UNIT V

Polymorphism and Virtual Functions: Compile-time polymorphism, runtime polymorphism, virtual functions.

Templates: Introduction, function templates, class templates.

Exception Handling: Introduction, exception handling mechanism, throwing mechanism, catching mechanism.

5.1 Polymorphism and Virtual Functions

5.1.1 Compile-time Polymorphism

Q1. What is polymorphism? What are its types?

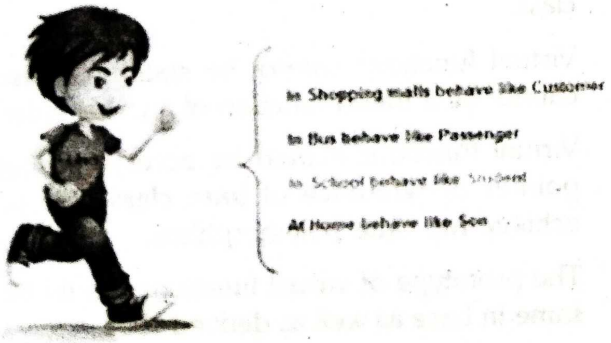
Ans : **(Imp.)**

The process of representing one Form in multiple forms is known as Polymorphism. Here one form represent original form or original method always resides in base class and multiple forms represents overridden method which resides in derived classes.

Polymorphism is derived from 2 greek words: poly and morphs. The word "poly" means many and morphs means forms. So polymorphism means many forms.

Real life example of Polymorphism

Suppose if you are in class room that time you behave like a student, when you are in market at that time you behave like a customer, when you at your home at that time you behave like a son or daughter, Here one person have different-different behaviors.



Type of polymorphism

- Compile time polymorphism
- Run time polymorphism

5.1.2 Runtime Polymorphism

Q2. Explain briefly about runtime polymorphism.

Ans : **(Imp.)**

Run-time polymorphism takes place when functions are invoked during run time. It is also known as dynamic binding or late binding. Function overriding is used to achieve run-time polymorphism.

(a) Function Overriding

When a member function of a base class is redefined in its derived class with the same parameters and return type, it is called function overriding in C++. The base class function is said to be overridden.

The function call is resolved during run time and not by the compiler.

Example to illustrate function overriding in C++

```
#include <iostream>
using namespace std;
class base {
public:
virtual voiddisplay(){
cout<<"Function of base class"<<endl;
}
};
```

```
class derived : public base {
public:
voiddisplay(){
cout<<"Function of derived class"<<endl;
}
};
int main(){
derived d1;
d1.display();
return0;
}
```

Output

Function of derived class

(b) Operator Overloading

We can also overload operators in C++. We can change the behavior of operators for user-defined types like objects and structures.

For example, the '+' operator, used for addition, can also be used to concatenate two strings of std::string class. Its behavior will depend on the operands.

Example of Operator Overloading in C++

```
//Using + operator to add complex numbers
#include <iostream>
using namespace std;
class complex {
private:
float real, imag;
public:
complex(float r=0, float i=0){
real = r;
imag = i;
}
complex operator + (complex const &obj) {
complex result;
result.real = real + obj.real;
result.imag = imag + obj.imag;
return result;
}
void display() {
cout<<real<<"+i"<<imag<<endl;
}
};
int main() {
complex c1(12.4,6), c2(7.9,8);
complex c3 = c1 + c2;
c3.display();
return 0;
}
```

Output

20.3+i14

5.1.3 Virtual Functions.

Q3. What is virtual function? What are the rules for virtual functions?

Ans : **(Imp.)**

A virtual function a member function which is declared within base class and is re-defined (Overriden) by derived class.When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a virtual keyword in base class.
- The resolving of function call is done at Run-time.

Rules for Virtual Functions

1. They Must be declared in public section of class.
2. Virtual functions cannot be static and also cannot be a friend function of another class.
3. Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
4. The prototype of virtual functions should be same in base as well as derived class.

They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.

A class may have virtual destructor but it cannot have a virtual constructor.

