Object Oreinted Programming With C++

MODULE-4: Program development using Inheritance

Compiled by: Dr. Amita Jain

amita.jain@vsit.edu.in

VSIT | Vidyalankar School of Information Technology

NAAC Accredited College

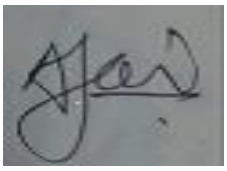
Vidyalankar School of Information Technology
Wadala (E), Mumbai
www.vsit.edu.in

VSIT | Vidyalankar School of Information Technology

NAAC Accredited College

Certificate

This is to certify that the e-book titled "OBJECT ORIENTED PROGRAMMING WITH C++" comprises all elementary learning tools for a better understating of the relevant concepts. This e-book is comprehensively compiled as per the predefined eight parameters and guidelines.



Signature

Dr. Amita Jain

Assistant Professor

Department of IT

Date: 23-12-2019

⚠ *DISCLAIMER: The information contained in this e-book is compiled and distributed for educational purposes only. This e-book has been designed to help learners understand relevant concepts with a more dynamic interface. The compiler of this e-book and Vidyalankar School of Technology give full and due credit to the authors of the contents, developers and all websites from wherever information has been sourced. We acknowledge our gratitude towards the websites YouTube, Wikipedia, and Google search engine. No commercial benefits are being drawn from this project.*

Unit IV Program development using Inheritance

Contents

- Introduction, understanding inheritance
- Advantages provided by inheritance
- choosing the access specifier
- Derived class declaration
- derived class constructors
- class hierarchies
- multiple inheritance
- multilevel inheritance
- containership, hybrid inheritance
- Exception Handling: Introduction
- Exception Handling Mechanism.
- Concept of throw & catch with example

Recommended Books

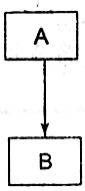
- **Texts Books:**
- Schaum's Outline Of Theory And Problems Of Programming With C++ .
  By John R.Hubbard, Tatamcgraw-Hill
- Object Oriented Programming with C++,
  By E.Balagurusamy,Fourth Edition, TATAMCGRAW-HILL.
- Object Oriented Programming with C++,
  By P. Sarang 2nd Edition,(PHI) EEE edition
- **Reference Books:**
- C++ programming , 3rd Edition, Bjarne Stroustrup
- Mastering C++, 2nd Edition, Venugopalan, TataMcgrawHill
- C++ Programming, , Robert Lafore,
- C++ for Beginners, P. M. Harwani, X-Team Series,
  Prerequisites and Linking

Unit IV	Pre-requisites	Sem. I	Sem. III	Sem. IV	Sem. V	Sem. VI
Program development using Inheritance	Basic C++	C	Python	Core Java	AWP, project	Project

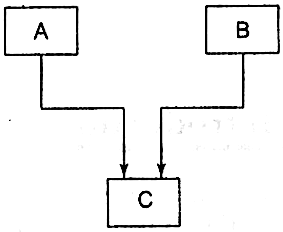
INHERITANCE – Reusability is another feature of OOPs. The C++ classes can be reused in several ways. Once a class has been written and tested it can be adapted by the other programmers to suit requirements. This is basically done by creating new classes, reusing the properties of the existing ones. The mechanism of deriving a new class from the old one is called inheritance or derivation. The old class is referred as base class and the new one is referred as derived class. The derived class inherits some or all the properties from the base class. A class can also inherit properties from more than one class or from more than one level. A derived class with only one base class is called as single inheritance and one with several base classes is called multiple inheritance. On the other hand the traits of one class may be inherited by more than one class. This process is called as hierarchical inheritance. The mechanism of deriving a class from another derived class is known as multilevel inheritance.

The mechanism of deriving a new class from an old one is called ***inheritance (or derivation)***. The old class if referred to as the ***basic class*** and the new one is called the ***derived class*** or ***subclass***.

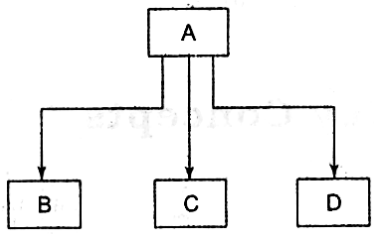
A class can also inherit properties from more than one class or from more than one Level. A derived class with only one base class, is called single inheritance and one with several base classes in called multiple inheritance. On the other hand, the traits of one class may be inherited by more than one class. This process is known as hierarchical inheritance. The mechanism of deriving a class from another 'derived class' is known as multilevel inheritance.



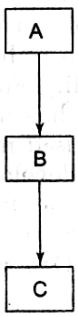
(a) Single Inheritance



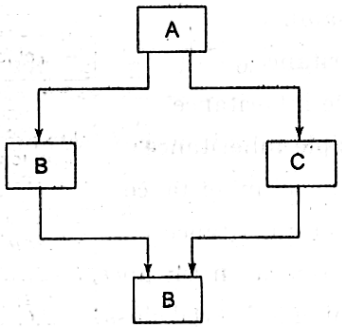
(b) Multiple Inheritance



(c) Hierarchical Inheritance



(d) Multilevel Inheritance



(e) Hybrid Inheritance

Fig. : Forms of inheritance.

