What the What?

CException is simple exception handling in C. It is significantly faster than full-blown C++ exception handling but loses some flexibility. It is portable to any platform supporting setjmp/longjmp.

So what's it good for? Mostly error handling. Passing errors down a long chain of function calls gets ugly. Sometimes really ugly. So what if you could just specify certain places where you want to handle errors, and all your errors were transferred there? Let's try a lame example:

Error Handling without CException:

```
UINT32 functionC(void) {

  //do some stuff

  if (there_was_a_problem)

    return ERR_BAD_BREATH;

  //this stuff never gets called because of error

}


UINT32 functionB(void) {

  //do some stuff

  UINT32 retval = functionC( );

  if (retval)

    return retval;

  //this stuff never gets called because of error

}

UINT32 functionA(void) {

  //do some stuff

  UINT32 retval = functionB( );

  if (retval)

    return retval;

  //this stuff never gets called because of error

}

void main(void) {

  //do some stuff

  UINT32 retval = functionA( );

  if (retval) {

    puts("THERE WAS AN ERROR");

    return retval;

  }

  //this stuff never gets called because of error

}
```

Error Handling with CException:

```
void functionC(void) {

  //do some stuff

  if (there_was_a_problem)

    Throw(ERR_BAD_BREATH);

  //this stuff never gets called because of error

}


void functionB(void) {

  //do some stuff

  functionC( );

  //this stuff never gets called because of error

}

void functionA(void) {

  //do some stuff

  functionB( );

  //this stuff never gets called because of error

}

void main(void) {

  CEXCEPTION_T e;

  Try {

    //do some stuff

    functionA( );

    //this stuff never gets called because of error

  }

  Catch(e) {

    puts("THERE WAS AN ERROR");

    return e;

  }

}
```

CException uses C standard library functions setjmp and longjmp to operate. As long as the target system has these two functions defined, this library should be useable with very little configuration. It even supports environments where multiple program flows are in use, such as real-time operating systems... we started this project for use in embedded systems... but it obviously can be used for larger systems too.

There are about a gabillion exception frameworks using a similar setjmp/longjmp method out there... and there will probably be more in the future. Unfortunately, when we started our last embedded project, all those that existed either (a) did not support multiple tasks (therefore multiple stacks) or (b) were way more complex than we really wanted. CException was born.

Why?

0. It's ANSI C, and it beats passing error codes around.

1. You want something simple... CException throws a single id. You can define those ID's to be whatever you like. You might even choose which type that number is for your project. But that's as far as it goes. We weren't interested in passing objects or structs or strings... just simple error codes. Fast. Easy to Use. Easy to Understand.

2. Performance... CException can be configured for single tasking or multitasking. In single tasking, there is very little overhead past the setjmp/longjmp calls (which are already fast). In multitasking, your only additional overhead is the time it takes you to determine a unique task id (0 to num_tasks).

Where?

For the latest version, go to http://cexception.sourceforge.net

How?

Code that is to be protected are wrapped in Try { } blocks. The code inside the Try block is "protected", meaning that if any Throws occur, program control is directly transferred to the start of the Catch block. The Catch block immediately follows the Try block. It's ignored if no errors have occurred.

A numerical exception ID is included with Throw, and is passed into the Catch block. This allows you to handle errors differently or to report which error has occurred... or maybe it just makes debugging easier so you know where the problem was Thrown.

Throws can occur from anywhere inside the Try block, directly in the function you're testing or even within function calls (nested as deeply as you like). There can be as many Throws as you like, just remember that execution of the guts of your Try block ends as soon as the first Throw is triggered. Once you throw, you're transferred to the Catch block. A silly example:

```
void SillyExampleWhichPrintsZeroThroughFive(void)
{
  CEXCEPTION_T e;
  int i;
  while (i = 0; i < 6; i++) {
    Try {
      Throw(i);
      //This spot is never reached
    }
    Catch(e) {
      printf("%i ", e);
    }
  }
}
```

Limitations

This library was made to be as fast as possible, and provide basic exception handling. It is not a full-blown exception library like C++. Because of this, there are a few limitations that should be observed in order to successfully utilize this library:

0. Do not directly "return" from within a Try block, nor "goto" into or out of a Try block.
   - The "Try" macro allocates some local memory and alters a global pointer. These are cleaned up at the top of the "Catch" macro. Gotos and returns would bypass some of these steps, resulting in memory leaks or unpredictable behavior.

1. If **(a)** you change local (stack) variables within your Try block, **and (b)** wish to make use of the updated values after an exception is thrown, those variables should be made volatile.
   - Note that this is ONLY for locals and ONLY when you need access to them after a throw.
   - Compilers optimize (and thank goodness they do). There is no way to guarantee that the actual memory location was updated and not just a register unless the variable is marked volatile.

