# Unit-5

# Templates, Exceptions

# Topics to cover:

- **Templates**: Introduction to templates, function templates and class templates.

# Templates

- Template supports generic programming.
- It allows developing reusable software components such as functions, classes, etc supporting different data types in a single frame work.
- A template in c++ allows the construction of a family of template functions and classes to perform the same operation on different data types.
- They perform appropriate operations depending on the data type of the parameters passed to them.
- What we'd prefer to do is write "generic code"
  - Code that is type-independent
  - Code that is compile-type polymorphic across types

# Templates

- C++ has the notion of templates
- A function or class that accepts a type as a parameter
  - specify (one or more) types or values as arguments to it

- At compile-time, the compiler will generate the “specialized” code from your template using the types you provided
  - Your template definition is NOT runnable code
  - Code is only generated if you use your template

# Templates in C++

```
template <typename T>
T min (T a, T b)
{
return a<b? a : b;
}
```

min(5,10)

```
int min (int a, int b)
{
return a<b? a : b;
}
```

Compiler generates function when integer arguments are passed

min(2.3,7.76)

```
float min (float a, float b)
{
return a<b? a : b;
}
```

Compiler generates function when float arguments are passed

# What is the difference between function overloading and templates?

- Both function overloading and templates are examples of polymorphism features of OOP.
- Function overloading is used when multiple functions do quite similar (not identical) operations,
- Templates are used when multiple functions do identical operations.

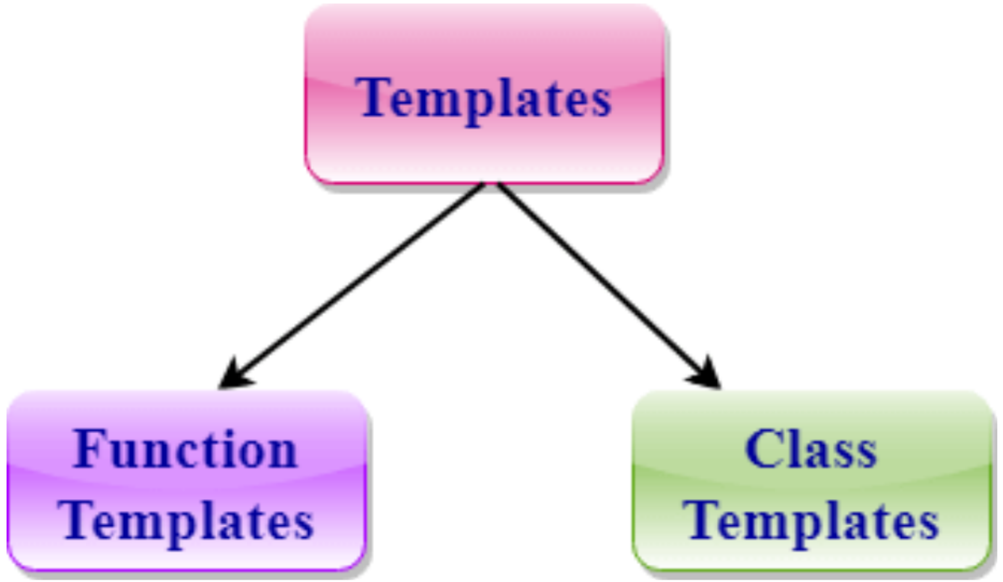
# Templates can be represented in two ways

## Function Templates:

- We can define a template for a function.
- For example, if we have an add() function, we can create versions of the add function for adding the int, float or double type values.

## Class Template:

- We can define a template for a class.
- For example, a class template can be created for the array class that can accept the array of various types such as int array, float array or double array.



# The general format of class template is

- Template all us to define generic classes .
- It is a simple process to create a generic class using a template with a anonymous type.

```
Template< class T >
class classname
{
    //.....
    // class member specification
    // with anonymous type T
    // whenever appropriate
    //.....
};
```

The class template definition is very similar to an ordinary class definition except the prefix template< class t > and the use of type T.

This prefix tells the compiler that we are going to declare a template and use T as a type name in the declaration.

