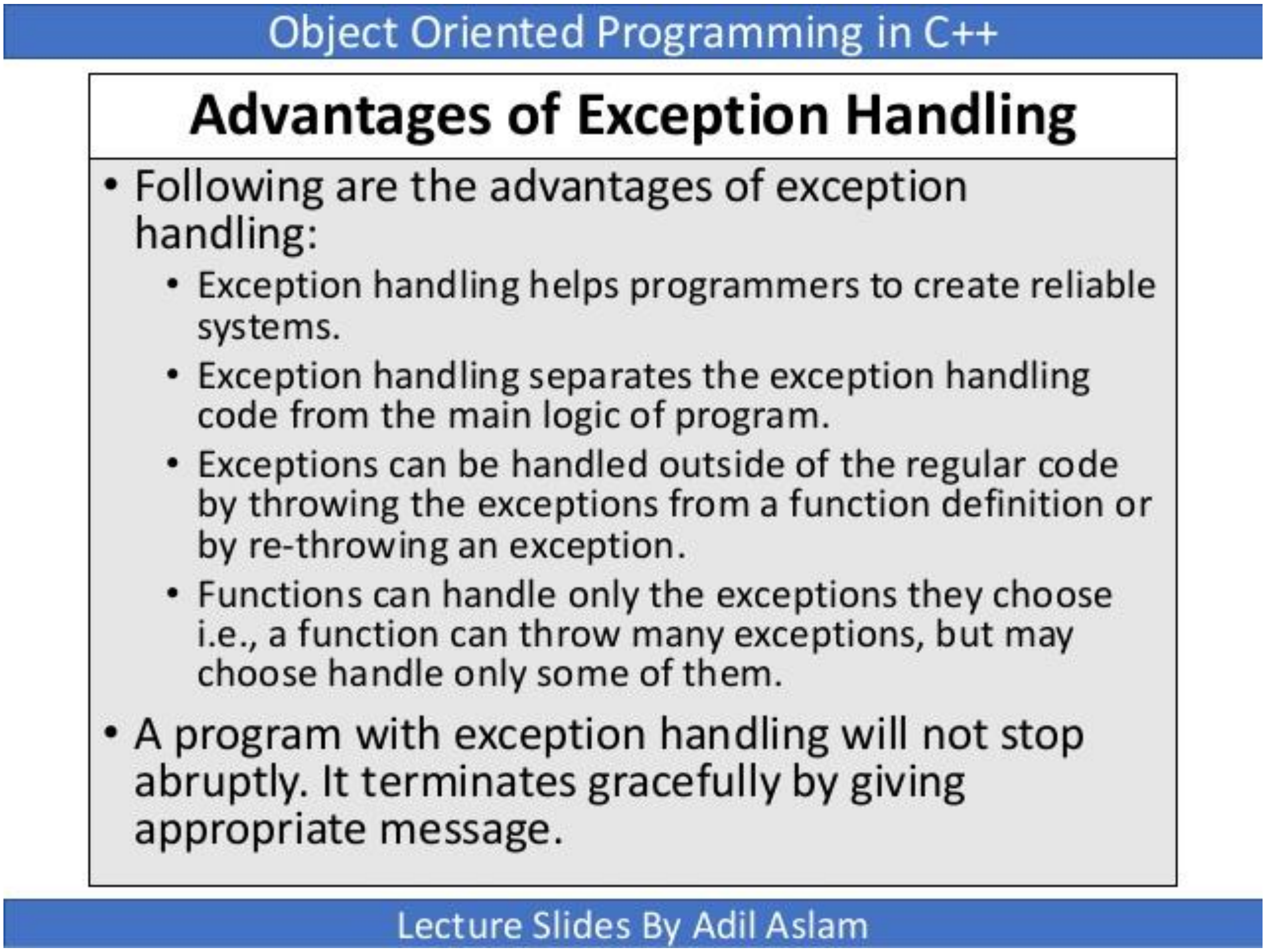


Object Oriented Programming in C++

Advantages of Exception Handling

- Following are the advantages of exception handling:
  - Exception handling helps programmers to create reliable systems.
  - Exception handling separates the exception handling code from the main logic of program.
  - Exceptions can be handled outside of the regular code by throwing the exceptions from a function definition or by re-throwing an exception.
  - Functions can handle only the exceptions they choose i.e., a function can throw many exceptions, but may choose handle only some of them.
- A program with exception handling will not stop abruptly. It terminates gracefully by giving appropriate message.