2. Memory which is malloc'd within a Try block is not automatically released when an error is thrown. This will sometimes be desirable, and othertimes may not. It will be the responsibility of the code you put in the Catch block to perform this kind of cleanup.
   - There's just no easy way to track malloc'd memory, etc., without replacing or wrapping malloc calls or something like that. This is a light framework, so these options were not desirable.

Details!

Try { }

Try is a macro which starts a protected block. It MUST be followed by a pair of braces or a single protected line (similar to an 'if'), enclosing the data that is to be protected. It MUST be followed by a Catch block (don't worry, you'll get compiler errors to let you know if you mess any of that up).

The Try block is your protected block. It contains your main program flow, where you can ignore errors (other than a quick Throw call). You may nest multiple Try blocks if you want to handle errors at multiple levels, and you can even rethrow an error from within a nested Catch.

Catch(e) { }

Catch is a macro which ends the Try block and starts the error handling block. The catch block is executed if and only if an exception was thrown while within the Try block. This error was thrown by a Throw call somewhere within Try (or within a function called within Try, or a function called by a function called within Try... you get the idea.).

Catch receives a single id of type CEXCEPTION_T which you can ignore or use to handle the error in some way. You may throw errors from within Catches, but they will be caught by a Try wrapping the Catch, not the one immediately preceeding.

Throw(e)

The method of throwing an error. Throws should only occur from within a protected (Try...Catch) block, though it may easily be nested many function calls deep without an impact on performance or functionality. Throw takes a single argument, which is an exception id which will be passed to Catch as the reason for the error. If you wish to Rethrow an error, this can be done by calling Throw(e) with the error code you just caught. It IS valid to throw from a catch block.

ExitTry()

On rare occasion, you might want to immediately exit your current Try block but NOT treat this as an error. Don't run the Catch. Just start executing from after the Catch as if nothing had happened... That's what ExitTry is for.

Configuration

CException is a mostly portable library. It has one universal dependency, plus some macros which are required if working in a multi-tasking environment.

1. The standard C library setjmp must be available. Since this is part of the standard library, it's all good.

2. If working in a multitasking environment, you need a stack frame for each task. Therefore, you must define methods for obtaining an index into an array of frames and to get the overall number of id's are required. If the OS supports a method to retrieve Task ID's, and those Tasks are number 0, 1, 2... you are in an ideal situation. Otherwise, a more creative mapping function may be required. Note that this function is likely to be called twice for each protected block and once during a throw. This is the only added overhead in the system.

You have options for configuring the library, if the defaults aren't good enough for you. You can add defines at the command prompt directly. You can always include a configuration file before including CException.h. You can make sure CEXCEPTION_USE_CONFIG_FILE is defined, which will force make CException look for CExceptionConfig.h, where you can define whatever you like. However you do it, you can override any or all of the following:

CEXCEPTION_T

Set this to the type you want your exception id's to be. Defaults to an 'unsigned int'.

CEXCEPTION_NONE

Set this to a number which will never be an exception id in your system. Defaults to 0x5a5a5a5a.

CEXCEPTION_GET_ID

If in a multi-tasking environment, this should be set to be a call to the function described in #2 above. It defaults to just return 0 all the time (good for single tasking environments, not so good otherwise).

CEXCEPTION_NUM_ID

If in a multi-tasking environment, this should be set to the number of ID's required (usually the number of tasks in the system). Defaults to 1 (good for single tasking environments or systems where you will only use this from one task).

CEXCEPTION_NO_CATCH_HANDLER(id)

This macro can be optionally specified. It allows you to specify code to be called when a Throw is made outside of Try...Catch protection. Consider this the emergency fallback plan for when something has gone terribly wrong.

You may also want to include any header files which will commonly be needed by the rest of your application where it uses exception handling here. For example, OS header files or exception codes would be useful.

Finally, there are some hook macros which you can implement to inject your own target-specific code in particular places. It is a rare instance where you will need these, but they are here if you need them:

CEXCEPTION_HOOK_START_TRY - called immediately before the Try block

CEXCEPTION_HOOK_HAPPY_TRY - called immediately after the Try block if no exception was thrown

CEXCEPTION_HOOK_AFTER_TRY - called immediately after the Try block OR before an exception is caught

CEXCEPTION_HOOK_START_CATCH - called immediately before the catch

Testing

If you want to validate that CException works with your tools or that it works with your custom configuration, you may want to run the included test suite. This is the test suite (along with real projects we've used it on) that we use to make sure that things actually work the way we claim.

The test suite included makes use of the Unity Test Framework. It will require a native C compiler. The example makefile and rakefile both use MinGW's gcc. Modify either to include the proper paths to tools, then run 'make' to compile and run the test application.

C_COMPILER

The C compiler to use to perform the tests.

C_LIBS

The path to the C libraries (including setjmp).

UNITY_DIR

The path to the Unity framework (required to run tests... get it at http://github.com/ThrowTheSwitch/Unity if you lost it)

License?

This software is licensed under the MIT License: