OOP

EXCEPTION HANDLING

CHAPTER 10

KABIN DEVKOTA

UDITA BISTA

TABLE OF CONTENTS

.....

Our content is
divided into the given topics.
Each part will be described in the slides
that follow

ERROR HANDLING

The two most common types of bugs:

1. **Logic error:** Due to poor understanding of the problem and solution procedure.
2. **Syntactic error:** Due to poor understanding of language itself.

We often come across some peculiar problems other than logic and syntax errors. They are known as exceptions.
Exceptions are run time anomalies or unusual conditions that a program may encounter while executing. E.g. Division by zero,access to an array outside its bound, running out of memory or disk space etc.
The purpose of the exception handling mechanism is to provide means to detect and report “exceptional circumstances” so that appropriate action can be taken.

EXCEPTION HANDLING CONSTRUCTS (TRY, CATCH, THROW)

Exceptions are the anomalous or unusual events that change the normal flow of the program. Exception handling is the mechanism by which we can identify and deal with such unusual conditions. This mechanism suggests a separate error handling code that performs the following tasks:

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 (Handle the exception).

The error handling code basically consists of two segments, one to detect errors and to throw the exception, and the other to catch the exception and to take corrective actions.

CONSTRUCTS OF EXCEPTION HANDLING

In C++, the following keywords are used for exception handling.

a) try

b) catch

c) throw

1. The keyword try is used to preface a block of statements (surrounded by braces) which may generate exceptions. This block of statements is known as try block. When an exception is detected, it is thrown using a throw statement in the try block.
2. A catch block defined by the keyword catch is used to catch the exception thrown by the throw statement in the try block, and handles it properly.