DEFINING A DERIVED CLASS – A derived class is defined by specifying its relationship with the base class in addition to its own details. The general form of defining a derived class is:

Class derived class name: visibility mode base class name

{

members of derived class

}

The colon indicates that the derived class name is derived from the base class name. The visibility mode is either private or public where default is private. Examples:

Class ABC: private XYZ //private derivation

{

members of ABC

};

Class ABC: public XYZ //public derivation

{

members of ABC

};

Class ABC: XYZ //default private derivation

{

members of ABC

};

When a base class is privately inherited by a derived class public members of the base class become private members of the derived class and therefore the public members of the base class can be accessed only by the members functions of the derived class. On the other hand if the base class is publicly inherited public members of the base class become public members of the derived class and are therefore accessible to the objects of the derived class. (SINGINHT.CPP)

//////////SINGLE INHERITANCE : PUBLIC///////////////

#include<iostream.h>

#include<conio.h>

class B

{

```
        int a;//private not inheritable
    public:
        int b;//public ready for inherit
        void get_ab();
        int get_a();
        void show_a();
};

class D:public B //public derivation
{
        int c;
    public:
        void mul();
        void display();
};

void B::get_ab()
{
    a=5;
    b=10;
}

int B::get_a()
{
    return a;
}

void B:: show_a()
{
    cout<<" a = "<<a<<endl;
}
```

```
void D::mul()
{
   c=b*get_a();
}

void D:: display()
{
   cout<<" a = "<<get_a()<<endl;
   cout<<" b = "<<b<<endl;
   cout<<" c = "<<c<<endl;
}

void main()
{
   D d;
   clrscr();
   d.get_ab();
   d.mul();
   d.show_a();
   d.display();
   d.b=20;
   d.mul();
   d.display();
   getch();
}
```

(SININHPR.CPP)

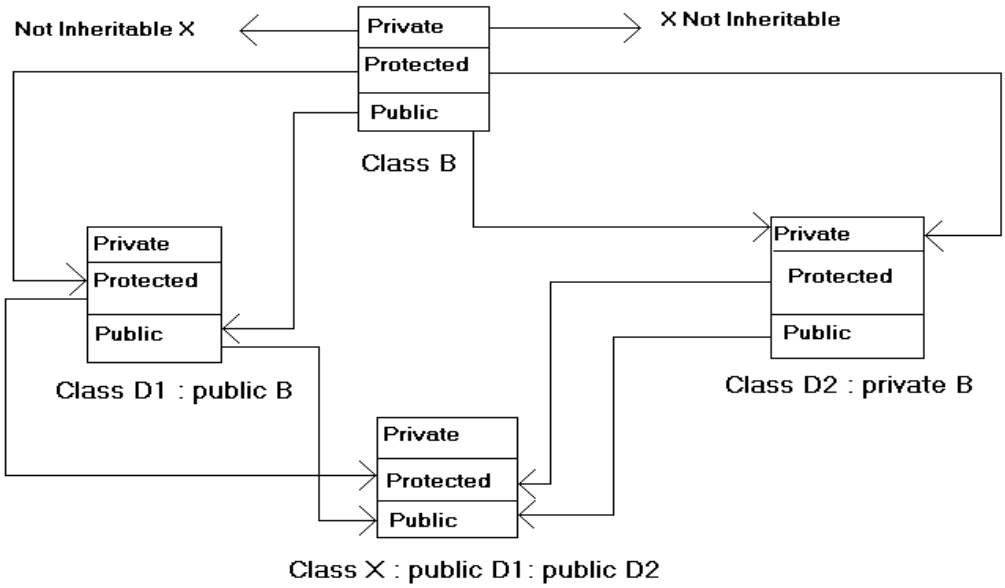
```
//////////////SINGLE INHERITANCE : PRIVATE//////////////////
#include<iostream.h>
#include<conio.h>
class B
```

```
{
   int a;
   public:
         int b;
         void get_ab();
         int  get_a();
         void show_a();
};

class D:private B //private derivation
{
   int c;
   public:
         void mul();
         void display();
};

void B::get_ab()
{
   cout<<" \n Enter values for a and b : ";
   cin >> a>> b;
}

int B::get_a()
{
   return a;
}

void B::show_a()
{
   cout<<" a = "<<a<<endl;
```

```
}

void D::mul()
{
    get_ab();
    c=b*get_a(); // a cannot be used directly
}

void D :: display()
{
    show_a(); //outputs the value of a
    cout<<"b = "<<b<<endl;
    cout<<"c = "<<c<<endl;
}

void main()
{
D d;

clrscr();
    d.mul();    //d.get_ab(); WON'T WORK
    d.display();//d.show_a();WON'T WORK

    //d.b = 20; WON'T WORK BECAUSE B HAS BECOME PRIVATE
    d.mul();
    d.display();
    getch();
}
```