- A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
- A 'virtual' is a keyword preceding the normal declaration of a function.
- When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

```
#include <iostream>
using namespace std;
class A
{
  int x=5;
   public:
   void display()
   {
        std::cout << "Value of x is : " <<
        x<<std::endl;
   }
};
class B: public A
{
     int y = 10;
     public:
     void display()
     {
          std::cout << "Value of y is : " <<y<<
          std::endl;
     }
};
int main()
{
      A *a;
      B b;
      a = &b;
     a->display();
      return 0;
}
```

Output:

Value of x is : 5

In the above example, * a is the base class pointer. The pointer can only access the base class members but not the members of the derived class. Although C++ permits the base pointer to point to any object derived from the base class, it cannot directly access the members of the derived class. Therefore, there is a need for virtual function which allows the base pointer to access the members of the derived class.

virtual function Example

Let's see the simple example of C++ virtual function used to invoked the derived class in a program.

```
#include <iostream>
{
 public:
 virtual void display()
 {
  cout << "Base class is invoked"<<endl;
 }
};
```

```
class B:public A
{
 public:
 void display()
 {
  cout << "Derived Class is invoked"<<endl;
 }
};
int main()
{
 A* a;          //pointer of base class
 B b;           //object of derived class
 a = &b;
 a->display();  //Late Binding occurs
}
```

Output:

Derived Class is invoked

Q4. What is pure virtual function? Explain.

Ans :

Pure Virtual Function

- A virtual function is not used for performing any task. It only serves as a placeholder.
- When the function has no definition, such function is known as "do-nothing" function.
- The "do-nothing" function is known as a pure virtual function. A pure virtual function is a function declared in the base class that has no definition relative to the base class.
- A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes.
- The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

Pure virtual function can be defined as:

```
virtual void display() = 0;
```

Let's see a simple example:

```
#include <iostream>
using namespace std;
class Base
{
    public:
    virtual void show() = 0;
};
class Derived : public Base
{
    public:
    void show()
    {
     std::cout << "Derived class is derived from
     the base class." << std::endl;
    }
};
int main()
{
    Base *bptr;
    //Base b;
    Derived d;
    bptr = &d;
    bptr->show();
    return 0;
}
```

Output:

Derived class is derived from the base class.

In the above example, the base class contains the pure virtual function. Therefore, the base class is an abstract base class. We cannot create the object of the base class.

5.2 TEMPLATES

5.2.1 Introduction

Q5. What is template? What are its types?

Ans :

(Imp.)

Templates in C++ programming allows function or class to work on more than one data type at once without writing different codes for

different data types. Templates are often used in larger programs for the purpose of code reusability and flexibility of program. The concept of templetes can be used in two different ways:

- Function Templates
- Class Templates

Use of templates

- Create a typesafe collection class (for example, a stack) that can operate on data of any type.
- Add extra type checking for functions that would otherwise take void pointers.
- Encapsulate groups of operator overrides to modify type behavior (such as smart pointers).

Most of these uses can be implemented without templates; however, templates offer several advantages:

- Templates are easier to write. You create only one generic version of your class or function instead of manually creating specializations.
- Templates can be easier to understand, since they can provide a straightforward way of abstracting type information.
- Templates are typesafe. Because the types that templates act upon are known at compile time, the compiler can perform type checking before errors occur.

5.2.2 Function Templates

Q6. What is function template? how to define it? Explain with an example. **(Imp.)**

Ans :

Function Templates

A function template works in a similar to a normal function, with one key difference.

A single function template can work with different data types at once but, a single normal function can only work with one set of data types.

Normally, if you need to perform identical operations on two or more types of data, you use function overloading to create two functions with the required function declaration.

However, a better approach would be to use function templates because you can perform the same task writing less and maintainable code.

How to declare a function template?

A function template starts with the keyword template followed by template parameter/s inside <> which is followed by function declaration.

```
template<class T>
    T someFunction(T arg)
    {
    ... .. ...
    }
```

In the above code, T is a template argument that accepts different data types (int, float), and class is a keyword.

You can also use keyword typename instead of class in the above example.

When, an argument of a data type is passed to someFunction( ), compiler generates a new version of someFunction() for the given data type.

C++ program to add two numbers using function template.

```
#include <iostream>
#include <conio.h>
using namespace std;
template<class t1,class t2>
void sum(t1 a,t2 b) // defining template function
{
    cout<<"Sum="<<a+b<<endl;
}
int main()
{
    int a,b;
    float x,y;
    cout<<"Enter two integer data: ";
    cin>>a>>b;
    cout<<"Enter two float data: ";
```

```
    cin>>x>>y;
    sum(a,b); // adding two integer type data
    sum(x,y); // adding two float type data
    sum(a,x); // adding a float and integer type data
    getch();
    return 0;
}
```

This program illustrates the use of template function in C++. A template function sum() is created which accepts two arguments and add them. The type of argument is not defined until the function is called. This single function is used to add two data of integer type, float type and, integer and float type. We don't need to write separate functions for different data types. In this way, a single function can be used to process data of various type using function template.

