Exceptions in C with Longjmp and Setjmp

Abstract

This document describes a very simple implementation (with many limitations) of a system to add exceptions on top of C using the libc calls longjump and setjump. This system does not pretend to be really useful in practice but it is a useful lesson about longjump and setjump with a fun example.

Introduction

Exception are a very powerful way to program error safe programs. Exceptions let you write straight code without testing for errors at each statement. In modern programming languages, such as C++, Java or C#, exceptions are expressed with the *try-throw-catch* statement.

```
...
try
{
   ...
   /* error prone statements */
   ...
}
catch(SomeExceptionType e)
{
   ...
   /* do something intelligent here*/
   ...
}
...
```

In previous example every exception raised by operations performed in *try-block* is passed to the right *catch-black*. If the exception type match **SomeExceptionType** than the code in that block is executed. Otherwise the exception is passed to the *try-block* that contains the actual one (if any).

Our solution is not a fully functional *try-throw-catch* system. It does not forward exceptions from one block to one more external if no handler is provided.

Real exception mechanisms need run-time support. We only want to explore the potentiality of **longjmp** and **setjmp** function with a non trivial example.

Longjmp And SetJmp

ANSI-C provide a lot of functions: math functions (**log**, **sqrt**...), string handling functions (**strdup**, **strcmp**, ...) and I/O functions (**getc**, **printf**, ...). All these functions are widely used and simple to understand (...**strtok** is not so intuitive after all...): only two functions are considered *strange beasts*.

These functions are **longjmp** and **setjmp**.

longjmp and **setjmp** are defined in *setjmp.h* header file...

```
#include <setjmp.h>
```

...and are defined as follows:

```
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

setjmp takes a **jmp_buf** type variable as only input and has a strange return behavior: it returns 0 when invoked directly and when **longjmp** is invoked with the same **jmp_buf** variable it returns the value passed as second argument of **longjmp**.

Do you think that this is obscure? Strange? Without sense?

Probably you are right! The behavior of there functions is really strange: you have a function (**setjmp** with two return values).

setjmp and **longjmp** mechanism works as follows: when **setjmp** is invoked the first time it returns 0 and fill the **jmp_buf** structure with the *calling environment* and the *signal mask*. The *calling environment* represents the state of registers and the point in the code where the function was called. When **longjmp** is called the state saved in the **jmp_buf** variable is copied back in the processor and computation starts over from the return point of **setjmp** function but the returned value is the one passed as second argument to **longjmp** function.

Probably now you are thinking something like: "Hey dude are you kiddin' me?". The replay is "No". The behavior is exactly the one stated before.

There are 10 kind of people in the world:

0. people thinking that this is awful (and probably are asking themselves why only two cases if there are 10 kind of people)
1. people thinking that it can be amazing!

The rest of document is for second ones.

Basic Try-Catch

First version is a real simple one. Probably if you are here you know the solution. (this solution was presented also by other authors... my contribution is represented by the second and the third version of the solution).

The general idea is to map **TRY** statement on **if** statement. The first time that it is invoked it return 0 and the executed code is the one stated in **then** branch. **CATCH** statement is simply the **else** statement. When the **THROW** statement is executed it simply calls the **longjmp** function with the second parameter equals to 1 (or anything not 0).

```
#include <stdio.h>
#include <setjmp.h>

#define TRY do{ jmp_buf ex_buf__; if( !setjmp(ex_buf__) ){
#define CATCH } else {
#define ETRY } }while(0)
#define THROW longjmp(ex_buf__, 1)

int
main(int argc, char** argv)
{
   TRY
   {
      printf("In Try Statement\n");
      THROW;
      printf("I do not appear\n");
   }
   CATCH
   {
      printf("Got Exception!\n");
   }
   ETRY;

   return 0;
}
```

In our solution we provide also an **ETRY** statement that represents the end of *try-throw-catch* block. This is needed because we include all operations performed by *try-throw-catch* block in a **do...while(0)** block. This solution has two main reasons:

- all the **TRY-ETRY** expression is treated as a single statement
- we can define multiple, not nested, **TRY-ETRY** statements in the same block (reuse of **ex_buf__** variable).

The following represents the compilation and execution steps of the previous example.