Lecture Slides By Adil Aslam

Exception Handling in c++

- An exception is an unexpected problem that arises during the execution of a program
- Exception Handling mechanism provide a way to transfer control from one part of a program to another. This makes it easy to separate the error handling code from the code written to handle the actual functionality of the program.
- C++ exception handling is built upon three keywords: try, catch and throw

try: Try block consists of the code that may generate exception. Exception are thrown from inside the try block.

throw: Throw keyword is used to throw an exception encountered inside try block. After the exception is thrown, the control is transferred to catch block.

catch: Catch block catches the exception thrown by throw statement from try block. Then, exception are handled inside catch block.

Every try catch should have a corresponding catch block. A single try block can have multiple catch blocks.

Syntax

```
try
{
   statements;
   ... ... ...
   throw exception;
}

catch (type argument)
{
   statements;
   ... ... ...
}
```

Multiple catch exception

Multiple catch exception statements are used when a user wants to handle different exceptions differently. For this, a user must include catch statements with different declaration.

Syntax

```
try
{
   body of try block
}

catch (type1 argument1)
{
   statements;
   ... ... ...
}

catch (type2 argument2)
{
   statements;
   ... ... ...
}
... ... ...
... ... ...
catch (typeN argumentN)
{
   statements;
   ... ... ...
}
```

Catch all exceptions

Sometimes, it may not be possible to design a separate catch block for each kind of exception. In such cases, we can use a single catch statement that catches all kinds of exceptions.

Syntax

```
catch (...)
{
   statements;
   ... ... ...
}
```

1. Following is a simple example to show exception handling in C++. The output of program explains flow of execution of try/catch blocks.

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

int main()
{
   int x = -1;

   // Some code
   cout << "Before try \n";
   try {
      cout << "Inside try \n";
      if (x < 0)
      {
         throw x;
         cout << "After throw (Never executed) \n";
      }
   }
   catch (int x ) {
      cout << "Exception Caught \n";
   }

   cout << "After catch (Will be executed) \n";
   return 0;
}
```

output:

```
Before try
Inside try
Exception Caught
After catch (Will be executed)
```

Ex of Exception handling:

```
#include <iostream>
using namespace std;
int main()
{
    int r,n,d;
    cout<<"enter n and d values";
    cin >>n>>d;
    try
    {
        if(d==0)
        throw d;
        r=n/d;
    }
    catch(int d)
    {
        cout<<"denominator is zero";
    }
    cout<<"r="<<r;
    return 0;
}
```

Ex:2

We can use multiple catch block in a single program using single try block.

```
//C++ program to divide two numbers using try catch block.

#include <iostream>
using namespace std;
int main()
{
   int a,b;
   cout << "Enter 2 numbers: ";
   cin >> a >> b;
   try
   {
      if (b != 0)
      {
         float div = (float)a/b;
         if (div < 0)
            throw 'e';
         cout << "a/b = " << div;
      }
      else
         throw b;

   }
   catch (int e)
   {
      cout << "Exception: Division by zero";
   }
   catch (char st)
   {
      cout << "Exception: Division is less than 1";
   }
   catch(...)
   {
      cout << "Exception: Unknown";
   }
     return 0;
}
```

output:

```
$ ./a.out
Enter 2 numbers: 4
0
Exception: Division by zero
$ ./a.out
Enter 2 numbers: 6
-1
Exception: Division is less than 1
$ ./a.out
Enter 2 numbers: 4
0
```

```
Exception: Division by zero
```

Ex: 3

```
 #include<iostream>
 using namespace std;
int main()
      {
           int a=2;

             try
             {

                 if(a==1)
                     throw a;                  //throwing integer exception

                 else if(a==2)
                     throw 'A';                //throwing character exception

                 else if(a==3)
                     throw 4.5;                 //throwing float exception

             }
             catch(int a)
             {
                 cout<<"\nInteger exception caught.";
             }
             catch(char ch)
             {
                 cout<<"\nCharacter exception caught.";
             }
             catch(double d)
             {
                 cout<<"\nDouble exception caught.";
             }

             cout<<"\nEnd of program.";

      }
```

output:

```
$ ./a.out

Character exception caught.
End of program
```

2. There is a special catch block called 'catch all' catch(…) that can be used to catch all types of exceptions. For example, in the following program, an int is thrown as an exception, but there is no catch block for int, so catch(…) block will be executed.

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

int main()
{
    try  {
       throw 10;
    }
    catch (char *excp)  {
        cout << "Caught " << excp;
    }
    catch (...)  {
        cout << "Default Exception\n";
    }
    return 0;
}
output:
Default Exception
```

3) Implicit type conversion doesn’t happen for primitive types. For example, in the following program ‘a’ is not implicitly converted to int

```
#include <iostream>
using namespace std;
  int main()
{
    try  {
       throw 'a';
    }
    catch (int x)  {
        cout << "Caught " << x;
    }
    catch (...)  {
        cout << "Default Exception\n";
    }
    return 0;
}
Output:
```

```
Default Exception
```

4) If an exception is thrown and not caught anywhere, the program terminates abnormally. For example, in the following program, a char is thrown, but there is no catch block to catch a char.

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

int main()
{
    try  {
       throw 'a';
    }
    catch (int x)  {
        cout << "Caught ";
    }
    return 0;
}
output:
terminate called after throwing an instance of 'char'
Aborted (core dumped)
```

5) In C++, try-catch blocks can be nested. Also, an exception can be re-thrown using “throw; ”

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

int main()
{
    try {
        try  {
            throw 20;
        }
        catch (int n) {
             cout << "Handle Partially ";
             throw;   //Re-throwing an exception
        }
    }
    catch (int n) {
        cout << "Handle remaining ";
    }
    return 0;
}
```

output:

```
Handle Partially Handle remaining
```

6) When an exception is thrown, all objects created inside the enclosing try block are destructed before the control is transferred to catch block.

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

class Test {
public:
   Test() { cout << "Constructor of Test " << endl; }
  ~Test() { cout << "Destructor of Test "  << endl; }
};

int main() {
  try {
    Test t1;
    throw 10;
  } catch(int i) {
    cout << "Caught " << i << endl;
  }
}
```

output:

```
Constructor of Test
Destructor of Test
Caught 10
```

Ex: catching all exceptions

```
#include<iostream>

using namespace std;

void test(int x)

{

try

{

if(x==0)

throw x;

if(x==-1)

throw 1.0;

if(x==1)

throw 'a';

}

catch(...)

{

cout<<"\nexception caught";

}

}


int main()

{

cout<<"testing default exception";

test(0);

test(-1);

test(1);

return 0;

}
```

output:

```
testing default exception
exception caught
exception caught
exception caught
```

Rethrowing exceptions

```
#include<iostream>
using namespace std;
void fun()
{
try
{
throw "hello";
}
catch(const char *a)
{
cout<<"\n exception caught inside func"<<a;
throw "hai"; // or throw; it will take arg as hello
}
}
int main()
{
try
{
fun();
}
catch(const char *a)
{
```

```
cout<<"\n exception caught inside main()"<<a;
}
return 0;
}
```

output:

```
exception caught inside funchello
 exception caught inside main()hai
```

Ex:Nested try – block

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

int main()
{
   try {
           try
           {
        throw 20;
          }
      catch (int n) {
         cout << "Handle Partially ";
//         throw;   //Re-throwing an exception
     }
throw 10;  //here with throw argument is must, otherwise we will get an error
  }
   catch (int n) {
      cout << "Handle remaining ";
   }
   return 0;
}
```

output:

```
Handle Partially
 Handle remaining
```

Exception Specifications:

It is possible to restrict a function to throw only certain specified exceptions.

This is achieved by   a throw list clause to the function definition.

Syntax:

```
type function(arg list) throw( type – list)
{
stmts;
}
```

the type-list specifies the type of exceptions that may be thrown. Throwing any other type will cause a abnormal termination.