Output

```
Enter two integer data: 6 10
Enter two float data: 5.8 3.3
Sum=16
Sum=9.1
Sum=11.8
```

Q7. Write a program to display largest among two numbers using function templates.

Ans : **(Imp.)**

// If two characters are passed to function template, character with larger ASCII value is displayed.

```
#include<iostream>
usingnamespace std;
// template function
template<class T>
T Large(T n1, T n2)
{
return(n1 > n2)? n1 : n2;
}
int main()
{
int i1, i2;
float f1, f2;
char c1, c2;
cout <<"Enter two integers:\n";
cin >> i1 >> i2;
cout <<Large(i1, i2)<<" is larger."<< endl;
cout <<"\nEnter two floating-point numbers:\n";
cin >> f1 >> f2;
cout <<Large(f1, f2)<<" is larger."<< endl;
cout <<"\nEnter two characters:\n";
cin >> c1 >> c2;
cout <<Large(c1, c2)<<" has larger ASCII value.";
return0;
}
```

Output

```
Enter two integers:
5
10
10 is larger.
Enter two floating-point numbers:
12.4
10.2
12.4 is larger.
Enter two characters:
z
Z
z has larger ASCII value.
```

In the above program, a function template Large() is defined that accepts two arguments n1 and n2 of data type T. T signifies that argument can be of any data type.

Q8. Write a Program to swap data using function templates.

Ans : **(Imp.)**

```
#include <iostream>
using namespace std;
template <typename T>
void Swap(T &n1, T &n2)
{
T temp;
temp = n1;
```

```
n1 = n2;
n2 = temp;
}
int main()
{
int i1 = 1, i2 = 2;
float f1 = 1.1, f2 = 2.2;
char c1 = 'a', c2 = 'b';

cout << "Before passing data to function template.\n";
cout << "i1 = " << i1 << "\ni2 = " << i2;
cout << "\nf1 = " << f1 << "\nf2 = " << f2;
cout << "\nc1 = " << c1 << "\nc2 = " << c2;
Swap(i1, i2);
Swap(f1, f2);
Swap(c1, c2);

cout << "\n\nAfter passing data to function template.\n";
cout << "i1 = " << i1 << "\ni2 = " << i2;
cout << "\nf1 = " << f1 << "\nf2 = " << f2;
cout << "\nc1 = " << c1 << "\nc2 = " << c2;
return 0;
}
```

Output

```
Before passing data to function template.
i1 = 1
i2 = 2
f1 = 1.1
f2 = 2.2
c1 = a
c2 = b
After passing data to function template.
i1 = 2
i2 = 1
f1 = 2.2
f2 = 1.1
c1 = b
c2 = a
```

In this program, instead of calling a function by passing a value, a call by reference is issued.

The Swap() function template takes two arguments and swaps them by reference.

Q9. Explain about how Multiple Types parameters are used with Function Templates.

Ans : **(Imp.)**

With templates, you may have more than one template-type parameters. It goes like:

template<class T1, class T2, ... >

Where T1 and T2 are type-names to the function template. You may use any other specific name, rather than T1, T2. Note that the usage of '...' above does not mean that this template

Let's have a simple example taking two template parameters:

```
Template<class T1, class T2>
void PrintNumbers(const T1& t1Data, const T2& t2Data)
{
cout <<"First value:"<< t1Data;
cout <<"Second value:"<< t2Data;
}
And we can simply call it as:
printNumbers(10, 100);    // int, int
PrintNumbers(14, 14.5);   // int, double
PrintNumbers(59.66, 150); // double, int
```

Another way of using the maxtemplate with arguments of different types is changing its definition in the following way:

```
template<typenameT1, typenameT2>
T1 max(constT1& a, constT2& b)
{
return (a > b) ? a : b;
}
void f() {
cout<< max(4, 5.5);// T1 isint, T2 isdouble
cout<< max(5.5, 4);// T1 isdouble, T2 isint
}
```

In similar fashion, function templates may have 3 or more type parameters, and each of them would map to the argument types specified in function call. As an example, the following function template is legal:

```
template<class T1, class T2, class T3>
T2 DoSomething(const T1 tArray[], T2 tDefaultValue, T3& tResult)
{
...
}
```

Q10. Write a program to implement function template with multiple arguments.