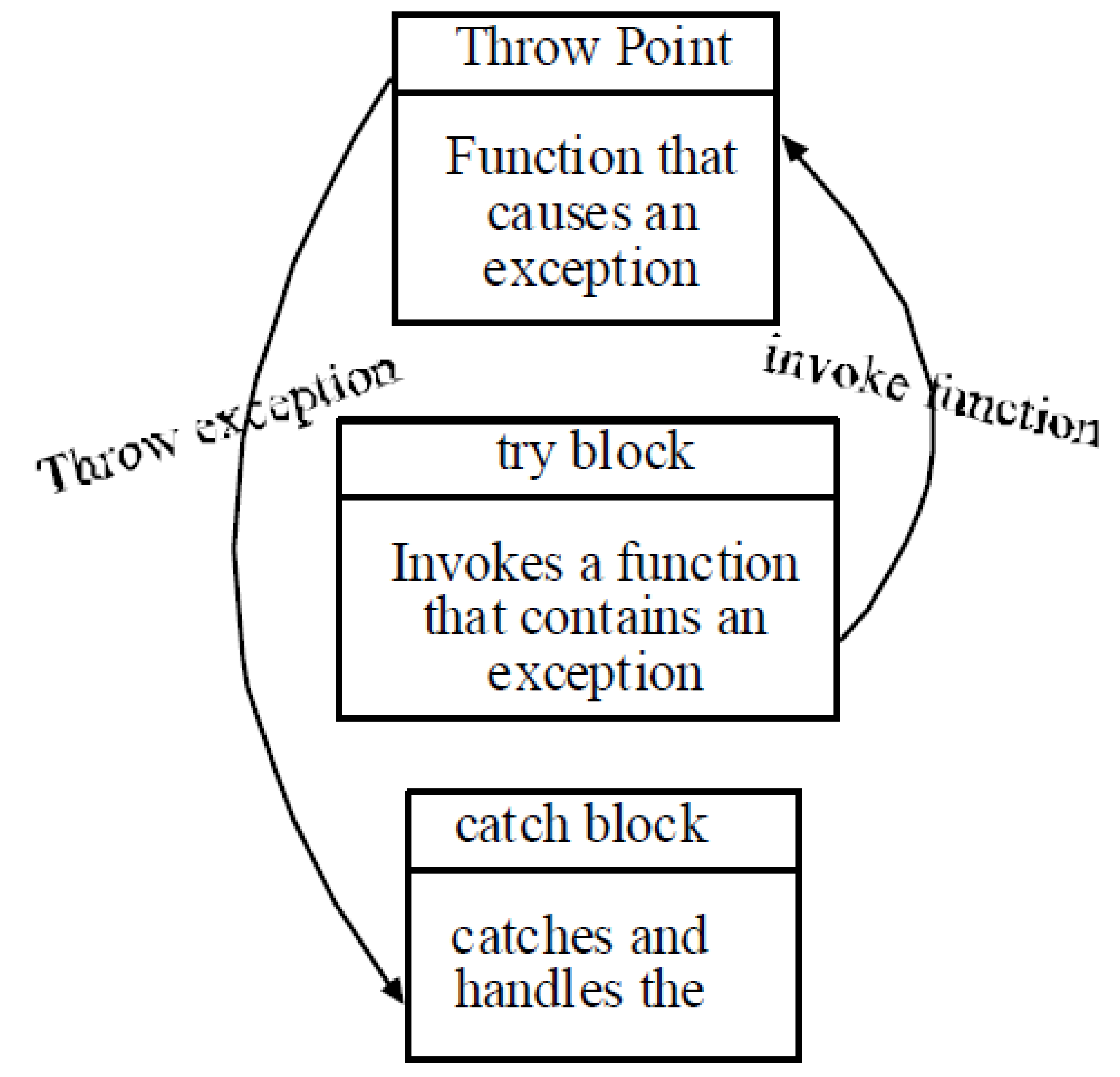

Fig : Function invoked by try block throwing exception

CONSTRUCTS OF EXCEPTION HANDLING

The catch block that catches the exception must immediately follow the try block that throws the exception. The general form of these two blocks are as follows:

```
...........
...........
try
{
 ..........   // block of statements which detects and
throw exception;  // throws an exception.
...........
...........
}
catch (type arg)
{
............   // block of statements that handles the exception
............   // thrown by the try block.
............
}
............
```

CONSTRUCTS OF EXCEPTION HANDLING

Try keyword is used to a block of statements surrounded by braces which may generate exceptions. When an exception is detected, it is thrown using the throw statement in try block, then the program control leaves the try block and enters the catch block comparing the catch argument try and the exception thrown type. If the type of the object matches the argument type in the catch statement, then the catch block is executed for handling the exception. If they don't match, the program is aborted with the help of the abort() function which is invoked by default. When no exception is detected and thrown, the control goes to the statement immediately after the catch block i.e. catch block is skipped. In handling exceptions we need to create a new kind of entity called an exception class.

CONSTRUCTS OF EXCEPTION HANDLING

```
#include<iostream>
using namespace std;
int main()
{
        int a,b,x;
        cout<<"Enter the values of a and b: ";
        cin>>a>>b;
        x = a-b;
        try
        {
                if(x!=0)
                cout<<"The result of (a/x) is "<<a/x<<endl;
                else
                throw(x);
        }
        catch(int i)
        {
                cout<<"Division by zero. X = "<<i<<endl;
        }
        return 0;
}
```

Output:

```
Enter the values of a and b: 4 2
The result of (a/x) is 2.
```

Output 2:

```
Enter the values of a and b: 8 8
Division by zero. X = 0
```

ADVANTAGE OVER CONVENTIONAL ERROR HANDLING

The error handling mechanism is somewhat new and very convenient in case of object oriented approach over conventional programming. When an error is detected, the error could be handled locally or not locally. Traditionally, when the error is not handled locally the function could

1. Terminate the program.
2. Return a value that indicates error.
3. Return some value and set the program in an illegal state.

The exception handling mechanism provides alternatives to these traditional techniques when they are dirty, insufficient and error prone. However, in the absence of exception all these three traditional techniques are used. Exception handling separates the error handling code from the other code making the program more readable. If the exception is not handled then the program terminates. Exception provides a way for code that detects a problem from which it cannot recover to the part of the code that might take necessary measures.

