain exception handling to handle "array index out of bound".

Ans :

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

int main(){
    int a[5] = {1,2,3,4,5};
    try
    {
        i = 0;
        while (1){
            if (i != 5)
            {
                cout << a[i] << endl;
                i++;
            }
            else
                throw 1;
        }
    }
    catch(int i)
    {
        cout << "Array index out of bounds exception: " << i << endl;
    }

    return 0;
}
```

Explanation :

This C++ program demonstrates how to use a try block to catch an array index out-of-bounds exception. It declares an integer array `a` with five elements and initiates it with values 1 to 5. Inside the try block, the program initializes the variable `i` to 0 and enters an infinite loop. The loop prints the elements of the array `a` one by one until the index `i` becomes equal to 5. When `i` reaches 5, the program throws an exception with value 1 using `throw`. The catch block catches this exception of type `int` and displays the message "Array index out of bounds exception" along with the value of `i`. The program thus gracefully handles the scenario of accessing an array element beyond its bounds and provides appropriate error handling.

Q12. CPP program using function template.

Ans : -----

Q13. CPP program using class template.

Ans : -----

Q14. CPP program for exceptional handling.

Ans : -----

Unit 5

Q1. Explain data hierarchy with example

Ans : Data hierarchy :

Data hierarchy refers to data elements in a hierarchical structure, where each level represents a different level of abstraction or detail. The data hierarchy is crucial for understanding and managing complex data structures effectively.

1. **Bits:** At the lowest level, data is represented as individual bits (0 or 1). The bit is the smallest unit of data and can represent a binary value.
2. **Bytes:** A byte is a group of 8 bits. It is the smallest addressable unit in memory in most computer architectures.
3. **Arrays:** Arrays are collections of elements of the same data type. They can be one-dimensional, multi-dimensional, or even dynamically allocated.
4. **Fields and Variables:** Fields are a logical grouping of related data elements, and variables are named memory locations that store values of specific data types. Variables can be of primitive data types (e.g., int, float, char) or user-defined data types (e.g., struct, class).

Example :

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

class Person {
```

```

public:
    string name;
    int age;
    string occupation;

    // Constructor to initialize the person's data
    Person(string name, int age, string occupation) {
        this->name = name;
        this->age = age;
        this->occupation = occupation;
    }

    // Function to display the person's information
    void displayInfo() {
        cout << "Name: " << name << "\n";
        cout << "Age: " << age << "\n";
        cout << "Occupation: " << occupation << "\n";
    }
};

int main() {
    // Create an object of the Person class
    Person person1("John Doe", 30, "Engineer");

    // Display the person's information using the member function
    person1.displayInfo();

    return 0;
}

```

Q2. Explain the stream classes.

Ans : Stream classes :

Stream classes are part of the Standard Library and are used for input and output operations. They provide a convenient and unified way to handle communication between a program and various input/output devices, such as the console, files, or other devices.

The key header files that deal with streams in C++ are **<iostream>** (for standard input/output) and **<fstream>** (for file input/output).

Example :

```
#include <iostream>
using namespace std;
int main() {
    int age;
    string name;

    // Input: Read data from the console
    cout << "Enter your name: ";
    cin >> name;

    cout << "Enter your age: ";
    cin >> age;

    // Output: Write data to the console
    cout << "Hello, " << name << "! You are " << age << " years old." << endl;

    return 0;
}
```

Q3. Explain file input output streams.

Ans : File input/output (I/O) streams are used to read data from and write data to files. C++ provides two main stream classes for file I/O, **ifstream** for input and **ofstream** for output.

These classes are part of the **<fstream>** header.

Explanation of file input/output streams in C++:

- **File Input Stream (ifstream):** ifstream is used for reading data from files. It allows you to open a file for reading and read data from it. To use this class, you need to include the **<fstream>** header.

Opening a File for Reading: To open a file for reading, you can use the **open()** member function of **ifstream** and pass the file name as an argument. If the file doesn't exist or cannot be opened, an error occurs.

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

int main() {
```

```

ifstream inputFile;
inputFile.open("example.txt");
if (!inputFile) {
    cout << "Error opening file." << endl;
    return 1;
}
inputFile.close();
return 0;
}

```

File Output Stream (ofstream): ofstream is used for writing data to files. It allows you to open a file for writing and write data to it. To use this class, you need to include the <fstream> header.

Opening a File for Writing: To open a file for writing, you can use the open() member function of ofstream and pass the file name as an argument. If the file doesn't exist, a new file will be created. If the file already exists, its contents will be truncated

Example :

