OBJECT ORIENTED PROGRAMMING (PCC-CS503)

Unit – 4

More extensions to C in C++ to provide OOP Facilities

Scope of Class

A class can have either:

1. Local scope (local class)
2. Global scope (global class)

Local Class: A class declared inside a function is known as a local class in C++ as it is local to that function.

An example of a local class is given as follows.

```
#include<iostream.h>
void func()          //some function
{
   class LocalClass        //local class
   {...
   };
}
int main()
{
   return 0;
}
```

Local classes must abide by the following rules:

1. A local class name can only be used in its function and not outside it i.e. the objects of such a class are local to the function scope.
2. The methods of a local class must be defined inside the local class only and not outside it.
3. A local class cannot have static data members but it can have static functions.
4. Local classes can access global types, variables and functions.
5. Local classes can access other local classes of same function.

A program that demonstrates a local class in C++ is given as follows.

```
#include<iostream.h>

void func()
{
   class LocalClass
   {
      private:
```

```
         int num;
      public:
         void getdata(int n)
         {
            num = n;
         }
         void putdata()
         {
         cout<<"The number is "<<num;
         }
   };
   LocalClass obj;
   obj.getdata(7);
   obj.putdata();
}
int main()
{
   cout<<"Demonstration of a local class"<<endl;
   func();
   return 0;
}
```

O/P:

```
Demonstration of a local class
The number is 7
```

Global Class: A class defined outside all methods is a global class because its objects can be created from anywhere in the program and its members can be accessed from anywhere, i.e. either inside or outside the class.

A program that demonstrates a global class in C++ is given as follows.

```
#include<iostream.h>
class GlobalClass   //local class
{...
}
GlobalClass g1;     //local object
;
GlobalClass g2;     //global object
}
int main()
{
   GlobalClass g3;  //local object
   return 0;
}
```

Example for static functions and static data members in Local classes:

```
  #include <iostream>
```

```
using namespace std;

void fun()
{
      class Test // local to fun
      {
      //static int a;      //NOT ALLOWED!
      public:
             static void method()
             {
                    cout << "Local Class method() called";
             }
      };

      Test::method();
}

int main()
{

      fun();
      return 0;
}

O/P:
Local Class method() called
```

Lab assignment:

1. Explore each of the statements stated below and give an appropriate example for the each with regard to Local classes:

a) A local class name can only be used in its function and not outside it i.e. the objects of such a class are local to the function scope.

b) The methods of a local class must be defined inside the local class only and not outside it.

c) A local class cannot have static data members but it can have static functions.

d) Local classes can access global types, variables and functions.

e) Local classes can access other local classes of same function.

2. Explore and write an appropriate example for the fact that "Global class members can be accessed from anywhere".

Ques: Can a local class have a global object?

Scope Resolution Operator

Scope Resolution Operator: is basically an operator used in order to resolute the scope. Thus, Scope resolution operator is used to get the hidden names due to variable scopes so that you can still use them.

In C++, the scope resolution operator can be used as both unary (:) and binary (::)

Scope resolution operator in C++ can be used for:

- Accessing a global variable when there is a local variable with same name
- Defining a function outside a class
- Accessing static member function and static variables
- In case of multiple Inheritance
- Namespace

Accessing a global variable when there is a local variable with same name:

```
#include<iostream.h>

int num = 30;  // Initializing a global variable num

int main()
{
  int num = 10; // Initializing the local variable num
  cout << "Value of global num is " << ::num;
  cout << "nValue of local num is " << num;
  return 0;
}
```

Defining a function outside a class:

```
#include<iostream.h>
using namespace std;
class Bike
{
public:
// Just the Function Declaration
void Speed();
};

// Defining the Speed function outside Bike class using ::
void Bike::Speed()
{
cout << "Speed of Bike is 90 KMPH";
}

int main()
{
Bike bike;
bike.Speed();
return 0;
}
```

Accessing static member function and static variables:

```
#include <iostream>

using namespace std;

class Demo
{
        private:
                //static data members
                static int X;
```

```
                static int Y;

        public:
        //static member function
        static void  Print()
        {
                cout <<"Value of X: " << X << endl;
                cout <<"Value of Y: " << Y << endl;
        }
};

//static data members initializations
int Demo :: X =10;
int Demo :: Y =20;


int main()
{
        Demo OB;
        //accessing class name with object name
        cout<<"Printing through object name:"<<endl;
        OB.Print();

        //accessing class name with class name
        cout<<"Printing through class name:"<<endl;
        Demo::Print();

        return 0;
}
```

O/P:

```
Printing through object name:
Value of X: 10
Value of Y: 20
Printing through class name:
Value of X: 10
Value of Y: 20
```

NOTE: According to the rule of static in C++, only static member function can access static data members. Non-static data member can never be accessed through static member functions.

