09/22/2014 discussion

<http://www.tutorialspoint.com/cplusplus/cpp_exceptions_handling.htm>

C++ Exception Handling

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;

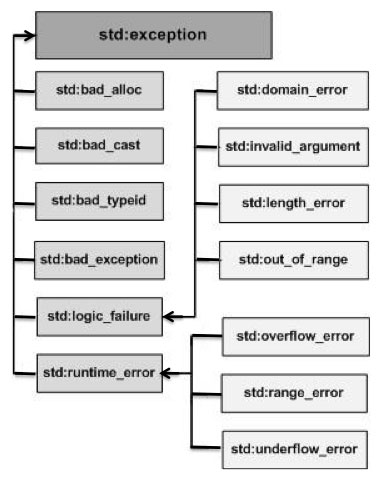
}

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.

Unit Testing in C++ and Objective-C just got easier

<http://www.levelofindirection.com/journal/2010/12/28/unit-testing-in-c-and-objective-c-just-got-easier.html>

## Introducing CATCH

CATCH is a brand new unit testing framework for C, C++ and Objective-C. It stands for 'C++Automated Test Cases in Headers', although that shouldn't downplay the Objective-C bindings. In fact my initial motivation for starting it was dissatisfaction with OCUnit.

### *Why do we need another Unit Testing framework for C++ or Objective-C?*

There are [plenty of unit test frameworks for C++](http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks#C.2B.2B). [Not so many for Objective-C](http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks#Objective-C) - which primarily has OCUnit (although you could also coerce a C or C++ framework to do the job).

They all have their strengths and weaknesses. But most suffer from one or more of the following problems:

* Most take their cues from JUnit, which is unfortunate as JUnit is very much a product of Java. The idiom-mismatch in C++ is, I believe, one of the reasons for the slow uptake of unit testing and TDD in C++.
* Most require you to build libraries. This can be a turn off to anyone who wants to get up and running quickly - especially if you just want to try something out. This is especially true of exploratory TDD coding.
* There is typically a certain amount of ceremony or boilerplate involved. Ironically the frameworks that try to be faithful to C++ idioms are often the worst culprits. Eschewing macros for the sake of purity is a great and noble goal - in application development. For a DSL for testing application code, especially since preprocessor information (e.g. file and line number) are required anyway) the extra verbosity seems too high a price to pay to me.
* Some pull in external dependencies
* Some involve a code generation step

The list goes on, but these are the criteria that really had me disappointed in what was out there, and I'm not the only one. But can these be overcome? Can we do even better if we start again without being shackled to the ghost of JUnit?

### *What's the CATCH?*

Well, to start, here's my three step process for getting up and running with CATCH:

1. Download the headers from [github](https://github.com/philsquared/Catch) into subfolder of your project
2. #include "catch.hpp"
3. There is no step 3!

Ok, you might need to actually write some tests as well. Let's have a look at how you might do that:

[Update: Since my original post I have made some small, interface breaking, changes - for example the name of the header included below. I have updated this post to reflect these changes - in case you were wondering]

#include "catch\_with\_main.hpp"  
  
TEST\_CASE( "stupid/1=2", "Prove that one equals 2" )  
{  
    int one = 2;  
    REQUIRE( one == 2 );  
}

Short and to the point, but this snippet already shows a lot of what's different about CATCH:

* The assertion macro is REQUIRE( expression ), rather than the, now traditional, REQUIRE\_EQUALS( lhs, rhs ), or similar. Don't worry - lhs and rhs are captured anyway - more on this later.
* The test case is in the form of a free function. We could have made it a method, but we don't need to
* We didn't name the function. We named the test case. This frees us from couching our names in legal C++ identifiers. We also provide a longer form description that serves as an active comment
* Note, too, that the name is hierarchical (as would be more obvious with more test cases). The convention is, as you might expect, "root/branch1/branch2/.../leaf". This allows us to easily group test cases without having to explicitly create suites (although this can be done too).
* There is no test context being passed in here (although it could have been hidden by the macro - it's not). This means that you can freely call helper functions that, themselves, contain REQUIRE() assertions, with no additional overhead. Even better - you can call into application code thatcalls back into test code. This is perfect for mocks and fakes.
* We have not had to explicity register our test function anywhere. And by default, if no tests are specified on the command line, all (automatically registered) test cases are executed.
* We even have a main() defined for us by virtue of #including "catch\_with\_main.hpp". If we just #include that in one dedicated cpp file we would #include "catch.hpp' in our test case files instead. We could also write our own main that drives things differently.