```

#include <iostream>
#include <fstream>
using namespace std;
int main() {
    ofstream outputFile;
    outputFile.open("output.txt");
    if (!outputFile) {
        cout << "Error opening file." << endl;
        return 1;
    }
    int num = 42;
    string text = "Hello, file!";
    outputFile << num << endl;
    outputFile << text << endl;
    outputFile.close();
    return 0;
}

```

Q4. Explain file handling function.

Ans : File Handling : File handling refers to the method of storing data in the C++ program in the form of an output or input that might have been generated

while running a C program in a data file, i.e., a binary file or a text file for future analysis and reference in that very program.

Why files are needed?

- When a program is terminated, the entire data is lost. Storing in a file will preserve your data
- even if the program terminates. ☐ If you have to enter a large number of data, it will take a lot of time to enter them all.
- However, if you have a file containing all the data, you can easily access the contents of the file
- using a few commands in C. ☐ You can easily move your data from one computer to another without any changes.

File Operations In C, you can perform four major operations on files, either text or binary:

1. Creating a new file
2. Opening an existing file
3. Closing a file
4. Reading from and writing information to a file

Q5. Explain file pointer and its type?

Ans : A **file pointer** is a mechanism used to keep track of the current position within a file during input/output operations. It acts as a cursor that points to a specific location within the file. This cursor moves as data is read from or written to the file. The file pointer is an essential concept when working with files because it allows you to control and manipulate the **reading and writing** positions within the file.

C++ provides file pointers for both input and output streams when working with file I/O. For input streams, the file pointer marks the location from which

data is read, and for output streams, the file pointer marks the location where data is written.

1. **Initial Position:** When you open a file for reading or writing, the file pointer is typically set to the beginning of the file (position 0). You can move the file pointer to a specific location within the file using functions like **seekg()** (for input) and **seekp()** (for output).
2. **Moving the File Pointer:** You can move the file pointer to a specific position using the **seekg()** function for input streams and the **seekp()** function for output streams. These functions allow you to set the position relative to the beginning, end, or the current position within the file.
3. **Reading and Writing:** When you read data from a file using input streams, the file pointer advances automatically as data is consumed. Similarly, when you write data to a file using output streams, the file pointer moves ahead automatically after each write operation.
4. **Binary Files:** When working with binary files, you can use the **seekg()** and **seekp()** functions to move the file pointer to specific positions to read or write binary data at precise locations within the file.

```
5. #include <iostream>
6. #include <fstream>
7. using namespace std;
8. int main() {
9.     ofstream outputFile("example.txt");
10.    if (!outputFile) {
11.        cout << "Error opening file." << endl;
12.        return 1;
13.    }
14.    outputFile << "This is a line of text." << endl;
15.    outputFile << "Another line of text." << endl;
16.    outputFile.seekp(0);
17.    outputFile << "New first line." << endl;
18.    outputFile.close();
19.    return 0;
20.}
21.
```

Q6. Explain various file opening modes.

Ans : When working with files, you can specify different file opening modes to control how the file is opened and accessed. These opening modes determine whether the file is opened for reading, writing, appending, or a combination of these operations. The file opening modes are specified as flags when opening a file using **ifstream**, **ofstream**, or **fstream** from the **<fstream>** header.

1. **ios::in (Input Mode):** This mode is used to open a file for reading. It allows you to read data from the file.
2. **ios::out (Output Mode):** This mode is used to open a file for writing. If the file already exists, its content is truncated (cleared), and if the file does not exist, a new file is created.
3. **ios::app (Append Mode):** This mode is used to open a file for writing, but the output is appended to the end of the file, preserving its existing content. If the file does not exist, a new file is created.

```
4. #include <iostream>
5. #include <fstream>
6. using namespace std;
7. int main() {
8.     ofstream outputFile("example.bin", ios::out | ios::binary);
9.     if (!outputFile) {
10.         cout << "Error opening file." << endl;
11.         return 1;
12.     }
13.     outputFile.close();
14.     return 0;
15. }
```

Q7. Explain error handling during file operations

Ans : Error handling during file operations in C++ is crucial to ensure that your program gracefully handles unexpected situations that may occur while working with files. File-related errors can occur due to various reasons, such as incorrect file paths, file permissions, or disk full conditions. Proper error handling helps you detect and respond to these errors appropriately, preventing crashes or undefined behavior in your program.

In C++, the standard file I/O classes (ifstream, ofstream, and fstream) provide error handling mechanisms to check and handle errors that may occur during file operations

Using Exceptions:

C++ also supports exception handling to deal with errors during file operations. By default, file stream classes throw exceptions when an error occurs. You can use a try block to perform file operations and a catch block to handle any exceptions that may be thrown.

Example:

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    ifstream inputFile("example.txt");
    if (!inputFile.good()) {
        cout << "Error opening file." << endl;
        return 1;
    }
    if (inputFile.fail()) {
        cout << "Error reading data from the file." << endl;
        return 1;
    }
    inputFile.close();
    return 0;
}
```

Q8. What is file pointer? Write a note on file opening & file closing.

Ans : A file pointer in C++ is a mechanism used to keep track of the current position within a file during input/output (I/O) operations. It acts as a cursor or marker that points to a specific location within the file. As data is read from or written to the file, the file pointer moves to indicate the current position for the next operation.

When you open a file for reading or writing, the file pointer is typically set to the beginning of the file (position 0). As data is read or written, the file pointer advances accordingly. The file pointer is essential when working with files, as it allows you to control and manipulate the reading and writing positions within the file.

File Opening in C++: To work with files in C++, you use file stream objects (**ifstream**, **ofstream**, or **fstream**) from the **<fstream>** header. To open a file, you call the **open()** member function on the file stream object and provide the file name and the file opening mode as arguments.

File Closing in C++: After performing the necessary file operations, you should close the file using the **close()** member function. Closing the file ensures that all data is correctly flushed (written to the file) and that any file resources are released, preventing any potential data loss.

Q9. CPP program for file handling.

Ans : -----