Ans :

The following is an example of a template supporting multiple types:

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

    template <typename T, typename U>

    void squareAndPrint(T x, U y)
    {

    T result;
    U otherVar;

    cout << "X: " << x << " " << x * x
    << endl;
    cout << "Y: " << y << " " << y * y
    << endl;
    };

main()
{

    int ii = 2;
    float jj = 2.1;

    squareAndPrint<int,float>(ii, jj);

}
```

Output

X: 2 4

Y: 2.1 4.41

A single type can only be specified once.

Q11. Explain about how to Overload of Template Functions.

Ans :

A template function also supports the overloading mechanism. It can be overloaded by a normal function or a template function. While invoking these functions, an error occurs if no accurate match is met. No implicit conversion is carried out in the parameters of template functions. The compiler observes the following rules for choosing an appropriate function when the program contains overloaded functions:

Searches for an accurate match of functions; if found, it is invoked searches for a template function through which a function that can be invoked with an accurate match can be generated; if found, it is invoked.

Example of Overloading Template Function

C++ program to overload template function for sum of numbers.

```
#include <iostream>
#include <conio.h>
using namespace std;
template<class t1>
void sum(t1 a,t1 b,t1 c)
{
    cout<<"Template function 1: Sum = "<<a+b+c<<endl;
}
template <class t1,class t2>
void sum(t1 a,t1 b,t2 c)
{
    cout<<"Template function 2: Sum = "<<a+b+c<<endl;
}
```

```
void sum(int a,int b)
{
    cout<<"Normal function: Sum = "<<a+b<<endl;
}
int main()
{
  int a,b;
  float x,y,z;
  cout<<"Enter two integer data: ";
  cin>>a>>b;
  cout<<"Enter three float data: ";
  cin>>x>>y>>z;
  sum(x,y,z); // calls first template function
  sum(a,b,z); // calls first template function
  sum(a,b); // calls normal function
  getch();
  return 0;
}
```

In this program, template function is overloaded by using normal function and template function. Three functions named sum() are created. The first function accepts three arguments of same type. The second function accepts three argument, two of same type and one of different and, the third function accepts two arguments of int type. First and second function are template functions while third is normal function. Function call is made from main() function and various arguments are sent. The compiler matches the argument in call statement with arguments in function definition and calls a function when match is found.

Output

```
Enter two integer data: 5 9
Enter three float data: 2.3 5.6 9.5
Template function 1: Sum = 17.4
Template function 2: Sum = 23.5
Normal function: Sum = 14
```

5.2.3 Class Templates

Q12. What is Class Template ? How to define it.

Ans : **(Imp.)**

Like function templates, you can also create class templates for generic class operations.

Sometimes, you need a class implementation that is same for all classes, only the data types used are different.

Normally, you would need to create a different class for each data type OR create different member variables and functions within a single class.

This will unnecessarily bloat your code base and will be hard to maintain, as a change is one class/function should be performed on all classes/ functions.

However, class templates make it easy to reuse the same code for all data types.

How to declare a class template

```
template<class T>
class className
{
... .. ...
public:
T var;
T someOperation(T arg);
... .. ...
};
```

In the above declaration, `T` is the template argument which is a placeholder for the data type used.

Inside the class body, a member variable var and a member function someOperation() are both of type T.

Q13. How to create a class template object?

Ans :

To create a class template object, you need to define the data type inside a <> when creation.

```
className<dataType> classObject;
```

For example:

```
className<int> classObject;
className<float> classObject;
className<string> classObject;
```

Q14. Write a program to display Simple calculator using Class template.

Ans :

Program to add, subtract, multiply and divide two numbers using class template

```
#include<iostream>
usingnamespace std;
template<class T>
classCalculator
{
private:
T num1, num2;
public:
Calculator(T n1, T n2)
{
num1 = n1;
num2 = n2;
}

void displayResult()
{
cout <<"Numbers are: "<< num1 <<" and "<<
num2 <<"."<< endl;
cout <<"Addition is: "<< add()<< endl;
cout <<"Subtraction is: "<< subtract()<< endl;
cout <<"Product is: "<< multiply()<< endl;
cout <<"Division is: "<< divide()<< endl;
}

T add(){return num1 + num2;}
T subtract(){return num1 - num2;}
T multiply(){return num1 * num2;}
T divide(){return num1 / num2;}
};

int main()
{
Calculator<int> intCalc(2,1);
Calculator<float> floatCalc(2.4,1.2);
cout <<"Int results:"<< endl;
intCalc.displayResult();
cout << endl <<"Float results:"<< endl;
floatCalc.displayResult();
return0;
}
```

Output

```
Int results:
Numbers are: 2 and 1.
Addition is: 3
Subtraction is: 1
Product is: 2
Division is: 2
Float results:
Numbers are: 2.4 and 1.2.
Addition is: 3.6
Subtraction is: 1.2
Product is: 2.88
Division is: 2
```

In the above program, a class template Calculator is declared.

The class contains two private members of type T: num1 & num2, and a constructor to initalize the members.

Q15. Write a C++ program to use class template.

Ans :