That's a lot of interesting stuff packed into just a few lines of test code. It's also got more wordy than I wanted. Let's take a bit more of a tour by example.

### *Information is power*

Here's another contrived example:

TEST\_CASE( "example/less than 7", "The number is less than 7" )  
{  
    int notThisOne = 7;  
  
    for( int i=0; i < 7; ++i )  
    {  
        REQUIRE( notThisOne > i+1  );  
    }  
}

In this case the bug is in the test code - but that's just to make it self contained. Clearly the requirement will be broken for the last iteration of i. What information do we get when this test fails?

notThisOne > i+1 failed for: 7 > 7

(We also get the file and line number, but they have been elided here for brevity). Note we get the original expression and the values of the lhs and rhs as they were at the point of failure. That's not bad, considering we wrote it as a complete expression. This is achieved through the magic of expression templates, which we won't go into the details of here (but feel free to look at the source - it's probably simpler than you think).

Most of the time this level of information is exactly what you need. However, to keep the use of expression templates to a minimum we only decompose the lhs and rhs. We don't decompose the value of i in this expression, for example. There may also be other relevant values that are not captured as part of the test expression.

In these cases it can be useful to log additional information. But then you only want to see that information in the event of a test failure. For this purpose we have the INFO() macro. Let's see how that would improve things:

TEST\_CASE( "example/less than 7", "The number is less than 7" ){  
    int notThisOne = 7;  
  
    for( int i=0; i < 7; ++i ){  
        INFO( "i=" << i );  
        REQUIRE( notThisOne > i+1  );  
    }  
}

This gives us:

info: 'i=6' notThisOne > i+1 failed for: 7 > 7

But if we fix the test, say by making the for loop go to i < 6, we now see no output for this test case (although we can, optionally, see the output of successful tests too).

### *A SECTION on specifications*

There are different approaches to unit testing that influence the way the tests are written. Each approach requires a subtle shift in features, terminology and emphasis. One approach is often associated with [Behaviour Driven Development](http://en.wikipedia.org/wiki/Behavior_Driven_Development)(BDD). This aims to present test code in a language neutral form - encouraging a style that reads more like a specification for the code under test.

While CATCH is not a dedicated BDD framework it offers a several features that make it attractive from a BDD perspective:

* The hiding of function and method names, writing test names and descriptions in natural language
* The automatic test registration and default main implementation eliminate boilerplate code that would otherwise be noise
* Test data generators can be written in a language neutral way (not fully implemented at time of writing)
* Test cases can be divided and subdivided into SECTIONs, which also take natural language names and descriptions.

We'll look at the test data generators another time. For now we'll look at the SECTION macro.

Here's an example (from the unit tests for CATCH itself):

TEST\_CASE( "succeeding/Misc/Sections/nested", "nested SECTION tests" )  
{  
    int a = 1;  
    int b = 2;  
      
    SECTION( "s1", "doesn't equal" )  
    {  
        REQUIRE( a != b );  
        REQUIRE( b != a );  
  
        SECTION( "s2", "not equal" )  
        {  
            REQUIRE\_FALSE( a == b);  
        }  
    }  
}

Again, this is not a great example and it doesn't really show the BDD aspects. The important point here is that you can divide your test case up in a way that mirrors how you might divide a specification document up into sections with different headings. From a BDD point of view your SECTION descriptions would probably be your "should" statements.

There is more planned in this area. For example I'm considering offering a GIVEN() macro for defining instances of test data, which can then be logged.

In Kevlin Henney's LHR framework, mentioned in the opening link, he used SPECIFICATION where I have used TEST\_CASE, and PROPOSITION for my top level SECTIONs. His equivalent of my nested SECTIONs are (or were) called DIVIDERs. All of the CATCH macro names are actually aliases for internal names and are defined in one file (catch.hpp). If it aids utility for BDD or other purposes, the names can be aliased differently simply by creating a new mapping file and using that.

### *CATCH up*

There is much more to cover but I wanted to keep this short. I'll follow up with more. For now here's a (yet another) list of some of the key features I haven'talready covered:

* Entirely in headers
* No external dependencies
* Even test fixture classes and methods are self registering
* Full Objective-C bindings
* Failures (optionally) break into the interactive debugger, if available
* Floating point tolerances supported in an easy to use way
* Several reporter classes included - including a JUnit compatible xml reporter. More can be supplied

<https://github.com/philsquared/catch/blob/master/docs/tutorial.md>