C++ Exception Handling[1]

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,** and **throw**.

- **throw:** A program throws an exception when a problem shows up. This is done using a **throw** keyword.
- **catch:** A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.
- **try:** A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

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 looks like the following:

```
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** statements. The operand of the throw statements 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.

Following is an example of throwing an exception when dividing by zero condition occurs:

```
double division(int a, int b)
{
   if( b == 0 )
   {
      throw "Division by zero condition!";
   }
   return (a/b);
}
```

Catching Exceptions:

The **catch** block following the **try** block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try
{
   // protected code
}catch( ExceptionName e )
{
  // code to handle ExceptionName exception
}
```

Above code will catch an exception of **ExceptionName** type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows:

```
try
{
   // protected code
}catch(...)
{
  // code to handle any exception
}
```

The following is an example, which throws a division by zero exception and we catch it in catch block.

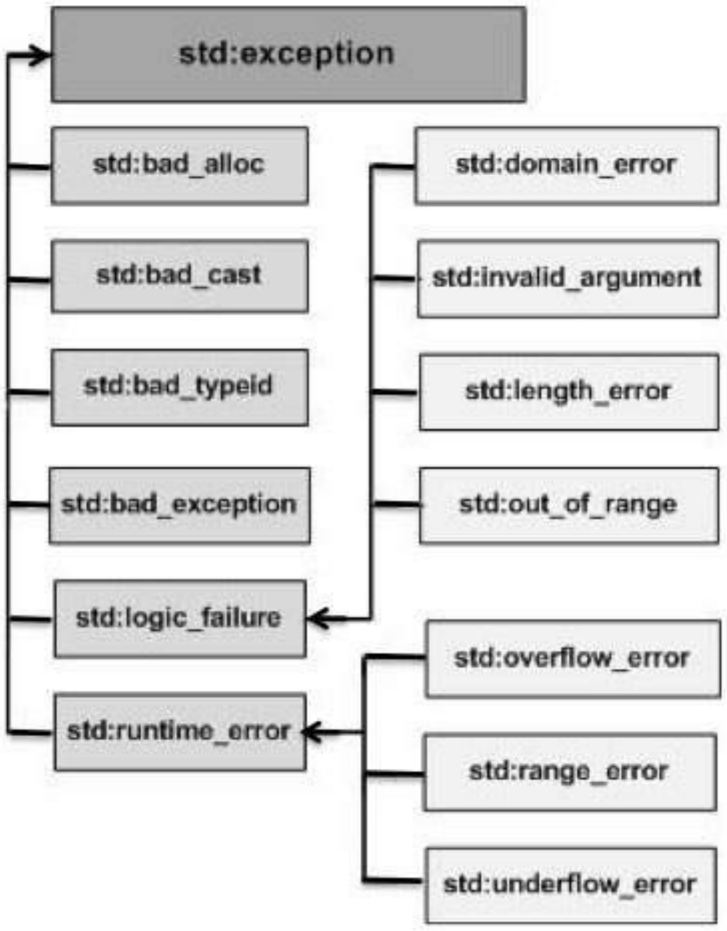
```
#include <iostream>
using namespace std;

double division(int a, int b)
{
   if( b == 0 )
   {
      throw "Division by zero condition!";
   }
   return (a/b);
}

int main ()
{
   int x = 50;
   int y = 0;
   double z = 0;

   try {
     z = division(x, y);
     cout << z << endl;
   }catch (const char* msg) {
     cerr << msg << endl;
   }

   return 0;
}
```

Because we are raising an exception of type **const char***, so while catching this exception, we have to use const char* in catch block. If we compile and run above code, this would produce the following result:

```
Division by zero condition!
```

C++ Standard Exceptions:

C++ provides a list of standard exceptions defined in **<exception>** which we can use in our programs. These are arranged in a parent-child class hierarchy shown below:



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

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:

You can define your own exceptions by inheriting and overriding **exception** class functionality. Following is the example, which shows how you can use std::exception class to implement your own exception in standard way:

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

struct MyException : public exception
{
  const char * what () const throw ()
  {
    return "C++ Exception";
  }
};

int main()
{
  try
  {
    throw MyException();
  }
  catch(MyException& e)
  {
    std::cout << "MyException caught" << std::endl;
    std::cout << e.what() << std::endl;
  }
  catch(std::exception& e)
  {
    //Other errors
  }
}
```

This would produce the following result:

```
MyException caught
C++ Exception
```

Here, **what()** is a public method provided by exception class and it has been overridden by all the child exception classes. This returns the cause of an exception.

C++ Templates[2]

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

There is a single definition of each container, such as **vector**, but we can define many different kinds of vectors for example, **vector <int>** or **vector <string>**.

You can use templates to define functions as well as classes, let us see how do they work:

Function Template:

The general form of a template function definition is shown here:

```
template <class type> ret-type func-name(parameter list)

{

   // body of function

}
```

Here, type is a placeholder name for a data type used by the function. This name can be used within the function definition.

The following is the example of a function template that returns the maximum of two values:

```
#include <iostream>

#include <string>
```

```
using namespace std;


template <typename T>
inline T const& Max (T const& a, T const& b)
{
    return a < b ? b:a;
}
int main ()
{


    int i = 39;
    int j = 20;
    cout << "Max(i, j): " << Max(i, j) << endl;


    double f1 = 13.5;
    double f2 = 20.7;
    cout << "Max(f1, f2): " << Max(f1, f2) << endl;


    string s1 = "Hello";
    string s2 = "World";
    cout << "Max(s1, s2): " << Max(s1, s2) << endl;


   return 0;
}
```

If we compile and run above code, this would produce the following result:

```
Max(i, j): 39

Max(f1, f2): 20.7

Max(s1, s2): World
```

Class Template:

Just as we can define function templates, we can also define class templates. The general form of a generic class declaration is shown here:

```
template <class type> class class-name {

.

.

.

}
```

Here, **type** is the placeholder type name, which will be specified when a class is instantiated. You can define more than one generic data type by using a comma-separated list.

Following is the example to define class Stack<> and implement generic methods to push and pop the elements from the stack:

```
#include <iostream>

#include <vector>

#include <cstdlib>

#include <string>

#include <stdexcept>


using namespace std;
```

```
template <class T>
class Stack {
  private:
    vector<T> elems;     // elements

  public:
    void push(T const&);  // push element
    void pop();               // pop element
    T top() const;            // return top element
    bool empty() const{       // return true if empty.
        return elems.empty();
    }
};

template <class T>
void Stack<T>::push (T const& elem)
{
    // append copy of passed element
    elems.push_back(elem);
}

template <class T>
void Stack<T>::pop ()
{
    if (elems.empty()) {
        throw out_of_range("Stack<>::pop(): empty stack");
    }
```

```
       // remove last element
    elems.pop_back();
}

template <class T>
T Stack<T>::top () const
{
    if (elems.empty()) {
        throw out_of_range("Stack<>::top(): empty stack");
    }
       // return copy of last element
    return elems.back();
}

int main()
{
    try {
        Stack<int>         intStack;  // stack of ints
        Stack<string> stringStack;    // stack of strings

        // manipulate int stack
        intStack.push(7);
        cout << intStack.top() <<endl;

        // manipulate string stack
        stringStack.push("hello");
        cout << stringStack.top() << std::endl;
        stringStack.pop();

        stringStack.pop();

    }

    catch (exception const& ex) {

        cerr << "Exception: " << ex.what() <<endl;

        return -1;

    }

}
```

If we compile and run above code, this would produce the following result:

```
7

hello

Exception: Stack<>::pop(): empty stack
```

References:

[1,2: https://www.tutorialspoint.com]

Further online reading: https://msdn.microsoft.com/en-us/library/y097fkab.aspx