```
#include <iostream>
#include <conio.h>
using namespace std;
template<class t1,class t2>
class sample
{
  t1 a;
  t2 b;
  public:
    void getdata()
    {
```

```
        cout<<"Enter a and b: ";
        cin>>a>>b;
    }
    void display()
    {
        cout<<"Displaying values"<<endl;
        cout<<"a="<<a<<endl;
        cout<<"b="<<b<<endl;
    }
};

int main()
{
    sample<int,int> s1;
    sample<int,char> s2;
    sample<int,float> s3;
    cout <<"Two Integer data"<<endl;
    s1.getdata();
    s1.display();
    cout <<"Integer and Character data"<<endl;
    s2.getdata();
    s2.display();
    cout <<"Integer and Float data"<<endl;
    s3.getdata();
    s3.display();
    getch();
    return 0;
}
```

In this program, a template class sample is created. It has two data a and b of generic types and two methods: getdata() to give input and display() to display data. Three object s1, s2 and s3 of this class is created. s1 operates on both integer data, s2 operates on one integer and another character data and s3 operates on one integer and another float data. Since, sample is a template class, it supports various data types.

Output

```
Two Integer data
Enter a and b: 7 11
Displaying values
a=7
b=11
Integer and Character data
Enter a and b: 4 v
Displaying values
a=4
b=v
Integer and Float data
Enter a and b: 14 19.67
Displaying values
a=14
b=19.67
```

Q16. Explain, how to use class templates with multiple parameters.

Ans :

While creating templates, it is possible to specify more than one type. We can use more than one generic data type in a class template. They are declared as a comma-separated list within the template as below:

Syntax:

```
template<class T1, class T2, ...>
class classname
{
    ...
    ...
};
```

Example

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

// Class template with two parameters
template<class T1, class T2>
class Test
{
    T1 a;
    T2 b;
  public:
    Test(T1 x, T2 y)
```

```
    {
        a = x;
        b = y;
    }
    void show()
    {
        cout << a << " and " << b << endl;
    }
};

// Main Function
int main()
{
    // instantiation with float and int type
    Test <float, int> test1 (1.23, 123);
    // instantiation with float and char type
    Test <int, char> test2 (100, 'W');
    test1.show();
    test2.show();
    return 0;
}
```

5.3 Exception Handling

5.3.1 Introduction

Q17. What is exception? Why to use exceptions.

Ans : **(Imp.)**

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

Following are main advantages of exception handling over traditional error handling.

Separation of Error Handling code from Normal Code: In traditional error handling codes, there are always if else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try catch blocks, the code for error handling becomes separate from the normal flow.

Functions/Methods can handle any exceptions they choose: A function can throw many exceptions, but may choose to handle some of them. The other exceptions which are thrown, but not caught can be handled by caller. If the caller chooses not to catch them, then the exceptions are handled by caller of the caller.

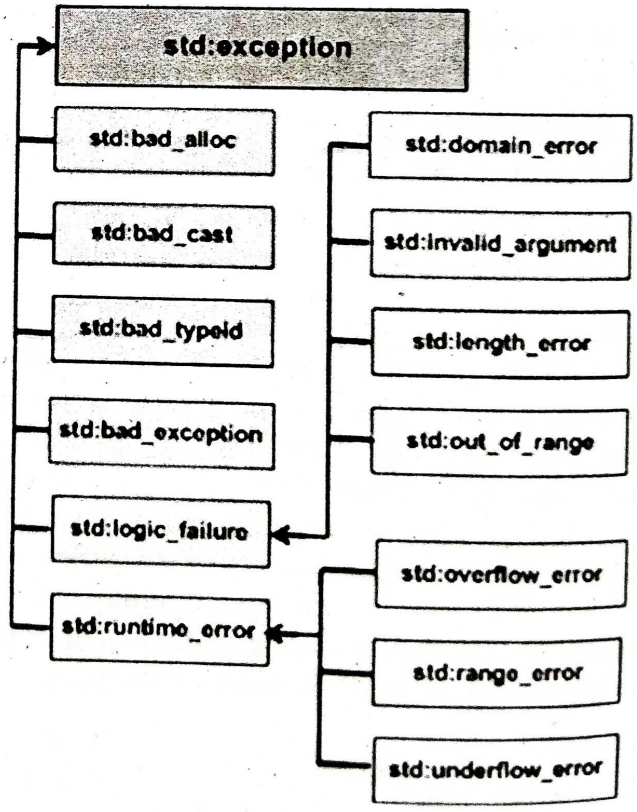
In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way.