# class template example

```
#include <iostream>
using namespace std;
template<class T>
class A
{
    public:
    T num1;
    T num2;
    A(T n1, T n2)
    {
        num1=n1;
        num2=n2;
    }

    void add()
    {
        cout << "Addition of num1 and num2 :
" << num1+num2<<endl;
    }
};
```

```
int main()
{
    A<int> intadd(3,4);
    A<float> floatadd(2.2, 5.6);
    intadd.add();
    floatadd.add();
    return 0;
}
```

```
Addition of num1 and num2 : 7
Addition of num1 and num2 : 7.8
```

# C++ class templates with multiple parameters

- We can use more than one generic data type in a class template.
- They are declared as a comma separated list within the template specification

```
templates< class T1, class T2 >
class classname
{
    .....
    ......(body of the class)

};
```

```
#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;
        }
};
```

```
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;
}
```

```
1.23 and 123
100 and  W
```

# C++ Function Templates

- Are used to create a family of functions with different argument types.
- The general format of a function template is

```
template< class T >
returntype function_name(arguments of type T)
{
  //.........
  // body of function
  // with type T
  // whenever appropriate
  //.....
}
```

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

template <class T>
T max(T &a,T &b)
{
if (a>b)
    return a;
else
    return b;
}
int main()
{
char ch1,ch2;
cout<<"\nenter two characters:";
cin>>ch1>>ch2;
cout<<max(ch1,ch2);
```

```
int a,b;
cout<<"\nenter two integers:";
cin>>a>>b;
cout<<max(a,b);
float p,q;
cout<<"\nenter two float values:";
cin>>p>>q;
cout<<max(p,q);

return 0;
}
```

```
enter two characters:p a
p
enter two integers:3 10
10
enter two float values:4.6 4.1
4.6
```

# C++ Function Templates With Multiple Parameters

```
Template<class T1, class T2, ……>
return_type function_name(arguments of types T1, T2,….)
{
…….
…….(body of function)
…….
}
```

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

template<class X,class Y>
void fun(X a,Y b)
{
    cout << "Value of a is : " <<a<< endl;
    cout << "Value of b is : " <<b<< endl;
}

int main()
{
   fun(15,12.3);

   return 0;
}
```

```
Value of a is : 15
Value of b is : 12.3
```

# C++ Member Function Template

- A member function template is a template that defines a member function within a class template.
- It allows the function to be defined in such a way that it can accept one or more template parameters, in addition to the class template parameters.

```
Template< class T >
returntype classname< t >::functionname(arglist)
{
//.........
// function body
//......
}
```

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

template<class T> class MyClass {
public:
    // Template member function declaration
    template<class U> void display(U arg);
};

// Template member function definition
template<class T> template<class U>

void MyClass<T>::display(U arg) {
    cout << "Argument passed: " << arg << endl;
}
```

```cpp
int main() {
    MyClass<int> obj;
    obj.display(5);

    MyClass<double> obj2;
    obj2.display(3.14);

    return 0;
}
```

```
Argument passed: 5
Argument passed: 3.14
```

# Overloading of Function Template

- A template function may be overloaded either by template function or ordinary function template.

- In such types cases the overloading resolution is accomplished as follows

  1. call an general function that has an exact match.

  2. call a template function that could be create with an exactly match.

  3. Try the normal overloading resolution to ordinary functions and call the one that matches.

- An Error is generated when no match is found.

# Overloading of Function Template

- the overloaded template functions can differ in the parameter list.

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

template<class X>
void fun(X a)
{
    cout << "Value of a is : " <<a<< endl;
}

template<class X,class Y>
void fun(X b ,Y c)
{
   cout << "Value of b is : " <<b<< endl;
   cout << "Value of c is : " <<c<< endl;
}
int main()
{
   fun(10);
   fun(20,30.5);
   fun("Template is very easy!");
   return 0;
}
```

```
Value of a is : 10
Value of b is : 20
Value of c is : 30.5
Value of a is : Template is very easy!
```

# Restrictions of Generic Functions

- Generic functions perform the same operation for all the versions of a function except the data type differs.
- Let's see a simple example of an overloaded function which cannot be replaced by the generic function as both the functions have different functionalities.

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

void fun(double a)  {
    cout<<"value of a is : "<<a<<'\n';
}

void fun(int b)
{
    if(b%2==0)     {
        cout<<"Number is even";
    }
    else      {
        cout<<"Number is odd";
    }

}

int main()  {
   fun(4.6);
   fun(6);
   return 0;
}
```

# NON TYPE TEMPLATE ARGUMENTS

- Template can have multiple arguments.
- It's also possible to use non type arguments.
- That is in addition to the type argument T, we can also use other arguments or parameters such as strings, function names, constant expressions and built in types.