In case of multiple Inheritance

If you have two parent classes with same variable names and you are inheriting both of them in the child class, then you can use scope resolution operator with the class name to access the individual variables.

Example

```
#include<iostream.h>

class Parent1
{
protected:
int num;
public:
```

```
Parent1()
{ num = 100; }
};
class Parent2
{
protected:
int num;
public:
Parent2()
{ num = 200; }
};
class Child: public Parent1, public Parent2
{
public:
void function()
{
cout << "Parent1's num is " << Parent1::num;
cout << "nParent2's num is " << Parent2::num;
}
};
int main()
{
Child obj;
obj.function();
return 0;
}
```

Namespace

Suppose we have two namespaces & both contains class with same name. So to avoid any conflict we can use namespace name with the scope resolution operator. In the below example we are using ***std::cout***.

Example

```
#include<iostream>
int main(){
std::cout << "Hello" << std::endl;
}
```

Member Function of a Class

- Member functions are the functions which are the members of a particular class and it works on the data members of the class.
- The definition of member functions can be inside or outside the definition of class.

```
class className
{
returnType MemberFunction(arguments)
{
 //function body
}
. . . .
};
```

- If the member function is defined inside the class then it can be defined directly (i.e. without declaring the function)

```
class Cube
{
    public:
    int side;
    int getVolume()
    {
        return side*side*side;      //returns volume of cube
    }
};
 int main()
 {
     Cube C1;
     C1.side = 4;    // setting side value
     cout<< "Volume of cube C1 = "<< C1.getVolume();
 }
```

- If the member function is defined outside the class, then we have to use the scope resolution : : operator along with class name while defining the function and it obviously then becomes necessary to declare the function inside the class.

```
 class Cube
 {
     public:
     int side;
     int getVolume();
 }

 // member function defined outside class definition
 int Cube :: getVolume()
 {
     return side*side*side;
 }

 int main()
 {
     Cube C1;
     C1.side = 4;    // setting side value
     cout<< "Volume of cube C1 = "<< C1.getVolume();
 }
```

Assignment: Explore each of the following member function by yourself:

1. Static functions
2. Const functions
3. Inline functions
4. Friend functions

Access Specifiers

Access Specifier in C++ defines how the members of the class can be accessed. C++ has 3 types of access specifiers:

1. Private Access Specifier
2. Public Access Specifier
3. Protected Access Specifier

A class can have three different sections: public, protected, or private and each section can have its own data members and member functions.

Syntax of Declaring Access Specifiers in C++

```
class className
{

private:
  // private data members and private member functions

public:
  // public data members and public member functions

protected:
  // protected data members and protected member functions

};
```

Note: If no access specifier is specified, then by default, all members and function of a class are private.

Example:

```
class A
{
private:
    int a;
public:
    int b;
    void show()
    {
        a=10;
        b=20;
        //All the data members of the class can be accessed here
        cout<<"\nAccessing variable within the class"<<endl;
        cout<<"Value of a: "<<a<<endl;
        cout<<"Value of b: "<<b<<endl;
    }
protected:
      int c;
};

int main()
{
A obj; // create object
obj.show();
scout<<"\nAccessing variable outside the class"<<endl;
//'a' cannot be accessed as it is private
//cout<<"value of a: "<<obj.a<<endl;
```

```
      //'b' is public as can be accessed from any where
      cout<<"value of b: "<<obj.b<<endl;
      return 0;
      }
```

Concept of Inheritance

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

Syntax for Inheritance:

```
class derived-class: access-specifier/visibility modes base-class
{....}
```

Where access-specifier is one of **public, protected,** or **private**, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

Consider a base class **Animal** and its derived class **Dog** as follows –

```
// base class
class Animal
{
   public:
    void eat() {
        cout << "I can eat!" << endl;
    }

    void sleep() {
        cout << "I can sleep!" << endl;
    }
};

// derived class
class Dog : public Animal {    //inheritance

   public:
    void bark() {
        cout << "I can bark! Woof woof!!" << endl;
    }
};

int main() {
    // Create object of the Dog class
    Dog dog1;

    // Calling members of the base class
    dog1.eat();
    dog1.sleep();

    // Calling member of the derived class
    dog1.bark();

    return 0;
}
```