Base Class Visibility	Derived Class Visibility	
	Public Derivation	Private Derivation
Private	Not Inherited	Not Inherited
Protected	Protected	Private
Public	Public	Private

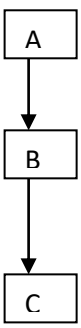
PROTECTED ACCESS SPECIFIER – C++ provides a third visibility modifier called as protected, which serve a limited purpose in inheritance. A member declared as protected is accessible by the member function within its class and any class immediately derived from it. It cannot be accessed by the functions outside these two classes. When a protected member is inherited in public mode it becomes protected in the derived class to and therefore it is accessible by member functions of the derived class. It is also ready for further inheritance. A protected member if inherited in private mode it becomes private in the derived class. Although it is available to member functions of the derived class it is not available for further inheritance.




```
///////PROTECTED ACCESS SPECIFIER//////////
#include<iostream.h>
#include<conio.h>
class base
{
        private:
          int pri;
        protected:
           int pro;
        public:
           int pub;
           base()
           {
              pri=0;
              pub=0;
              pro=0;
           }
```

```
        void show()
        {
            cout<<"\n pri = "<<pri;
            cout<<"\n pro = "<<pro;
            cout<<"\n pub = "<<pub;
        }
};

class derived : public base
{
   public:
   void incr()
   {
         //pri++;
         pro++;
         pub++;
   }
};

class derived2 : public derived
{
   public:
   void incr2()
   {
         //pri++;
         pub++;
         pro++;
   }
};

void main()
{
   derived2 d2;
   clrscr();
   d2.incr();
   //d2.pro++;//outside the scope of the class
   d2.pub++;
   d2.incr2();
   d2.show();
   getch();
}
```

MULTILEVEL INHERITANCE – In this type of inheritance the second class is derived from the first class and the third class is derived from the second. The first class becomes the grandfather the second becomes the father and the third becomes the child. The grandfather is also called as base class and the father is called as intermediate base class and the child is called as derived class. This process can be extended to any number of levels. (MULLEVEL.CPP)



```
//////MULTILEVEL INHERITANCE///////
#include<iostream.h>
#include<conio.h>
class student
{
   //private:
   protected:
         int roll;
   public:
         void getno()
         {
            cin>>roll;
         }
         void putno()
         {
            cout<<"\n Roll Number : "<<roll;
         }
};

class test :public student //first level derivation
{
   protected:
         int sub1,sub2;
   public:
      void getmarks()
      {
            cin>>sub1>>sub2;
      }
      void putmarks()
      {
            cout<<"\n Marks in 1 : "<<sub1;
```

```
            cout<<"\n Marks in 2 : "<<sub2;
        }
        void show()
        {
                cout<<"\n Base roll : "<<roll;
        }
};

class result:public test     //second level derivation
{
    int total; //private by default
    public:
            void display();
};

void result::display()
{
    total=sub1+sub2;
    putno();
    show();
    putmarks();
    cout<<"\n Total = : "<<total;
}

void main()
{
    result student1;//student1 created
    student1.getno();
    student1.getmarks();
    student1.display();
    getch();
```

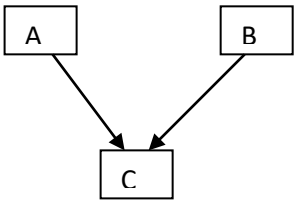
}

MULTIPLE INHERITANCE – When a class inherits the attributes from more than one class it is called as multiple inheritance. It allows us to combine features of several existing classes. The general from of multiple inheritance is:

Derived class name: visibility mode base class 1,visibility base class 2
{

Members of this class

}; (MULTIPLE.CPP)



https://www.youtube.com/watch?v=HUHnoruSXEk multiple inheritance

```
//////////MULTIPLE INHERITANCE///////////
#include<iostream.h>
#include<conio.h>
class M
{
   protected:
        int m;
   public:
        void getm();
};

class N
{
   protected:
        int n;
   public:
        void getn();
};

class P:public M,public N
{
   public:
        void display();
};

void M::getm()
{
   cin>>m;
}
```

```
void N::getn()
{
   cin>>n;
}

void P::display()
{
   cout<<m<<n<<m*n;
}

void main()
{
   P p;
   clrscr();
   p.getm();
   p.getn();
   p.display();
   getch();
}
```