```
template< class T, int size >
class array
{
T a[size];
//.....
//.....
};
```

- This template supplies the size of the array as an argument in above program.
- This implies that the size of the array is known to the compiler at the compiler time itself.
- The arguments or paramters must be specified whenever a template class is created

```
array< int,10 > a1; // Array of 10 integers
array< float,5 > a2; // array of 5 floats
array< char,20 > a3; //string of size 20
```

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

template<class T, int size>
class A
{
    public:
    T arr[size];
    void insert()
    {
        int i =1;
        for (int j=0;j<size;j++)        {
            arr[j] = i;
            i++;
        }
    }

    void display()
    {
        for(int i=0;i<size;i++)        {
            std::cout << arr[i] << " ";
        }
    }
};
```

```
int main()
{
    A<int,10> t1;
    t1.insert();
    t1.display();
    return 0;
}
```

```
1 2 3 4 5 6 7 8 9 10
```

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

template <class T>
void print( T a)
{
    cout<<a<<endl;
}

template <class T>
void print( T a, int n)
{
    int i;
    for (i=0;i<n;i++)
        cout<<a<<endl;
}
```

```
int main()
{
print(1);
print(3.4);
print(455,3);
print("hello",3);
}
```

```
1
3.4
455
455
455
hello
hello
hello
```

# Why Templates?

- C++ supports a powerful feature known as a template to implement the concept of generic programming.
- A template allows us to create a family of classes or family of functions to handle different data types.
- Template classes and functions eliminate the code duplication of different data types and thus makes the development easier and faster.
- Multiple parameters can be used in both class and function template.
- Template functions can also be overloaded.
- We can also use nontype arguments such as built-in or derived data types as template arguments.


## Write a program :

To apply Bubble sort for array of integer  and array of float values using Templates