O/P:

```
I can eat!
I can sleep!
I can bark! Woof woof!!
```

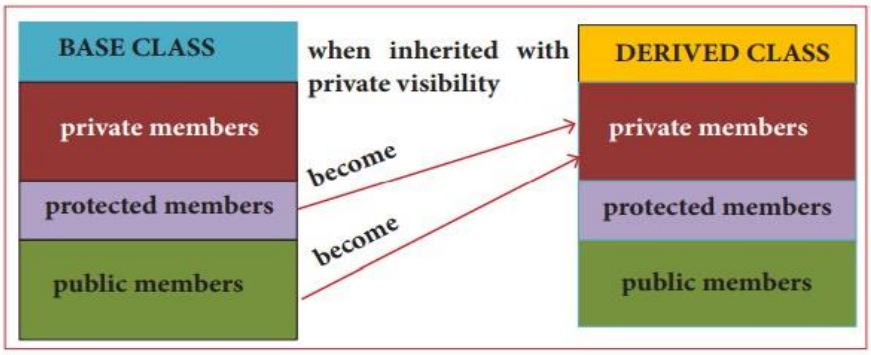
Visibility Modes

An important feature of Inheritance is to know which member of the base class will be acquired by the derived class. This is done by using visibility modes.

The accessibility of base class by the derived class is controlled by visibility modes. The three visibility modes are private, protected and public. The default visibility mode is private. Though visibility modes and access specifiers look similar, the main difference between them is Access specifiers control the accessibility of the members with in the class where as visibility modes control the access of inherited members with in the class.

Private visibility mode

When a base class is inherited with private visibility mode the public and protected members of the base class become 'private' members of the derived class



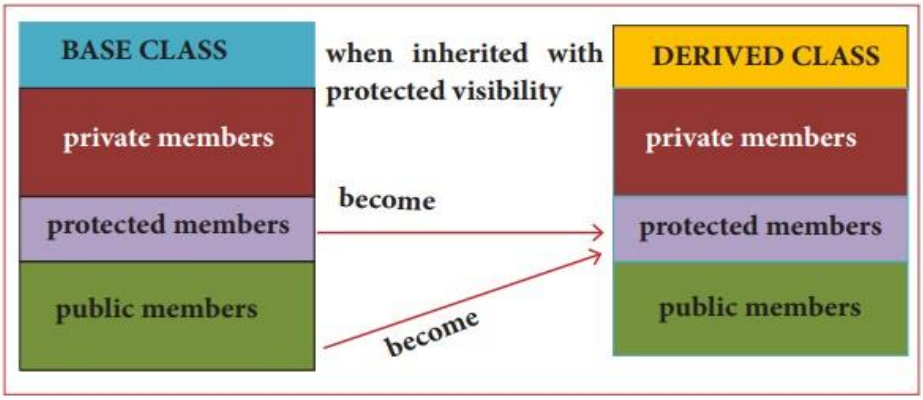
Example:

```
// private access specifier.cpp
#include <iostream>
using namespace std;

class base
{
        private:
         int x;

        protected:
            int y;

        public:
            int z;

        base() //constructor to initialize data members
        {
           x = 1;
           y = 2;
           z = 3;
        }
};
```

```
class derive: private base
{
        //y and z becomes private members of class derive and x remains
private
        public:
            void showdata()
            {
               cout << "x is not accessible" << endl;
                cout << "value of y is " << y << endl;
                cout << "value of z is " << z << endl;
            }
};
int main()
{
     derive a; //object of derived class
     a.showdata();
     //cout<<a.x;   not valid : private member can't be accessed outside of
class
     //cout<<a.y;   not valid : y is now private member of derived class
     //cout<<a.z;   not valid : z is also now a private member of derived
class
     return 0;
}       //end of program
```

Protected visibility mode

When a base class is inherited with protected visibility mode the protected and public members of the base class become 'protected members' of the derived class