HYBRID INHERITANCE – In this type of inheritance various types of above inheritances are combined together. (HYBRID.CPP)

```
//////////HYBRID INHERITANCE///////////////
#include<iostream.h>
#include<conio.h>
class student
{
   protected:
        int roll;
   public:
        void getno()
```

```
        {
            cin>>roll;
        }
        void putno()
        {
            cout<<"\n "<<roll;
        }
};

class test : public student
{
    protected:
        int part1,part2;
    public:
        void getmarks()
        {
            cin>>part1;
            cin>>part2;
        }
        void putmarks()
        {
           cout<<endl;
           cout<<part1<<endl;
           cout<<part2<<endl;
        }
};

class sports
{
    protected:
        int score;
```

```
    public:
            void getscore()
            {
                cin>>score;
            }
            void putscore()
            {
                cout<<"\n "<<score;
            }
};

class result:public test,public sports
{
    int total;
    public:
      void display()
      {
            total=part1+part2+score;
            cout<<"\n Display the roll no"<<endl;
            putno();
            cout<<"\n Displaying Marks "<<endl;
            putmarks();
            cout<<"\n Displaying score "<<endl;
            putscore();
            cout<<"\n Total = : "<<total;
      }
};

void main()
{
    result student1;
```

```
    clrscr();
    student1.getno();
    student1.getmarks();
    student1.getscore();
    student1.display();
    getch();
}
```

CONSTRUCTORS AND DESTRUCTORS IN DERIVED CLASSES – Constructors play an important role in initializing objects. As long as no base class constructor takes any arguments the derived class need not have a constructor function. However if any base class contains a constructor with one or more arguments then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructors. While applying inheritance we create object of derived class. So the derived class has to pass arguments to the base class constructor. When both the derived and the base classes contain constructors the base class constructor is executed first and then the derived class constructor. The execution of the constructors is given below

Method Of Inheritance	**Order of execution**
Class B : public A **{** **};**	**A (); base constructor** **B (); derived constructor**
Class A : public B, public C **{** **};**	**B (); base first** **C (); base second** **A (); derived constructor**
Class A : public B, virtual public C **{** **};**	**C (); virtual base** **B (); ordinary base** **A (); derived**

```
#include<iostream.h>
#include<conio.h>
class alpha
{
   private:
        int x;
```

```
    public:
        alpha(int i)
        {
            x=i;
            cout<<"\n\t Alpha Initialized ";
        }
        void showx()
        {
            cout<<"\n\t Value of x is : "<<x<<endl;
        }
};

class beta
{
    private:
        float y;
    public:
        beta(float j)
        {
            y=j;
            cout<<"\n\t Beta Initialized ";
        }
        void showy()
        {
            cout<<"\n\t Value of y is : "<<y<<endl;
        }
};

class gamma:public beta,public alpha
{
    private:
```

```
        int m,n;
    public:
        gamma(int a,float b,int c,int d):alpha(a),beta(b)
        {
            m=c;
            n=d;
            cout<<"\n\t Gamma Initialized ";
        }
        void showmn()
        {
            cout<<"\n\t Value of m is : "<<m<<endl;
            cout<<"\n\t Value of n is : "<<n<<endl;
        }
};

void main()
{
    gamma g(5,10.75,20,30);
    cout<<endl;
    g.showx();
    g.showy();
    g.showmn();
    getch();
}
```

NESTING OF CLASSES – C++ supports another way of inheriting properties of one class into another. This approach takes a view that an object can be a collection of many other objects. That is a class can contain objects of another class as its members. This kind of relationship is called as containership or nesting.

(OBJMEMPR.CPP)

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

```
#include<stdio.h>
class carburettor
{
   private:
        char type;
        int cost;
   public:
        void setdata()
        {
           cout<<"\n\t Enter type : ";
           cin>>type;
           cout<<"\n\t Enter cost : ";
           cin>>cost;
        }
        void putdata()
        {
           cout<<"\n\t Type : "<<type;
           cout<<"\n\t Cost : "<<cost;
        }
};

class car
{
   private:
        char model[25],dtype[20];
        carburettor c;//embedded object
   public:
        void setcardata()
        {
           c.setdata();
           cout<<"\n\t Enter model : ";
```

```
            gets(model);
            cout<<"\n\t Enter drive type : ";
            gets(dtype);
        }
        void putcardata()
        {
            c.putdata();
            cout<<"\n\t Model : "<<model;
            cout<<"\n\t Drive type : "<<dtype;
        }
};

void main()
{
   car mycar;
   clrscr();
   mycar.setcardata();
   mycar.putcardata();
   getch();
}
```

(OBJMEMPU.CPP)

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
class carburettor
{
   private:
        char type;
        int cost;
   public:
```

```
        void setdata()
        {
            cout<<"\n\t Enter type : ";
            cin>>type;
            cout<<"\n\t Enter cost : ";
            cin>>cost;
        }
        void putdata()
        {
            cout<<"\n\t Type : "<<type;
            cout<<"\n\t Cost : "<<cost;
        }
};

class car
{
    private:
        char model[25],dtype[20];
    public:
        void setdata()
        {
            cout<<"\n\t Enter model : ";
            gets(model);
            cout<<"\n\t Enter drive type : ";
            gets(dtype);
        }
        void putdata()
        {
            cout<<"\n\t Model : "<<model;
            cout<<"\n\t Drive type : "<<dtype;
        }
```