# Unit-5

# Exceptions

# Topics to cover:

**Introduction to exception:** try-catch throw, multiple catch, catch all, rethrowing exception, User defined exceptions.

# Exception Handling

- Exceptions are run time anomalies or unusual conditions that a program may encounter while executing.
- The unusual conditions could be faults, causing an error which in turn causes the program to fail.
- The error handling mechanism of c++ is generally referred to as exception handling.
- Anomalies may be, division by zero, access to an array outside its boundary, running out of memory disk space etc.
- Exceptions are new features added in ANSI C++.

# Exception Types

- Exceptions are classified into

1. Synchronous and
2. Asynchronous

# Synchronous exception

- The exceptions which occur during the program execution, due to some fault in the input data or technique that is not suitable to handle the current class of data with in a program is known as synchronous exception.
- Example:
- errors such as out of range, overflow, underflow and so on.

# Asynchronous exception

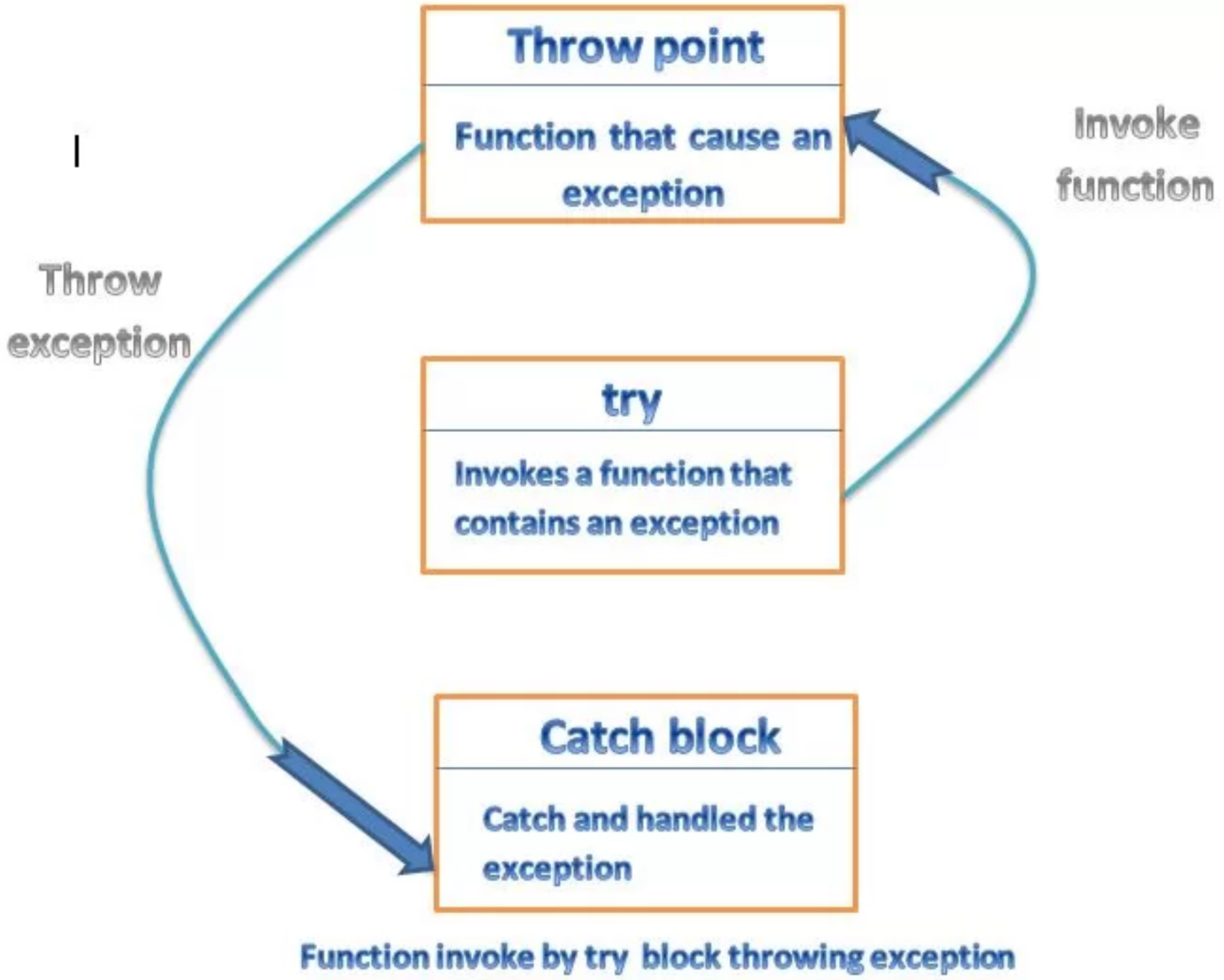
- The exceptions caused by events or faults unrelated to the program and beyond the control of program are asynchronous exceptions.
- For example, errors such as keyboard interrupts, hardware malfunctions, disk failure and so on.

# Exception Handling Mechanism

1. Find the problem (Hit the exception)
2. Inform that an error has occurred (Throw the exception)
3. Receive the error information(catch the exception)
4. Take corrective actions(Handled the exception)

- Throwing an unhandled exception causes the standard library function **terminate()** to be invoked.
- By default, terminate() calls **abort()** to stop the program.

# Exception Handling Model:



The General Format of code for this kind of relationship is shown below

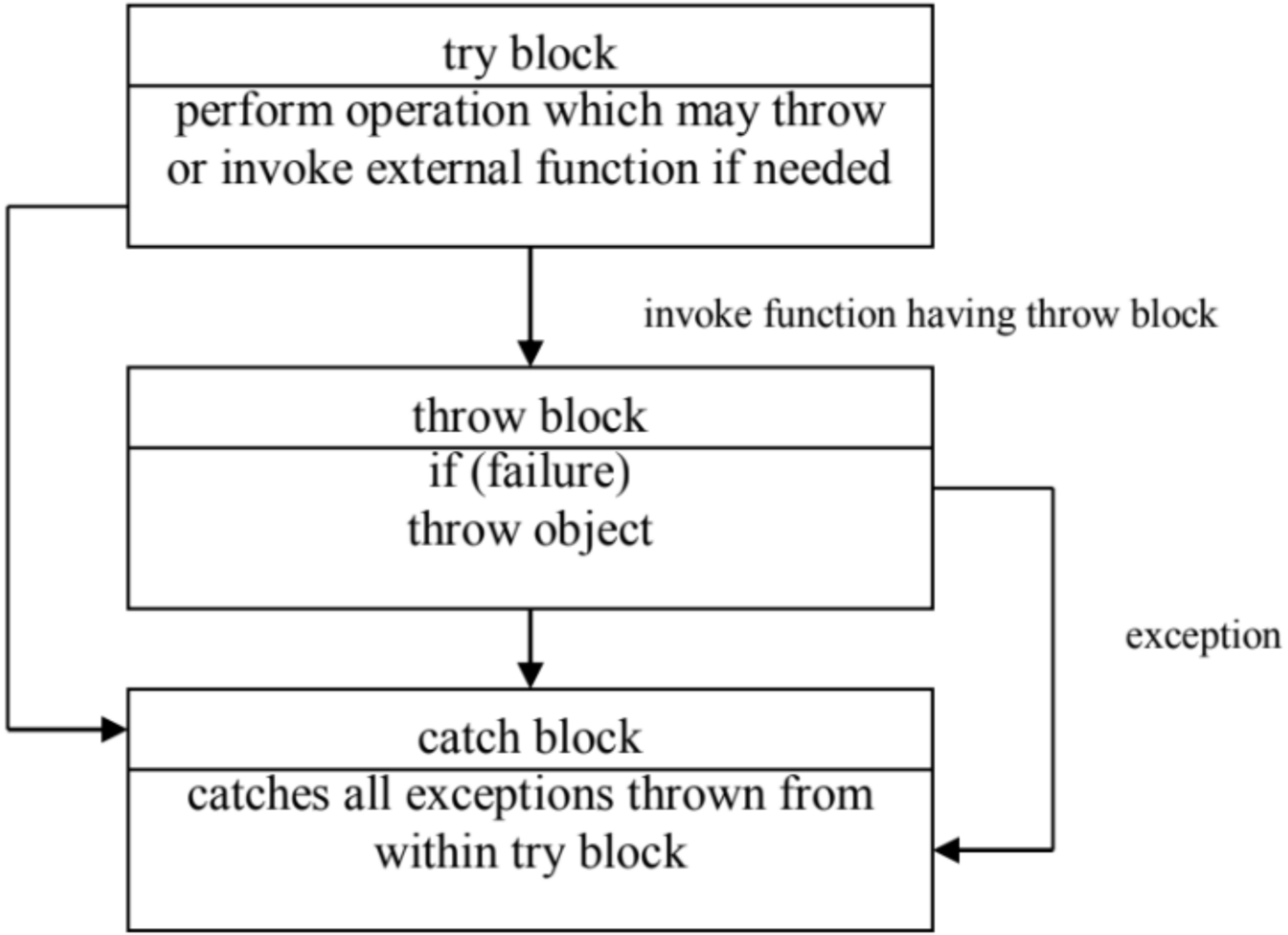
```
type function(arg list) // function with exception
{
.......
.......
throw (object); // throws exception
.......
.......
}