Grouping of Error Types: In C++, both basic types and objects can be thrown as exception. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, categorize them according to types.

Q18. What are the standard exceptions in C++. List them in order.

Ans :

C++ provides a list of standard exceptions defined in <exception> which we can use in our programs. These are arranged in a parent-child class hierarchy shown below:



Here is the small description of each exception mentioned in the above hierarchy:

Exception	Description
std::exception	An exception and parent class of all the standard C++ exceptions.
std::bad_alloc	This can be thrown by new.
std::bad_cast	This can be thrown by dynamic_cast.
std::bad_exception	This is useful device to handle unexpected exceptions in a C++ program
std::bad_typeid	This can be thrown by typeid.
std::logic_error	An exception that theoretically can be detected by reading the code.
std::domain_error	This is an exception thrown when a mathematically invalid domain is used
std::invalid_argument	This is thrown due to invalid arguments.
std::length_error	This is thrown when a too big std::string is created
std::out_of_range	This can be thrown by the at method from for example a std::vector and std::bitset<>::operator[]().
std::runtime_error	An exception that theoretically can not be detected by reading the code.
std::overflow_error	This is thrown if a mathematical overflow occurs.
std::range_error	This is occured when you try to store a value which is out of range.
std::underflow_error	This is thrown if a mathematical underflow occurs.

5.3.2 Exception Handling Mechanism

Q19. Explain about how to handle exceptions in C++.

(OR)

Explain about different keywords to handle exceptions.

(Imp.)

Ans :

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

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

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

try: A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
// protected code
}catch( ExceptionName e1 )
{
// catch block
}catch( ExceptionName e2 )
{
// catch block
}catch( ExceptionName eN )
{
// catch block
}
```

You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

5.3.3 Throwing Mechanism

Q20. Explain the throwing mechanism in exception handling.

Ans : **(Imp.)**

Exception handling in C++ revolves around these three keywords:

- **throw –** when a program encounters a problem, it throws an exception. The throw keyword helps the program perform the throw.
- **catch –** a program uses an exception handler to catch an exception. It is added to the section of a program where you need to handle the problem. It's done using the catch keyword.
- **try –** the try block identifies the code block for which certain exceptions will be activated. It should be followed by one/more catch blocks.
- n exception in C++ is thrown by using the throw keyword from inside the try block. The throw keyword allows the programmer to define custom exceptions.
- Exception handlers in C++ are declared with the catch keyword, which is placed immediately after the try block. Multiple handlers (catch expressions) can be chained - each one with a different exception type. Only the handler whose argument type matches the exception type in the throw statement is executed.
- C++ does not require a finally block to make sure resources are released if an exception occurs.

Throwing Exceptions

Exceptions can be thrown anywhere within a code block using throw statements. The operand of the throw statements determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs:

```
double division(int a,int b){
if( b ==0){
throw"Division by zero condition!";
}
return(a/b);
}
```

C++ Throw Exception Example

The following example shows the syntax for throwing and catching exceptions in C++:

```
#include <iostream>
#include <stdexcept>

using namespace std;

int AddPositiveIntegers(int a, int b)
{
  if (a < 0 || b < 0)
    throw std::invalid_argument("AddPositiveIntegers arguments must be positive");

  return (a + b);
}

int main()
{
  try
  {
    cout << AddPositiveIntegers(-1, 2); //exception
  }

  catch (std::invalid_argument& e)
  {
    cerr << e.what() << endl;
    return -1;
  }

  return 0;
}
```

In this C++ throw exception example, the AddPositiveIntegers() function is called from inside the try block in the main() function. The AddPositiveIntegers() expects two integers a and b as arguments, and throws an invalid_argument exception in case any of them are negative.

The std::invalid_argument class is defined in the standard library in the <stdexcept> header file. This class defines types of objects to be thrown as exceptions and reports errors in C++ that occur because of illegal argument values.

The catch block in the main() function catches the invalid_argument exception and handles it.

Throwing exceptions from C++ constructors

An exception should be thrown from a C++ constructor whenever an object cannot be properly constructed or initialized. Since there is no way to recover from failed object construction, an exception should be thrown in such cases.

Constructors should also throw C++ exceptions to signal any input parameters received outside of allowed values or range of values.

Since C++ constructors do not have a return type, it is not possible to use return codes. Therefore, the best practice is for constructors to throw an exception to signal failure.

The throw statement can be used to throw an C++ exception and exit the constructor code.

5.3.4 Catching Mechanism

Q21. Explain about the catching mechanism in exception handling.

Ans : **(Imp.)**

The catch block following the try block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try{
// protected code
}catch(ExceptionName e ){
// code to handle ExceptionName exception
}
```

Above code will catch an exception of ExceptionName type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows:

```
try{
// protected code
}catch(...){
// code to handle any exception
}
```

Q22. Write a program to perform exception handling for Divide by zero Exception.