```
        carburettor c;//embedded object
};

void main()
{
   car mycar;
   clrscr();
   mycar.c.setdata();
   mycar.setdata();
   mycar.c.putdata();
   mycar.putdata();
   getch();
}
```

POINTERS TO DERIVED CLASS – We can use pointers not only to the base objects but also to the objects of the derived class. Therefore a single pointer can be made to point to objects belonging to different classes. But in this type of pointing the pointer can access only the inherited members and not the original members of the derived class. Hence we must use another pointer of derived class. (PTRTODER.CPP)

```
#include<iostream.h>
#include<conio.h>
class BC
{
   public:
        int b;
        void show()
        {
           cout<<"\n\t b = "<<b;
        }
};

class DC:public BC
{
```

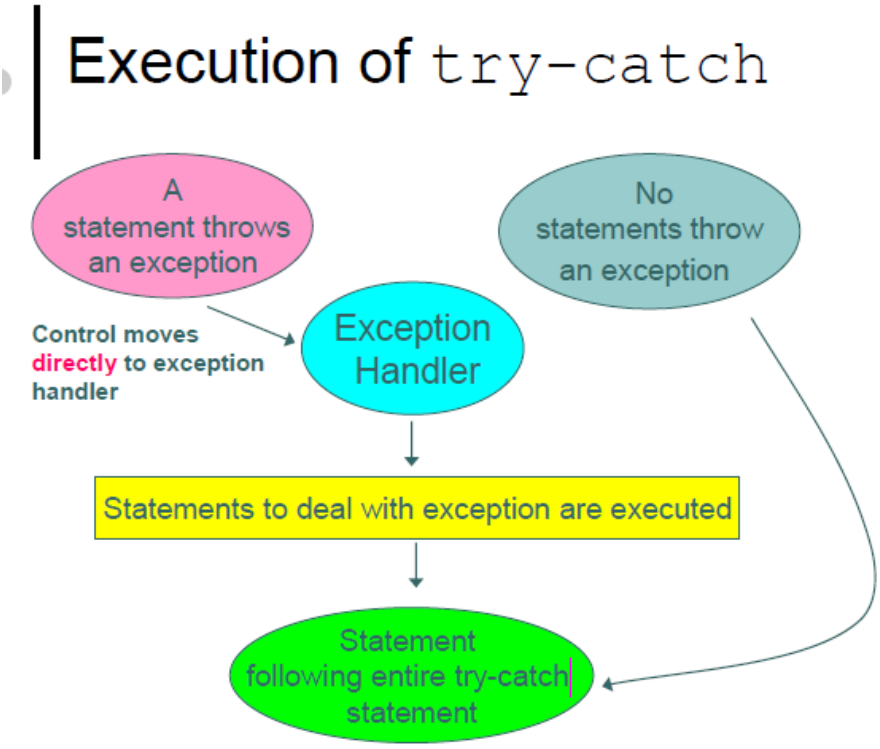
```
    public:
        int d;
        void show()
        {
            cout<<"\n\t b = "<<b;
            cout<<"\n\t d = "<<d;
        }
};

void main()
{
    clrscr();
    BC *bptr; //base pointer
    BC base;
    bptr=&base;//base address
    bptr->b=100;//access to BC via base pointer
    cout<<"\n\t bptr points to the base object ";
    bptr->show();
    DC derived; //derived class
    bptr=&derived;//address of derived object
    bptr->b=200;//access to DC via base pointer
    /*bptr->d =300; WONT WORK */
    cout<<"\n\t bptr points to the derived object ";
    bptr->show();
    DC *dptr;//derived type pointer
    dptr=&derived;
    dptr->d=300;
    cout<<"\n\t dptr is derived type pointer ";
    dptr->show();
    cout<<"\n\t Using ((DC *)bptr ";
    ((DC *)bptr)->d=400;
```

```
    ((DC *)bptr)->show();
    getch();
}
```

Exception handling

C++ provides a build-in error handling mechanism that is called *exception handling*. Using exception handling, you can more easily manage and respond to run-time errors. C++ exception handling is built upon three keywords: **try**, **catch** and **throw**. In the most general terms, program statements that you want to monitor for exceptions are contained in a **try** block. If an exception (i.e. an error) occurs within the **try** block, it is thrown (using **throw**). The exception is caught, using **catch**, and processed.

As stated, any statement that throws an exception must have been executed from within a **try** block (a function called from within a **try** block can also throw exception.) Any exception must be caught by a **catch** statement that immediately follows the **try** statement that throws the exception. The general form if **try** and **catch** are as shown:




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

```
catch(typeN arg) {
// catch block
}
```

The **try** block must contain the portion of your program that you want to monitor for errors. This can be as specific as monitoring a few statements within one function or as all encompassing as enclosing the **main( )** function code within the **try** block (which effectively causes the entire program to be monitored).

> When an exception is thrown, it is caught by its corresponding **catch** statement, which processes the exception. There can be more than one **catch** statement

associated with a **try**. The **catch** statement that is used is determined by the type of the exception. That is, if the data type specified by a **catch,** matches that of the exception, that **catch** statement is executed (all other are bypassed). When an exception is caught, *arg* will receive its value. If you don't need access to the exception itself, specify only *type* in the **catch** clause (*arg* is optional). Any type of data can be caught, including classes that you create. In fact, class types are frequently used as exceptions.

The general form of a **throw** statement is

```
throw exception;
```

throw must be executed either from within the **try** block or from any function that the code within the block calls (directly or indirectly). *exception* is the value thrown.

If you throw an exception for which there is no applicable **catch** statement, an abnormal program termination might occur. If your compiler complies with Standard C++, throwing an unhandled exception causes the standard library function **terminate( )** to be invoked. By default, **terminate( )** calls **abort( )** to stop your program, but you can specify your own termination handler, if you like. You will need to refer to your compiler's library reference for details.