try
{

.....
..... invoke function here
.....
}

catch (type arg) // catches exception
{
.......
....... Handles exception here
.......
}
```

Note: The try block is immediately followed by the catch block, irrespective of the location of the throw point


try block
perform operation which may throw
or invoke external function if needed
invoke function having throw block
throw block
if (failure)
throw object
exception
catch block
catches all exceptions thrown from
within try block

# Exception Handling Model:

## throw construct:

- The keyword throw is used to raise an exception when an error is generated in the computation.
- The throw expression initialize a temporary object of the typeT used in thorw (T arg).

syntax: throw T;

## catch construct:

- The exception handler is indicated by the catch keyword. It must be used immediately after the statements marked by the try keyword.
- The catch handler can also occur immediately after another catch
- Each handler will only evaluate an exception that matches.

```
syntax :  catch(T)
{
    // error meassges
}
```

# try construct:

- The try keyword defines a boundary within which an exception can occur.
- A block of code in which an exception can occur must be prefixed by the keyword try.
- Following the try keyword is a block of code enclosed by braces.
- If an exception occurs, the program flow is interrupted.

```
try
{
    ...
if (failure)
    throw T;
}
catch(T)
{
    ...
}
```

# Throwing and Catch Exception

- When an exception that is desired to be handled is detected , it is thrown using the throw statement(throw keyword) in one of the following forms show below

```
throw(exception);
throw exception;
throw; // this statement for re-throwing an exception
```

- The operand object exception may be of any type, including constant, it is also possible to throw objects not intended for error handling
- Whenever an exception is thrown, it will be caught by the catch statement(block) associated with the try block.
- ie., the control exist the current try block, and is transferred to the catch block after that try block

# Catch blocks mechanism

- As stated earlier, code for handling exception is included in catch blocks.
- A catch block looks like any function definition who catch the exception and is of the form

```
catch(type arg)
{
// statement for
// managing exception
}
```

- The type indicates the type of the exception that catch block handles.
- The argument arg is an optional parameter name.

# Exception Handling example

```
#include<iostream>
using namespace std;
int main()
{
int a,b;
cout<<"enter two numbers:";
cin>>a>>b;
try
{
    if (b==0)
        throw b;
    else
        cout<<a/b;
}
catch(int x)
{
    cout<<"2nd operand can't be 0";
}
}
```

```
enter two numbers:4 0
2nd operand can't be 0
```

```
#include<iostream>
using namespace std;
int main()
{
    int a,b;
    cout << "Enter values of a and b \n";
    cin >> a;
    cin >> b;

    int x=a-b;
    try
      {
         if(x!=0)
         {
            cout << "Result(a/x)=" << a/x << "\n";
         }
         else
         {
            throw(x); // throws int object
         }
      }
```

```
 catch(int i)
    {
       cout << "EXCEPTION caught:=" << x << "\n";
    }
cout<<"END";
}
```