If we wish to prevent a function from throwing any exception , we make the type-list empty.(i.e throw();)

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

void test(int x) throw(int,double)
{
if(x==0)
throw x;
else
if (x==1)
throw 'x';
else
if(x==-1)
throw 1.0;
}

int main()

{

try

{

cout<<"testing throw restrictions:";

test(0);

test(1);

test(-1);

}

catch(char c)

{

cout<<"\ncaught a charexception";

}

catch(int c)

{

cout<<"\ncaught a int exception";

}

catch(double c)

{

cout<<"\ncaught a double exception";

}

cout<<"\n end of main";

return 0;

}
```

output:

```
testing throw restrictions:
```

```
caught a int exception
 end of main
```

Stack Unwinding in C++

The process of removing function entries from function call stack at run time is called Stack Unwinding. Stack Unwinding is generally related to Exception Handling. In C++, when an exception occurs, the function call stack is linearly searched for the exception handler, and all the entries before the function with exception handler are removed from the function call stack. So exception handling involves Stack Unwinding if exception is not handled in same function (where it is thrown).

```
#include <iostream>
   using namespace std;
  // A sample function f1() that throws an int exception
void f1() throw (int) {
  cout<<"\n f1() Start ";
  throw 100;
  cout<<"\n f1() End ";
}
  // Another sample function f2() that calls f1()
void f2() throw (int) {
  cout<<"\n f2() Start ";
  f1();
  cout<<"\n f2() End ";
}
  //Another sample function f3() that calls f2() and handles exception thrown by f1()
void f3() {
  cout<<"\n f3() Start ";
  try {
    f2();
  }
  catch(int i) {
   cout<<"\n Caught Exception: "<<i;

  }

  cout<<"\n f3() End";

}

// A driver function to demonstrate Stack Unwinding  process

int main()

{

  f3();

  return 0;