```
// A simple exception handling example
#include <iostream>
using namespace std;
int main( ) {
cout << "Start\n";
try { // start a try block
cout << "Inside try block\n";
throw 10; // throw an error
cout << "This will not execute\n";
}
catch( int i) { // catch an error
cout << "Caught One! Number is: ";
cout << i << "\n";
}
cout << "end";
return 0;
}
```

This program displays the following:

```
start
Inside try block
Caught One! Number is: 10
end
```

As you can see, once an exception has been thrown, control passes to the **catch** expression and the **try** block is terminated. That is **catch** is not called. Rather, program execution is transferred to it. (The stack is automatically reset as needed to accomplish this) Thus, the **cout** statement following the **throw** will never execute.

After the **catch** statement executes, program control continues with the statements following the **catch.** Often, however, a **catch** block will end with a call to **exit( )** or **abort( )**, or some other function that causes program termination because exception handling is frequently used to handle catastrophic errors.

Remember that the type of the exception must match the type specified in a **catch** statement.

An exception can be thrown from a statement that is outside the **try** block as long as the statement is within a function that is called from within the **try** block.

```
// Throwing an exception from a function outside
// the try block
#include <iostream>
using namespace std;
void Xtest(int test) {
cout << "Inside Xtest, test is: " << test << \n";
if (test) throw test;
}
int main( ) {
cout << "start\n";
try { // start a try block
cout << "Inside try block\n";
Xtest(0);
Xtest(1);
Xtest(2);
}
catch (int i) { // catch an error
cout << "Caught one! Number is: ";
cout << i << "\n";
  }


cout << "end";
return 0;
}
```

This program displays:

```
start
Inside try block
Inside Xtest, test is: 0
Inside Xtest, test is: 1
Caught one! Number is: 1
end
```

A **try** block can be localised in a function. When this is the case, each time the function is entered, the exception handling relative to that function is reset. Here is an example:

```
#include <iostream>
using namespace std;
// A try/catch can be handle inside a function
// other than main( ).
void Xhandler(int test) {
try {
if (test) throw test;
}
catch(int i) {
cout << "Caught one! Ex. #: " << i << "\n";
}
}
int main( ) {
cout << "start";
Xhandler(1);
Xhandler(2);
Xhandler(0);
Xhandler(3);
cout << "end";
return 0;
}
```

This program displays:

```
start
Caught one! Ex. #: 1
Caught one! Ex. #: 2
Caught one! Ex. #: 3
end
```

As you can see, three exceptions are thrown. After each exception, the function returns. When the function is called again, the exception handling is reset.

As stated before, you can have more than one **catch** associated with a **try**. In fact, it is common to do so. However each **catch** must catch a different type of exception. For example,

```
#include <iostream>
using namespace std;
// Different type of exception can be caught.
void Xhandler(int test) {
try {
if (test) throw test;
else throw "Value is zero";
}
catch(int i) {
cout << "Caught one! Ex. #: " << i << "\n";
}
catch(char *str) {
cout << "Caught a string: " << str << "\n";
}
}
int main( ) {
cout << "start";
Xhandler(1);
Xhandler(2);
Xhandler(0);
Xhandler(3);
cout << "end";
return 0;
}
```

This program displays:

```
start
Caught one! Ex. #: 1
Caught one! Ex. #: 2
Caught one! Ex. #: 3
end
```

More about exception handling

In some circumstances you will want an exception handler to catch all exceptions instead of just a certain type. Simply use this form of **catch**:

https://www.youtube.com/watch?v=eTy5ZP_dRIY exception handling

```
catch(...) {
// process all exception
}
```

Also, you can control what type of exceptions a function can throw outside itself. In fact, you can also prevent a function from throwing any exceptions whatsoever. To apply these restrictions, you must add a **throw** clause to the function definition. The general form is as follows,

```
ret-type-func-name(arg-list) throw(type-list)
{
// ....
}
```

Here only those data types contained in the comma-separated list *type-list* may be thrown by the function. Throwing any other type of expression will cause the program termination. If you don't want a function to be able to throw *any* exceptions, use an empty list.

If your compiler complies with Standard C++, when a function attempts to throw a disallowed exception the standard library function **unexpected( )** is called. By default, this causes the **terminate( )** function to be called, which causes abnormal program termination. However, you can specify your own termination handler, if you like. You will need to refer to your compiler documentation for directions.

If you wish to rethrow an exception from within an exception handler, you can do so by simply calling **thrown**, by itself, with no exception. This causes the current exception to be passed on to an outer **try/catch** sequence.