```
Enter values of a and b
3 3
EXCEPTION caught:=0
END
```

```
// Throw point outside the try block

#include <iostream>
using namespace std;

void divide(int x, int y, int z)
{
    cout <<"\n we are inside the function \n";
    if((x-y)!=0)          // it is ok
    {
        int r = z/(x-y);
        cout << "Result= " << r <<"\n";
    }

    else              // There is a problem
    {
        throw(x-y); // throw point
    }
}
```

```
 int main()
{
    try
    {
        cout <<"we are inside the try block \n";
        divide(10,20,30); // invoke divide()
        divide(10,10,20); // invoke divide()
    }
    catch(int i)
    {
        cout << "caught The exception \n";
    }
}
```

```
we are inside the try block

 we are inside the function
Result= -3

 we are inside the function
caught The exception
```

A **try** block can be localized to a function.
Each time the function is entered, the exception handling relative to that function is reset.

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

//Localize a try/catch to a function

void Xhandler(int test)
{
    try
    {
    if(test) throw test;
    }
    catch(int i)
    {
    cout<<"Caught Exception #:  "<<i<<"\n";
    }

}
```

```
 int main()
{
    cout << "Start \n";
    Xhandler(1);
    Xhandler(2);
    Xhandler(0);
    Xhandler(3);
    cout<<"End\n";
}
```

```
Start
Caught Exception #: 1
Caught Exception #: 2
Caught Exception #: 3
End
```

# Catching Class Types

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

class MyException {
public:
  char message[80];
  int what;

  MyException() { *message = 0; what = 0; }

  MyException(char *s, int e) {
    strcpy(message, s);
    what = e;
  }
};
```

```
int main()
{
  int i;

  try {
    cout << "Enter a positive number: ";
    cin>>i;
    if(i<0)
        throw MyException("Not Positive", -1);
  }
  catch (MyException e) { // catch an error
    cout << e.message << ": ";
    cout << e.what << "\n";
  }

  return 0;
}
```

```
Enter a positive number: -4
Not Positive : -4
```

# Handling Derived-Class Exceptions

- **catch** class for a base class will also match any class derived from that base.
- To catch exceptions of both a base class type and a derived class type, put the derived class first in the **catch** sequence.
- If not, the base class **catch** will also catch all derived classes.

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

class D: public B{
};

int main() {
  D derived;

  try{
     throw derived;
  }

  catch(B b){
     cout<< "caught a base class\n";
  }

  catch(D d){
      cout<< "This wont execute\n";
  }
  return 0;
}
```

As derived is an object that has B as a base class, it will be caught by 1st catch and 2nd catch will never execute

To fix, reverse the order of the catch clauses

# Multiple catch statements

```
try
{
    // try block
}
catch(type1 arg)
{
    // catch block1
}
catch(type2 arg)
{
    // catch block2
}
....
....
catch(typeN arg)
{
    //catch block N
}
```

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

int main() {

    double numerator, denominator,
           arr[4] = {0.0, 0.0, 0.0, 0.0};
    int index;

    cout << "Enter array index: ";
    cin >> index;

    try {

        // throw exception if array out of bounds
        if (index >= 4)
            throw "Error: Array out of bounds!";

        // not executed if array is out of bounds
        cout << "Enter numerator: ";
        cin >> numerator;
        cout << "Enter denominator: ";
        cin >> denominator;

        // throw exception if denominator is 0
        if (denominator == 0)
            throw 0;

        // not executed if denominator is 0
        arr[index] = numerator / denominator;
        cout << arr[index] << endl;
    }
```

Contd.,

```
    // catch "Array out of bounds" exception
    catch (const char* msg) {
        cout << msg << endl;
    }

    // catch "Divide by 0" exception
    catch (int num) {
        cout << "Error: Cannot divide by " << num << endl;
    }

    // catch any other exception
    catch (...) {
        cout << "Unexpected exception!" << endl;
    }

    return 0;
}
```


```
Run1:
Enter array index: 5
Error: Array out of bounds!