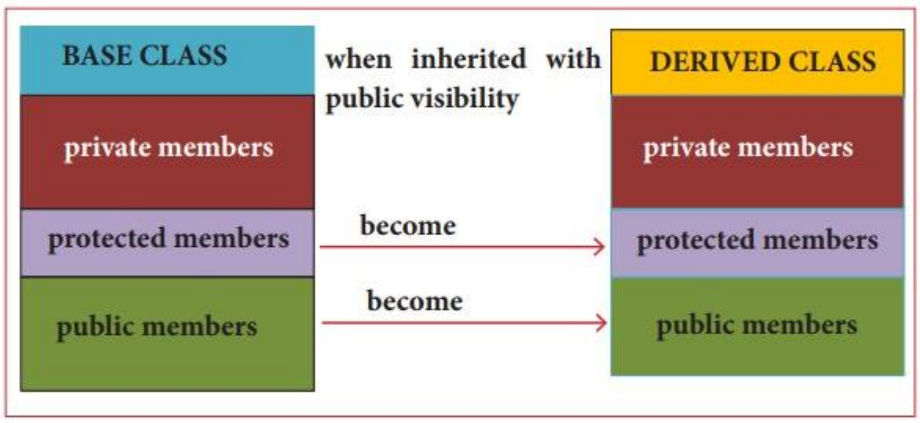
Example:

```
// protected access specifier.cpp
#include <iostream>
using namespace std;

class base
{
        private:
         int x;

        protected:
            int y;

        public:
```

```
            int z;

        base() //constructor to initialize data members
        {
           x = 1;
           y = 2;
           z = 3;
        }
};

class derive: protected base
{
        //y and z becomes protected members of class derive
        public:
            void showdata()
            {
               cout << "x is not accessible" << endl;
                cout << "value of y is " << y << endl;
                cout << "value of z is " << z << endl;
            }
                void add(int k)
                {
                        y=k;
                        cout<<y+5;
                }
};
int main()
{
     derive a; //object of derived class
     a.showdata();
        a.add(5);
     // cout<<a.x;    not valid : private member can't be accessed outside
of class
     // cout<<a.y;   not valid : y is now private member of derived class
     // cout<<a.z;    not valid : z is also now a private member of derived
class
     return 0;
}
```

Public visibility mode

When a base class is inherited with public visibility mode, the protected members of the base class will be inherited as protected members of the derived class and the public members of the base class will be inherited as public members of the derived class.



Example:

```
// public access specifier.cpp
#include <iostream>
using namespace std;

class base
{
      private:
        int x;

      protected:
          int y;

      public:
          int z;

      base() //constructor to initialize data members
      {
         x = 1;
         y = 2;
         z = 3;
      }
};

class derive: public base
{
      //y becomes protected and z becomes public members of class derive
      public:
          void showdata()
          {
             cout << "x is not accessible" << endl;
               cout << "value of y is " << y << endl;
               cout << "value of z is " << z << endl;
          }
};
int main()
{
     derive a; //object of derived class
     a.showdata();
     //cout<<a.x;   //not valid : private member can't be accessed outside
of class
     //cout<<a.y;   //not valid : y is now private member of derived class
     cout<<a.z;   //valid : z is also now a public member of derived class
     return 0;
}
```

this Keyword

In case of C++, “this” keyword is basically a pointer that points to the current instance (object) of the class. Hence, this is known as **this Pointer in C++**.

There are following uses of this keyword in the C++:

- Access the **currently executing object** of a class.
- Access the **data members** of the *currently executing object*.
- Calling the **member functions** associated with the *currently executing object*.
- To *resolve* the **shadowing issue**, when a *local variable has a same name as an instance variable*.

Access the ***currently executing object*** of a class:

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

class A
{
public:
void message()
{
cout<< "Hello from A" << "\n";
cout<< this;
}
};

int main()
{
A ob;
ob.message();
}
```

O/P:

```
Hello from A
0x7ffc10d3552f
```

Access the **data members** of the *currently executing object*:

```
#include<iostream>

using namespace std;

class A
{
private:
int a=10;

public:

void message()
{
cout<< "Hello from A" << "\n";
cout<< this->a;
}
};
int main()
{
A ob;
ob.message();
}
```

O/P:

```
Hello from A
10
```

Calling the **member functions** associated with the *currently executing object*:

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

```
class A
{
private:
int a=10;

public:
void message();
void hello();
};

//Definition of message() function of A class
void A :: message()
{
cout<< "Hello from A" << "\n";
this->hello();
}