}
```

output:

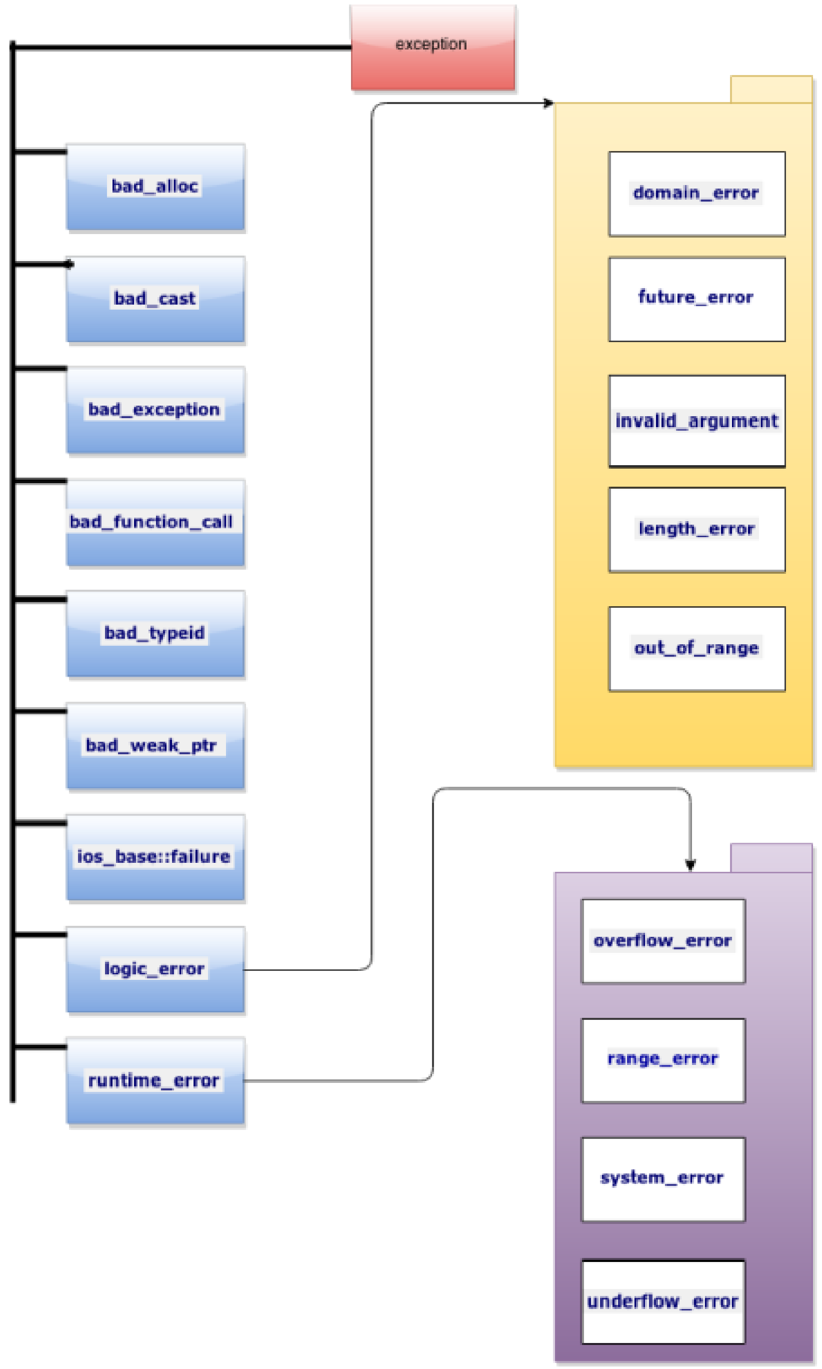
```
f3() Start

 f2() Start

 f1() Start

 Caught Exception: 100

 f3() End
```

In the above program, when f1() throws exception, its entry is removed from the function call stack (because f1() doesn’t contain exception handler for the thrown exception), then next entry in call stack is looked for exception handler. The next entry is f2(). Since f2() also doesn’t have handler, its entry is also removed from function call stack. The next entry in function call stack is f3(). Since f3() contains exception handler, the catch block inside f3() is executed, and finally the code after catch block is executed. Note that the following lines inside f1() and f2() are not executed at all.

C++ Standard Exceptions

C++ provides a list of standard exceptions defined in **<exception>** which we can use in our programs. The information about happened exception is provided by **what()** member function of the exception class

Exceptions are arranged in a parent-child class hierarchy shown below −



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

Exceptions derived directly from **exception** class:

bad_alloc	Happens when there is a failure of memory allocation

bad_cast	Is thrown when dynamic_cast is used incorrect
bad_exception	Exception that is thrown by unexpected handler
bad_function_call	Thrown when an empty (not implemented) function is called
bad_typeid	Thrown by typeid function
bad_weak_ptr	Exception that can be thrown by shared_ptr class constructor
ios_base::failure	Base class for all the stream exceptions
logic_error	Base class for some logic error exceptions
runtime_error	Base class for some runtime error exceptions

Exceptions derived indirectly from **exception** class through **logic_error**:

domain_error	Thrown when an error of function domain happens
future_error	Reports an exception that can happen in **future** objects(See more info about **future** class)
invalid_argument	Exception is thrown when invalid argument is passed
length_error	Is thrown when incorrect length is set
out_of range error	Is thrown when out of range index is used

Exceptions derived indirectly from **exception** class through **runtime_error**:

overflow_error	Arithmetic overflow exception
range_error	Signals range error in computations
system_error	Reports an exception from operating system
underflow_error	Arithmetic underflow exceptions

Ex:

```
int f(int n)
{
   if (n == 1)
 {
      throw logic_error("0");
      cout << "7" << endl;
   }

}

int main()
{
   try {
       f(1);
   }
   catch (exception &e) {
      cout << e.what() << endl;
   }

   return 0;
}
```

output:

```
0
```

Create our own exceptions:

Sometimes, you will need to create your own exception classes. This can be done for different purposes. For example, you want to send some information from the place, where exception happened, to the catch block. It can be done by extending class **exception.** In this case, you can create a constructor for the derived class and override its member function **what().**

```
#include<iostream>
#include<exception>
using namespace std;
class ZeroDivisionException :public exception
{
public:
	ZeroDivisionException(int data)
	{
		someData = data;
	}
	//override what function
	const char* what()
	{
		return "Zero division error";
	}
	int someData;
};

double divide(double a, double b)
{
	if (b == 0) //division by zero!!!!
		throw ZeroDivisionException(a);
	else
		return a/b;
}
```

```
int main()
{
try
{
       double res = divide(1, 0);
}
catch (ZeroDivisionException e)
{
       cout << e.what() << endl;
       cout << "Trying to divide " << e.someData << " by zero" << endl;
}
return 0;
}
```

output:

```
Zero division error
Trying to divide 1 by zero
```

ZeroDivisionException class stores an integer data member **someData.** Now we can get the information from the place, where exception happened, in the catch block. For this purpose, we will re-write **divide** function
Parameter **a** is passed and returned back from the function. We can use it in catch block