Run2:
Enter array index: 2
Enter numerator: 4
Enter denominator: 0
Error: Cannot divide by 0
```

# Exception Handling Options

- Catching All Exceptions
- Restricting Exceptions
- Re-throwing An Exception

# Catch All Exceptions

- We can force the catch statement to catch all the exceptions instead of a certain type alone.
- This could be achieved by defining the catch statement using ellipses as follows

```
catch(...)
{
// statements for processing
// all exceptions
}
```

Note: Remember catch(…) should always be placed last in the list of handlers.

placing it before other catch blocks would prevent those blocks from catching exception

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

void test(int x)
{
    try
    {
        if (x==0) throw x;
        if (x==-1) throw 'x';
        if (x==1) throw 1.0;
    }
    catch (...)
    {
        cout << "Caught an exception \n";
    }
}

int main()
{
    cout << "Testing generic catch \n";
    test(-1);
    test(0);
    test(1);
}
```

```
Testing generic catch
Caught an exception
Caught an exception
Caught an exception
```

```
#include <iostream>
using namespace std;
int main()
{
   try {
  int age = 15;
  if (age >= 18) {
    cout << "Access granted - you are old enough.";
  } else {
    throw 505;
  }
}
catch (...) {
  cout << "Access denied - You must be at least 18 years old.\n";
}
   return 0;
}
```

# Re-throwing An Exception

```
#include <iostream>
using namespace std;
void MyHandler()
{
   try
   {
      throw "hello\n";
   }
   catch (const char*)
   {
   cout <<"Caught exception inside MyHandler\n";
   throw; //rethrow char* out of function
   }
}
```

```
int main() {
   cout<< "Main start\n";
   try   {
      MyHandler();
   }
   catch(const char*)   {
      cout <<"Caught exception inside Main\n";
   }
      cout << "Main end\n";
      return 0;
}
```

```
Main start
Caught exception inside
MyHandler
Caught exception inside Main
Main end
```

# Re-throwing An exception

- A handler may decide to re-throw the exception caught without processing it.

- In such situations, we may simply invoke throw without any arguments as,

  throw;

  - This cause the current exception to be thrown to the next enclosing try/catch sequence and is caught by a catch statement or block listed after that enclosing try block.
  - When an exception is re-thrown, it will be caught by the same catch statement or nay other catch in that group Rather it will be caught by an appropriate catch in the outer try catch sequence only

  - A catch handler itself may detect and throw an exception. Here again the exception thrown will not be caught by any catch statements in that group. It will be passsed on to the next outer try.catch sequence for processing

```
#include <iostream>
using namespace std;
void divide(double x, double y)
{
    cout << "Inside function \n";
    try
    {
        if (y==0.0)
            throw y; // throwing double
        else
            cout << "Division=" << x/y << "\n";
    }
    catch (double)  // catch a double
    {
        cout << "caught double inside function \n";
        throw ; // re-throwing double
    }
    cout << "End of function \n \n";
}
```

```
int main() {
    cout << "Inside main \n";
    try    {
        divide(20.0,0.0);
    }
    catch (double)    {
        cout << "Caught double inside main \n";
    }
    cout << "End of main \n";
    return 0;
}
```

```
Inside main
Inside function
caught double inside function
Caught double inside main
End of main
```

# Restricting Exceptions/ Specifying Exceptions

- It is possible to restrict a function to throw only certain specified exceptions.
- This is achieve by adding a keywords throw *list* clause to the function definition.

```
type function(arg-list) throw (type-list)
{
      ..........
      ..........
      ..........
function body
}
```

- The type list specifies the type of exceptions that may be thrown.
- if Throwing any other type of exceptions in program it will cause abnormal program termination.
- if we wish to prevent a function from throwing any exception, we may do by making the type-list empty inside arguments. That is we must use