```
// Catches all exceptions
#include <iostream>
using namespace std;
void Xhandler(int test) {
try {
if (test==0) throw test; // throw int
if (test==1) throw 'a'; // throw char
if (test==2) throw 123.23;// throw double
}
catch(...) { // catch all exceptions
cout << "Caught one!\n";
}
}
int main( ) {
cout << "start\n";
Xhandler(0);
Xhandler(1);
Xhandler(2);
cout << "end";
return 0;
}
```

This program displays:

```
start
Caught one!
Caught one!
Caught one!
end
```

One very good use for **catch(...)** is as last **catch** of a cluster of catches.

```
// Uses catch(...) as default
#include <iostream>
using namespace std;
void Xhandler(int test) {
```

```
try {
if (test==0) throw test; // throw int
if (test==1) throw 'a'; // throw char
if (test==2) throw 123.23;// throw double
}
catch(int i) { // catch an int exception
cout << "Caught " << i << "\n";
}
catch(...) { // catch all other exceptions
cout << "Caught one!\n";
}
}
int main( ) {
cout << "start\n";
Xhandler(0);
Xhandler(1);
Xhandler(2);
cout << "end";
return 0;
}
```

This program displays:

```
start
Caught 0
Caught one!
  Caught one!


Caught one!
end
```

The following program shows how to restrict the types of exceptions that can be thrown from a function:

```
// Restricting function throw types
#include <iostream>
using namespace std;
// can only throw ints, chars and doubles
void Xhandler(int test) throw(int, char, double) {
if (test==0) throw test; // throw int
if (test==1) throw 'a'; // throw char
if (test==2) throw 123.23;// throw double
}
int main( ) {
cout << "start\n";
try {
Xhandler(0); // also try passing 1 and
// 2 to Xhandler( )
}
catch(int i) {
cout << "Caught int\n";
}
catch(char c) {
cout << "Caught char\n";
}
catch(double c) {
```

```
cout << "Caught double\n";
}
cout << "end";
return 0;
}
```

Finally, here is an example of rethrowing an exception. An exception can only be rethrown from within a **catch** block. When you rethrow an exception, it will not be recaught by the same **catch** statement. It will propagate to an outer **catch** statement.

```
// Rethrowing an exception
#include <iostream>
using namespace std;
void Xhandler( ) {
try {
throw "hello"; // throw char *
catch(char *) { // catch a char *
cout << "Caught char * inside Xhandler\n";
throw ; // rethrow char * out of function
}
}
int main( ) {
cout << "start\n";
try {
Xhandler( );
}
catch(char *) {
cout << "Caught char * inside main\n";
}
cout << "end";
return 0;
}
```

This program displays:

```
start
Caught char * inside Xhandler
Caught char * inside main
end
```

Handling exceptions thrown by new

As you know, the modern specification for the **new** operator states that it will throw an exception of an allocation request fails.

In Standard C++, when an allocation request is not honoured, **new** throws a **bad_alloc** exception. If you don't catch this exception, your program will be terminated. Although this behaviour is fine for short sample program, in real applications you must catch this exception and process it in some rational manner. To have access to this exception, you must include the header **<new>** in your program.

Note that originally this exception was called **xalloc**, and many old compilers still use the older name. However, **bad_alloc** is the name specified by Standard C++, and it is the name that will be used in future.

In Standard C++, it is also possible to have **new** return null instead of throwing an exception when an allocation failure occurs. This form of new is most useful when you are compiling older code with a modern C++ compiler. It is also

valuable when you are replacing calls to **malloc( )** with **new**. This form of **new** is shown here:

p-var = **new**(**nothrow**) *type*;

Here *p-var* is a pointer variable of *type*. The **nothrow** from **new** works like the original version of **new,** from years ago. Since, it returns null on failure, it can be 'dropped into' older code and you won't have to add exception handling. However, for new code, exceptions provide a better alternative.

```
// Example of new that uses a try/catch to
// monitor for allocation failure
#include <iostream>
#include <new>
using namespace std;
int main( ) {
int *p;
try {
p = new int; // allocate memory for int
}
catch (bad_alloc xa) {
cout << "Allocation failure\n";
return 1;
}
for (*p=0; *p<10; (*p)++)
cout << *p << " ";
delete p; // free memory
return 0;
}
```

Since the the above program is unlikely to fail under normal circumstances, the following program forces an allocation failure. It does this by allocating memory until it is exhausted.

```
// Force an allocation failure
#include <iostream>
#include <new>
using namespace std;
int main( ) {
int *p;
// this will eventually run out of memory
do {
try {
p = new double(100000);
}
catch (bad_alloc xa) {
cout << "Allocation failure\n";
return 1;
}
cout << "Allocation OK.\n";
} while (p);
return 0;
}
```

The following program shows how to use the **new(nothrow)** alternative.