MULTIPLE EXCEPTION HANDLING

It is possible that a program segment has more than one condition to throw an exception. In such cases, we can associate more than one catch statement with a try statement(much like the conditions in a switch statement).

```
try
{
  //try block
}
catch(type1 arg)
{
 //catch block1
}
...........
...........
catch(typeN arg)
{
 //catch blockN
}
```

MULTIPLE EXCEPTION HANDLING

```
#include<iostream>
 using namespace std;
void test(int a, int b)
{
 try
 {
  if (b < 0)
throw b;
if(b == 0)
throw 1.1;
cout << "The Quotient = " << (a/b) << endl;
cout<<”End of try block”<<endl;
}
catch(int c)
{
 cout << "\nSecond Operand is less than zero" << endl;
 cout<<”caught an integer”<<endl;
 }
catch(double c)
 {
 cout<<"\nSecond operand is equal to zero"<<endl;
 cout<<”caught a double”<<endl;
 }
 cout<<”End of try-catch system”<<endl;
 }

int main()
 {
test(12,3);
test(12, -3);
 test(12,0);
return 0;
 }
```

Output:

```
The Quotient = 4
End of try block
End of try-catch system
Second Operand is less than zero
caught an integer
End of try-catch system
Second operand is equal to zero
caught a double
End of try-catch system
```

RE-THROWING EXCEPTION

We can make the handler(catch block) to rethrow the exception caught without processing it. In such situations, we may simply invoke “throw” without any argument.

throw;

This causes the current exception to be thrown to the next enclosing try/catch sequence and is caught by a catch statement of that try/catch block.

```
#include<iostream>
using namespace std;
void Calculate(double a,double b)
{
      cout<<”Inside the function”<<endl;
      try
            {
            if(b==0)
                   throw b;    //throwing double
            else
                   cout<<"\nThe result is:"<<a/b<<endl;
      }
      catch(double)    //catch a double
       {
            cout<<"\nCaught double value inside the catch block of the function";
            throw;     //rethrowing double
      }
}
```

```
int main()
 {
      cout<<”Inside the main block”<<endl;
      try
      {
            calculate(10.5,2.0);
            calculate(10.5,0.0);
       }
      catch(double)
      {
            cout<<"\nInside the Catch block if the main() function";
            cout<<"\nException due to 0"<<endl;
       }
      return 0;
      }
```

RE-THROWING EXCEPTION

Output:

```
Inside the main block
Inside the function
The result is 5.25

Inside the function
Caught double value inside the catch block of the function
Inside the Catch block if the main() function
Exception due to 0
```

CATCHING ALL EXCEPTION

In some situations, we may not be able to anticipate all possible types of exceptions and therefore may not be able to design independent catch handlers to catch them. In such circumstances, we can force a catch statement to catch all 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
}
```

CATCHING ALL 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(...)    //catch all
    {
        cout<<"caught an exception";
    }
}
```

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

Output:

```
Testing generic catch
caught an exception
caught an exception
caught an exception
```

NOTE:

It may be a good idea to use the catch(...) as a default statement along with other catch handlers so that it can catch all those exceptions which are not handled explicitly.

Remember, catch(..) should always be placed last in the list of catch handlers. Placing it before other catch blocks would prevent those blocks from catching exceptions.

EXCEPTION WITH ARGUMENTS

There may be situations where we need more information about the cause of exception. Let us consider there may be more than one function that throws the same exception in a program. It would be nice if we know which function threw the exception and the cause to throw an exception. This can be accomplished by defining the object in the exception handler. This means, adding data members to the exception class which can be retrieved by the exception handler to know the cause. While throwing, a throw statement throws an exception class object with some initial value, which is used by the exception handler.