//Definition of hello() function of A class
void A :: hello()
{
cout<< "Bonjour" << "\n";
cout<< "Hello" << "\n";
cout<< "Namaste" << "\n";
}

int main()
{
A ob;
ob.message();
}
```

O/P:

```
Hello from A
Bonjour
Hello
Namaste
```

To *resolve* the **shadowing issue**, when a *local variable has a same name as an instance variable*:

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

class Test
{
private:
int x;                  //data member : x
public:
void setX (int x) //local variable : x
{
      this->x = x;
}

/*Test(int x)
{
      this->x = x;
}*/
```

```
void print()
{
  cout << "x = " << x << endl;
}
};

int main()
{
Test obj;
obj.setX(20);
obj.print();
return 0;
}
```

Constructors

- Constructors are special functions which performs initialization of every object when being used in classes.
- The Compiler calls the Constructor whenever an object is created.
- Constructors initialize values to object members after storage is allocated to the object.
- Whereas, Destructor on the other hand is used to destroy the class object.
- While defining a contructor you must remember that the **name of constructor** will be same as the **name of the class**, and contructors will never have a return type.
- Constructors can be defined either inside the class definition or outside class definition using class name and scope resolution :: operator.

Syntax:

```
class A
{
    public:
    int x;
    // constructor
    A()
    {
        x=10;// object initialization
    }
};
```

Or

```
class A
{
    public:
    int i;
    A(); // constructor declared
};

// constructor definition
A::A()
{
    i = 1;
}
```

Types of Constructors in C++

Constructors are of three types:

1. Default Constructor
2. Parametrized Constructor
3. Copy Constructor

Default Constructors: Default constructor is the constructor which doesn't take any argument. It has no parameter.

Example:

```
class Cube
{
    public:
    int side;
    Cube()
    {
        side = 10;
    }
};

int main()
{
    Cube c;
    cout << c.side;
}
```

In this case, as soon as the object is created the constructor is called which initializes its data members.

A default constructor is so important for initialization of object members, that even if we do not define a constructor explicitly, the compiler will provide a default constructor implicitly.

```
class Cube
{
    public:
    int side=10;
};

int main()
{
    Cube c;
    cout << c.side;
}
```

Parameterized Constructors: These are the constructors with parameter. Using this Constructor you can provide different values to data members of different objects, by passing the appropriate values as argument.

```
class Student
{
    public:
    int rollno;
    string name;

    // Parameterized constructor
    Student(int x, string str)
    {
```

```
        rollno = x;
        name = str;
    }
};

int main()
{
    // student A initialized with roll no 11 and name John
    Student A(11, "John");
}
```

Constructor Overloading: Using multiple constructors in the same class.

```
class Student
{
    public:
    int rollno;
    string name;
    // first constructor
    Student(int x)
    {
        rollno = x;
        name = "None";
    }
    // second constructor
    Student(int x, string str)
    {
        rollno = x;
        name = str;
    }
};


int main()
{
    // student A initialized with roll no 10 and name None
    Student A(10);

    // student B initialized with roll no 11 and name John
    Student B(11, "John");
}
```

Destructor

A destructor is a special member function that works just opposite to constructor, unlike constructors that are used for initializing an object, destructors destroy (or delete) the object.

Syntax of Destructor

```
~class_name()
{
   //Some code
}
```

Similar to constructor, the destructor name should exactly match with the class name. A destructor declaration should always begin with the tilde(~) symbol as shown in the syntax above.

Destructor rules

1) Name should begin with tilde sign(~) and must match class name.
2) There cannot be more than one destructor in a class.
3) Unlike constructors that can have parameters, destructors do not allow any parameter.
4) They do not have any return type, just like constructors.
5) When you do not specify any destructor in a class, compiler generates a default destructor and inserts it into your code.

Example:

```
#include <iostream>
using namespace std;
class HelloWorld
{
  public:
  //Constructor
  HelloWorld()
  {
    cout<<"Constructor is called"<<endl;
  }
  //Destructor
  ~HelloWorld()
  {
    cout<<"Destructor is called"<<endl;
   }
   //Member function
   void display()
  {
     cout<<"Hello World!"<<endl;
   }
};

int main()
{
   //Object created
   HelloWorld obj;
   //Member function called
   obj.display();
   return 0;
}
```

O/P:

```
Constructor is called
Hello World!
Destructor is called
```

Friend class

Friend Class:

A **friend class** is a class that can access the private and protected members of a class in which it is declared as **friend**. This is needed when we want to allow a particular class to access the private and protected members of a class.