Ans :

```
#include<iostream.h>
#include<conio.h>
void main()
{
int a,b,c;
float  d;
clrscr();
cout<<"Enter the value of a:";
cin>>a;
cout<<"Enter the value of b:";
cin>>b;
cout<<"Enter the value of c:";
cin>>c;
try
{
if((a-b)!=0)
{
d=c/(a-b);
cout<<"Result is:"<<d;
}
else
{
throw(a-b);
}
}
catch(int i)
{
cout<<"Answer is infinite because a-b is:"<<i;
}
getch();
}
```

Output:

```
Enter the value for a: 20
Enter the value for b: 20
Enter the value for c: 40
Answer is infinite because a-b is: 0
```

Q23. Why do multiple catch statements are used in exceptions? Write the syntax to declare multiple catch statements.

(Imp.)

Ans :

Multiple catch exception statements are used when a user wants to handle different exceptions differently.

For this, a user must include catch statements with different declaration.

Multiple Catch Exception-Syntax:

```
try
{
body of try block
}
catch (type1 argument1)
{
statements;
...
... .
..
}
catch (type2 argument2)
{
statements;
...
... .
..
}
...
... .
    ...
catch (typeN argumentN)
{
statements;
... .
.. ..
. }
```

Q24. Write the syntax of multiple exceptions

Ans :

```
try
{
    // try section
}
catch  (object1)
{
    // catch section1
}
catch  (object2)
{
    // catch section2
}
. . . . . .
. . . . . .
catch  (type n object)
{
    // catch section-n
}
```

Q25. Write a program to perform exception handling with multiple catch.

Ans :

```
#include<iostream.h>
#include<conio.h>
void test(int x)
{
try
{
if(x>0)
throw x;
else
throw 'x';
}

catch(int x)
{
cout<<"Catch a integer and that integer is:"<<x;
}
catch(char x)
{
cout<<"Catch a character and that character
is:"<<x;
}
}

void main()
{
clrscr();
cout<<"Testing multiple catches\n:";
test(10);
test(0);
getch();
}
```

Output:

```
Testing multiple catches
Catch a integer and that integer is: 10
Catch a character and that character is: x
```

Short Question and Answers

1. What is polymorphism?

Ans :

The process of representing one Form in multiple forms is known as Polymorphism. Here one form represent original form or original method always resides in base class and multiple forms represents overridden method which resides in derived classes.

Polymorphism is derived from 2 greek words: poly and morphs. The word "poly" means many and morphs means forms. So polymorphism means many forms.

2. What is virtual function?

Ans :

A virtual function a member function which is declared within base class and is re-defined (Overriden) by derived class.When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a virtual keyword in base class.
- The resolving of function call is done at Run-time.

3. Rules for Virtual Functions

Ans :

i) They Must be declared in public section of class.

ii) Virtual functions cannot be static and also cannot be a friend function of another class.

iii) Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.

iv) The prototype of virtual functions should be same in base as well as derived class.

4. What is pure virtual function?

Ans :

- A virtual function is not used for performing any task. It only serves as a placeholder.
- When the function has no definition, such function is known as "do-nothing" function.
- The "do-nothing" function is known as a pure virtual function. A pure virtual function is a function declared in the base class that has no definition relative to the base class.
- A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes.
- The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

5. What is template?

Ans :

Templates in C++ programming allows function or class to work on more than one data type at once without writing different codes for different data types. Templates are often used in larger programs for the purpose of code reusability and flexibility of program. The concept of templetes can be used in two different ways:

- Function Templates
- Class Templates

Use of templates

- Create a typesafe collection class (for example, a stack) that can operate on data of any type.
- Add extra type checking for functions that would otherwise take void pointers.
- Encapsulate groups of operator overrides to modify type behavior (such as smart pointers).

6. Function template

Ans :

A function template works in a similar to a normal function, with one key difference.

A single function template can work with different data types at once but, a single normal function can only work with one set of data types.

Normally, if you need to perform identical operations on two or more types of data, you use function overloading to create two functions with the required function declaration.