EXCEPTION WITH ARGUMENTS

```
#include<iostream>
using namespace std;
class argument
 {
      int a, b;
      public:
            string msg;
            argument() {
                   a= b= 0;
            }
             argument(string name)
             {
                  msg = name;
            }
             void input()
             {
                  cout << "Enter the value of a and b : ";
                  cin >> a >> b;
            }
      void calculate()
      {
             if(b == 0)
             throw argument("Divided by Zero.");
             if(b < 0)
             throw argument("Divided by Negative number");
            cout << "Quotient = " << a/b << endl;
      }
};

int main()
 {
      argument A1;
      A1.input();
       try {
             A1.calculate();
       }
       catch(argument A)
       {
             cout << "Exception : " << A.msg << endl;
      }
       return 0;
 }
```

EXCEPTION SPECIFICATION FOR FUNCTION

It is possible to restrict a function to throw only certain specified exceptions. This is achieved by adding a throw list clause to the function definition.
The general form of using an exception specification is:

```
return_type function_name(arg_list) throw(type list)
{
 //body of function
}
```

The type list specifies the type of exceptions that may be thrown. Throwing any other type of exception will cause abnormal program termination.
If we wish to prevent a function from throwing any exception, we may do so bymaking the type list empty i.e.

```
return_type function_name(arg_list) throw()
{
//function body
}
```

EXCEPTION SPECIFICATION FOR FUNCTION

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

void calculate(int a,int b) throw(int)
{
      if(b==0)
              throw 2.0;
      if(b<0)
             throw 1;
      cout<<"\nQuotation is :"<<a/b;
 }
```

```
int main()
{
      int c,d;
      cout<<"\nEnter the value of c and d:";
      cin>>c>>d;
      try {
             calculate(c,d);
      }
      catch(double)
      {
             cout<<"\nException as Second operand is -ve";
      }

      return 0;
 }
```

Output1:

```
Enter the value of c and d:
5
0
```

Output2:

```
Enter the value of c and d:
5
-2
```

In first run, the calculate function is throwing double value which is not specified in the throw list. Hence the program terminates.
In second run, the calculate function is throwing an int exception, but there is no catch handler for int. So the program terminates.

HANDLING UNCAUGHT AND UNEXPECTED EXCEPTIONS

Handling uncaught exceptions:

If the exception is thrown and no exception handler is found(i.e. the exception is not caught) - the program calls the terminate() function. We can specify our own termination function with the set_terminate() function.

```
#include<iostream>
using namespace std;
void test_handler()
{
      cout<<”Program is terminated....”;
}
```

```
int main()
{
       set_terminate(test_handler);
       try
       {
              cout<<”Inside try block.”<<endl;
             throw 10;
      }
      catch(char c)
      {
             cout<<”caught exception”;
      }

      return 0;
}
```

Output:

```
Inside try block
Program is terminated....
```

Here, int is thrown but there is no catch handler to catch the int exception. So, the program is terminates calling the test_handler() function.

HANDLING UNCAUGHT AND UNEXPECTED EXCEPTIONS

Handling unexpected exceptions:

Similar to uncatch exception, When a function attempts to throw an exception that is not in the throw list, unexpected() function is invoked. Like in terminate() function, we can specify the function that is called by the unexpected() with the help of set_unexpected() function.

```
#include<iostream>
using namespace std;
void test_unexpected()
{
      cout<<"Program terminated due to unexpected exception"<<endl;
}
void calculate( int a, int b) throw(int)
{
     if(b == 0)
           throw 'A';
     if(b<0)
           throw 1.0;    //double value is thrown which is not allowed
     cout<<"Result is :"<<a/b<<endl;
}
```

```
int main()
{
     set_unexpected(test_unexpected);
     try
     {
           calculate(2,-7);
     }
     catch(int c)
     {
           cout<<"Integer Exception"<<endl;
     }
     catch(double d)
     {
           cout<<"Double Exception"<<endl;
     }
     return 0;
}
```

Output:

```
Program terminated due
to unexpected exception
```

Here, calculate() function is trying to throw a double value which is not included in the throw list, So, an unexpected exception occurs which calls the test_unexpected().

THANK YOU