```
// Demonstrate the new(nothrow) alternative and
// force a failure
```

```
#include <iostream>
#include <new>
using namespace std;
int main( ) {
int *p;
// this will eventually run out of memory
do {
p = new(nothrow) double(100000);
if (p) cout << "Allocation OK.\n";
else cout << "Allocation Error.\n";
} while (p);
return 0;
}
```

As shown here, when you use **nothrow** approach, you must check the pointer returned by **new** after each allocation request.

Questions:

Basic

1. Define Inheritance. Explain the different types of inheritance.
2. Describe the concept of composition with an example.
3. Explain the concept of single level inheritance with an example.
4. Write a short note on protected class members
5. Explain the concept of private and protected access.
6. Describe exceptions in C++?
7. How to implement exception handling in C++?

Intermediate:

1. Explain the concept of overriding and dominating the inherited members.
2. Explain Exception handling for a class with an example?
3. Explain benefits of Exception Handling?

Advanced:

1. Define a base class called animal with following members

    a. A string type data member to store the name of the animal
    b. An integer member to store the age of animal in years
    c. A method to display the name and age of the animal

    Derive 2 classes named cat and dog. Display the content of the object by calling the display method on the derived objects.

Multiple Choice Questions:

1. What are the things are inherited from the base class?
   a) Constructor and its destructor
   b) Operator=() members
   c) Friends
   d) All of the mentioned

2. Which design patterns benefit from the multiple inheritance?
   a) Adapter and observer pattern
   b) Code pattern
   c) Glue pattern
   d) None of the mentioned

3. Which of the following advantages we lose by using multiple inheritance?
   a) Dynamic binding
   b) Polymorphism
   c) Both a & b
   d) None of the mentioned

4. Which symbol is used to create multiple inheritance?
   a) Dot
   b) Comma
   c) Dollar
   d) None of the mentioned

5. What is meant by multiple inheritance?
   a) Deriving a base class from derived class
   b) Deriving a derived class from base class
   c) Deriving a derived class from more than one base class
   d) None of the mentioned

6. When a child class inherits traits from more than one parent class, this type of inheritance is called _____ inheritance.

   a) Hierarchical
   b) Hybrid
   c) Multilevel
   d) Multiple

7. The derivation of Child class from Base class is indicated by ____ symbol.

a) ::
b) :
c) ;
d) |

8. class X, class Y and class Z are derived from class BASE. This is _____ inheritance

a) Multiple
b) Multilevel
c) Hierarchical
d) Single

9. During a class inheritance in CPP, if the visibility mode or mode of derivation is not provided, then by default visibility mode is_____.

a) public
b) protected
c) private
d) friend

10. If the derived class is struct, then default visibility mode is_____

a) public
b) protected
c) private
d) struct can't inherit class

11. Can struct be used as Base class for inheritance ?

a) Yes
b) No

12. Private members of the class are not inheritable.

a.True
b. **False**

13. When a base class is privately inherited by the derived class, then_____

a) protected members of the base class become private members of derived class
b) public members of the base class become private members of derived class
c) both a and b
d) only b

14. What is difference between protected and private access specifiers in inheritance?

a) private member is not inheritable and not accessible in derived class
b) protected member is inheritable and also accessible in derived class
c) Both are inheritable but private is accessible in the derived class
d) Both are inheritable but protected is not accessible in the derived class

15. If base class has constructor with arguments, then it is _____ for the derived class to have constructor and pass the arguments to base class constructor.

a) Optional
b) Mandatory
c) Compiler dependent
d) Error

16. Generic catch handler is represented by _____
a. catch(..,)
b. catch(---)
c. catch(…)
d. catch(void x)

17. If inner catch handler is not able to handle the exception then_____ .

a. Compiler will look for outer try handler
b. Program terminates abnormally
c. Compiler will check for appropriate catch handler of outer try block
d. None of these

18. **How can we restrict a function to throw certain exceptions?**

a. Defining multiple try and catch block inside a function
b. Defining generic function within try block
c. **Defining function with throw clause**
d. It is not possible in CPP to restrict a function

19. **The code of statements which may cause abnormal termination of the program should be written under_____ block.**

a. Try
b. catch
c. Finally
d. None of these

20. **In nested try block, if inner catch handler gets executed, then _____ .**

a. Program execution stops immediately.
b. Outer catch handler will also get executed.
c. **Compiler will jump to the outer catch handler and then executes remaining executable statements of main().**
d. Compiler will execute remaining executable statements of outer try block and then the main().

21. **Throwing an unhandled exception causes standard library function then _____ to be invoked.**
a. stop()
b. aborted()
c. terminate()
d. Abandon()

22. **Exception handlers are declared with _____ keyword.**

a. Try
b. catch
c. throw
d. Finally

23. **An exception is thrown using _____ keyword in CPP.**

a. throws
b. throw
c. threw
d. Thrown