However, a better approach would be to use function templates because you can perform the same task writing less and maintainable code.

7. Overload of Template Functions.

Ans :

A template function also supports the overloading mechanism. It can be overloaded by a normal function or a template function. While invoking these functions, an error occurs if no accurate match is met. No implicit conversion is carried out in the parameters of template functions. The compiler observes the following rules for choosing an appropriate function when the program contains overloaded functions:

Searches for an accurate match of functions; if found, it is invoked searches for a template function through which a function that can be invoked with an accurate match can be generated; if found, it is invoked.

8. What is Class Template.

Ans :

Like function templates, you can also create class templates for generic class operations.

Sometimes, you need a class implementation that is same for all classes, only the data types used are different.

Normally, you would need to create a different class for each data type OR create different member variables and functions within a single class.

This will unnecessarily bloat your code base and will be hard to maintain, as a change is one class/ function should be performed on all classes/functions.

However, class templates make it easy to reuse the same code for all data types.

9. What is exception.

Ans :

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

Following are main advantages of exception handling over traditional error handling.

Separation of Error Handling code from Normal Code: In traditional error handling codes, there are always if else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try catch blocks, the code for error handling becomes separate from the normal flow.

10. Throwing mechanism.

Ans :

Exception handling in C++ revolves around these three keywords:

- **throw** – when a program encounters a problem, it throws an exception. The throw keyword helps the program perform the throw.
- **catch** – a program uses an exception handler to catch an exception. It is added to the section of a program where you need to handle the problem. It's done using the catch keyword.
- **try** – the try block identifies the code block for which certain exceptions will be activated. It should be followed by one/more catch blocks.
- n exception in C++ is thrown by using the throw keyword from inside the try block. The throw keyword allows the programmer to define custom exceptions.
- Exception handlers in C++ are declared with the catch keyword, which is placed immediately after the try block. Multiple handlers (catch expressions) can be chained - each one with a different exception type. Only the handler whose argument type matches the exception type in the throw statement is executed.

Choose the Correct Answers

1. Which type of function among the following shows polymorphism? [b]
   a) Inline function
   b) Virtual function
   c) Undefined functions
   d) Class member functions

2. Which among the following can show polymorphism? [c]
   a) Overloading ||
   b) Overloading +=
   c) Overloading <<
   d) Overloading &&

3. Syntax for Pure Virtual Function is _____. [c]
   a) virtual void show()==0
   b) void virtual show()==0
   c) virtual void show()=0
   d) void virtual show()=0

4. In a class, pure virtual functions in C++ is used [d]
   a) To create an interface
   b) To make a class abstract
   c) To force derived class to implement the pure virtual function
   d) All the above

5. From where does the template class derived? [c]
   a) Regular non-templated C++ class
   b) Templated class
   c) Both A or B
   d) None of the above

6. Why we use :: template-template parameter? [a]
   a) binding
   b) rebinding
   c) both binding & rebinding
   d) reusing

7. What is the syntax of class template? [a]
   a) template <paramaters> class declaration
   b) Template <paramaters> class declaration
   c) temp <paramaters> class declaration
   d) Temp <paramaters> class declaration

8. What is the correct syntax of defining function template/template functions? [d]
   a) template <class T> void(T a){cout<<a;}
   b) Template <class T> void(T a){cout<<a;}
   c) template <T> void(T a){cout<<a;}
   d) Template <T> void(T a){cout<<a;}

9. Which type of program is recommended to include in try block? [c]
   a) static memory allocation
   b) dynamic memory allocation
   c) const reference
   d) pointer

10. Which statement is used to catch all types of exceptions?
    a) catch()
    b) catch(Test t)
    c) catch(...)
    d) catch(Test)

Fill in the Blanks

1. Run time polymorphism is achieved only when a _____ is accessed through a pointer to the base class.
2. If a class contains pure virtual function, then it is termed as _____
3. When a virtual function is redefined by the derived class, it is called_____
4. Non-type template parameters provide the ability to pass a _____ at compile time.
5. _____ is used to create a function without having to specify the exact type.
6. The _____ of the object of template class varies depending on the type of template parameter passed to the class.
7. _____ parameter is legal for non-type template.
8. Return type of uncaught_exception() is _____
9. If inner catch handler is not able to handle the exception then_____
10. _____ is the basic of grouping standard exception classes.in c++?

Answers

1. Virtual function
2. Abstract Class
3. Overriding
4. Constant expression
5. Function template
6. Size
7. Pointer to member
8. Bool
9. Compiler will check for appropriate catch handler of outer try block
10. Namespace std