Function Class Example: Class ABC can access the private (and protected) members of XYZ. In this example we are ***passing object as an argument to the function*.**

```
#include <iostream>
using namespace std;
class XYZ
{
private:
   char ch='A';
   int num = 11;
public:

    /* This statement would make class ABC
    * a friend class of XYZ, this means that
    * ABC can access the private and protected
    * members of XYZ class. */

    friend class ABC;
};
class ABC
{
public:
   void disp(XYZ ob)
  {
      cout<<ob.ch<<endl;
      cout<<ob.num<<endl;
   }
};

int main()
{
   ABC obj1;
   XYZ obj2;
   Obj1.disp(obj2);
   return 0;
}
```

O/P:

```
A
11
```

Friend Function:

Similar to friend class, this function can access the private and protected members of another class. A global function can also be declared as friend as shown in the example below:

Friend Function Example

```
#include <iostream>
using namespace std;
class XYZ {
private:
   int num=100;
```

```
   char ch='Z';
public:
   friend void disp(XYZ obj);
};
//Global Function
void disp(XYZ ob)
{
   cout<<ob.num<<endl;
   cout<<ob.ch<<endl;
}
int main() {
   XYZ obj;
   disp(obj);
   return 0;
}
```

O/P:

```
100
Z
```

Class member as a friend

We can also make a function of one class as a friend of another class. We do this in the same way as we make a function as a friend of a class. The only difference is that we need to write **class_name ::** in the declaration before the name of that function in the class whose friend it is being declared. The friend function is only specified in the class and its entire body is declared outside the class. It will be clear from the example given below.

Syntax:

```
class A; // forward declaration of A needed by B

class B
{
    display( A a ); //only specified. Body is not declared
};

class A
{
    friend void B::display( A );
};

void B::display(A a) //declaration here
{
}
```

Example:

```
#include <iostream>
using namespace std;
class A; // forward declaration of A needed by B

class B
```

```
{
    public:
        void display(A obj); //no body defined, only declaration
};

class A
{
    int x;
    public:
        A()
        {
            x = 4;
        }
        friend void B::display(A);
};

void B::display(A obj)
{
    cout << obj.x << endl;
}

int main()
{
    A a;
    B b;
    b.display(a);
    return 0;
}
```

O/P:

```
4
```

Error handling (exception)

An **exception** is a problem that arises during the execution of a program, such as an attempt to divide by zero, trying to open a non existing file, etc. In these cases, programs compile perfectly but give an error during runtime and stop the code (even the correct code) from execution.

Exception handling can provide a way to transfer the control when from the part where the exception has occurred to the part where it can be handled, so that program can run without any error. C++ exception handling is built upon three keywords: **try, catch,** and **throw**.

- **try** − A **try** block allows you to define a block of code to be tested for errors while it is being executed. If at all this code encounters an error, a particular exceptions will be activated. It's followed by one or more catch blocks.
- **catch** – A catch block allows you to define a block of code to be executed, if an error occurs in the try block. The **catch** keyword indicates the catching of an exception.
- **throw** – A throw block throws an exception when a problem is detected, which lets us create a custom error.

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 as follows −

```
try
{
   // protected code
```

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

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

Throwing Exceptions

Exceptions can be thrown anywhere within a code block using **throw** statement. The operand of the throw statement 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.

Exception handling Example: Divide by zero

```
#include <iostream>
using namespace std;
int main()
{
	int a, b;
	cout<<"Enter two integer values: ";
	cin>>a>>b;
	try
	{
		if(b == 0)
		{
			throw b;
		}
		else
		{
			cout<<(a/b);
		}
	}
	catch(int ex)
	{
		cout<<"Exception: Denominator can not be zero : "<<ex;
	}
	return 0;
}
```

O/P:

```
Enter two integer values: 10

0

Exception: Denominator can not be zero : 0
```

Exercise 1: WAP that accepts a five digit palindrome number. If the number is not a palindrome, the program should be able to raise an exception that “the number has to be a palindrome”

Exercise2: WAP that accepts a three digit number and a character value from the user. If the number is between (150-180) or else if the characters entered are either a or k, the program should be able to raise exceptions for both the cases.

Exercise3: Explore various standard exceptions in C++.