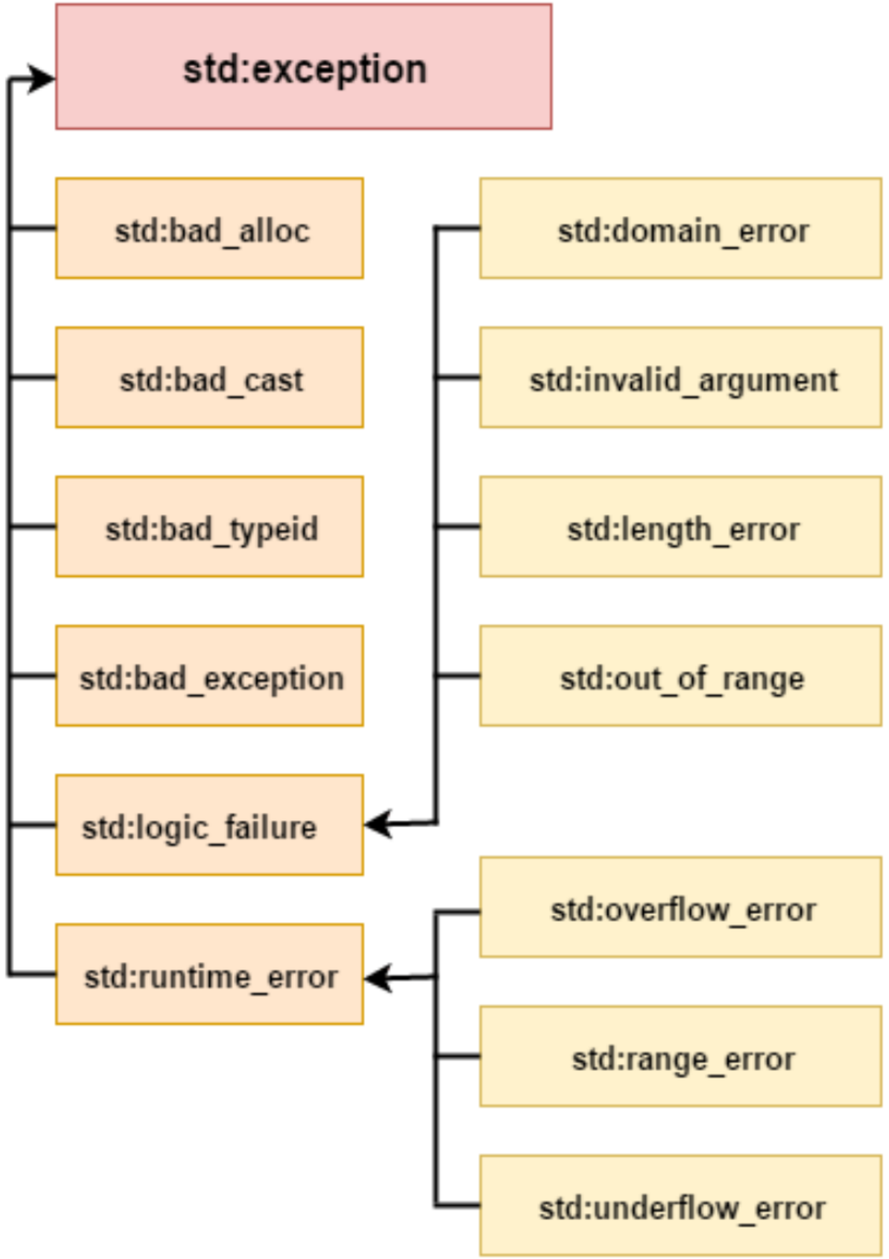
```
throw(); // Empty list
```

```
#include <iostream>
using namespace std;
void test(int x) throw(int,double,char)
{
    if(x==0) throw 'x'; // char
    else
    if(x==1) throw x; // int
    else
    if(x==-1) throw 1.0; // double
    cout<<"End of the function block \n";
}
int main()
{
    try
    {
        cout << "Testing throw restrictions \n";
        cout << “x==0 \n";
        test(0);
        cout << "x==1 \n";
        test(1);
        cout << "x==-1 \n";
        test(-1);
        cout << "x==2 \n";
        test(2);
    }
catch(char c)
    {
    cout << "caught a character \n";
    }
    catch(int m)
    {
    cout << "caught an integer \n";
    }
    catch(double d)
    {
    cout << "caught a double \n";
    }
    cout << "End of try catch system \n\n";
    return 0;
}
```

```
Testing throw restrictions
x==0
caught a character
End of try catch system
```

Write a program :

# C++ Standard Exceptions:




You can define your own exceptions by inheriting and overriding exception class functionality.

| 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. |

# Define New Exceptions:

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

class MyException : public exception {
    public:
   const char * what () {
     return "This is how MyException is handling
exception\n";
   }
};
```

the what() function is defined in the std::exception base class. It is a virtual member function declared in the std::exception class.

The what() function is typically overridden in derived exception classes to provide specific error messages for different types of exceptions.

```
int main() {
   try {
      throw MyException();
   }
   catch(MyException e) {
      cout << "MyException caught" <<endl;
      cout << e.what() <<endl;
   }

   catch(exception e) {
      cout << "library exception caught" <<endl;
   }

   return 0;
}
```

```
MyException caught
This is how MyException is handling exception
```

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

class MyException : public exception {
    public:
   const char * what () {
     return "divide error\n";
  }
};
```

Catches user defined exception

```
int main () {
int x, y;
cout << "Enter x: ";   cin >> x;
cout << "Enter y: ";   cin >> y;
try
{
   if(y==0)
   {
      MyException err;
      throw err;
   }
   else
      cout << x/y <<endl;
}
catch(MyException& e)  {
   cout << e.what();
}
catch(exception& e)  {
   cout << e.what();
}
return 0;
}
```

```
Enter x: 2
Enter y: 0
divide error
```

If not user defined catch

```
Enter x: 5
Enter y: 0
std::exception
```