```
[nids@vultus ttc]% gcc ex1.c
[nids@vultus ttc]% ./a.out
In Try Statement
Got Exception!
[nids@vultus ttc]%
```

Adding Exceptions

Real exception systems have the possibility to define various *kinds* of exceptions. These kinds are mapped over types and *catch* statements intercept exceptions using these types.

In our solution we cannot define different types for different exceptions. Our solution maps different exception on different return values of function **setjmp**. To do this we use defines like the following:

```
#define FOO_EXCEPTION (1)
```

Now, our **TRY-ETRY** must use a **switch** statement instead of **if** statement. Each **CATCH** statement is no more a simple **else** but it maps over a **case**.

CATCH now become a macro with parameters. Parameter represents the exception kind that is treated in that block. Each **CATCH** statement must also close the previous **case** block (with a **break**.)

```
#include <stdio.h>
#include <setjmp.h>

#define TRY do{ jmp_buf ex_buf__; switch( setjmp(ex_buf__) ){ case 0:
#define CATCH(x) break; case x:
#define ETRY } }while(0)
#define THROW(x) longjmp(ex_buf__, x)

#define FOO_EXCEPTION (1)
#define BAR_EXCEPTION (2)
#define BAZ_EXCEPTION (3)

int
main(int argc, char** argv)
{
   TRY
   {
      printf("In Try Statement\n");
      THROW( BAR_EXCEPTION );
      printf("I do not appear\n");
   }
   CATCH( FOO_EXCEPTION )
   {
      printf("Got Foo!\n");
   }
   CATCH( BAR_EXCEPTION )
   {
      printf("Got Bar!\n");
   }
   CATCH( BAZ_EXCEPTION )
   {
      printf("Got Baz!\n");
   }
   ETRY;

   return 0;
}
```

The following represents the compilation and execution steps of the previous example.

```
[nids@vultus ttc]% gcc ex2.c
[nids@vultus ttc]% ./a.out
In Try Statement
Got Bar!
[nids@vultus ttc]%
```

Adding Finally-Block

A nasty feature of real exception systems is represented by **finally** statement.

Finally statement is really powerful. The block guarded by **finally** statement is executed after the **try** block or any of the **catch** blocks. In real exception systems **finally** block is executed also is **try** or **catch** block execute an exit or return statement. We cannot build over the language a system like this.

Our **FINALLY** statement is executed in three cases:

- after a **TRY** block code (with out exiting)
- after a **CATCH** block code (with out exiting)
- when an exception kind is not a known one

Ok... wait a moment... how can we do it?

In line of principle it is simple: our **FINALLY** block must map the **default** case of the switch. This respect exactly the third event (*when an exception kind is not a known one*).

To respect the other two events it looks simple enough: instead of breaking out of switch, we must jump to the **default** case for instance with:

```
goto default;
```

...bad answer: ANSI-C does not provide you to *goto* to a case of a switch (but it works perfectly with C#).

We must find another solution for this problem. Do you know *Duff Device*? It is a really funny way to use a switch statement to do hand made loop unrolling. It uses a **do{...}while(0)** weaved in a **switch** statement.

We want to use a similar technique. Read the following code:

```
...
switch( /* some expression */ )
{
   case 0: while(1){
           ...
    /* case 0 code here */
           ...
    break;
   case 1:
           ...
           /* case 1 code here */
           ...
           break;
   }
   default:
           ...
           /* default case code here */
           ...
}
...
```

Yes... it is a **while** statement weaved in a **switch** statement. **Break** statements, when invoked, exit from the **while** statement and not from **switch** because **while** is the nearest one.

Now our system is complete: we must weave a **while** in the **switch** statement. Our **TRY-ETRY** now become:

```
#include <stdio.h>
#include <setjmp.h>

#define TRY do{ jmp_buf ex_buf__; switch( setjmp(ex_buf__) ){ case 0: while(1){
#define CATCH(x) break; case x:
#define FINALLY break; } default:
#define ETRY } }while(0)
#define THROW(x) longjmp(ex_buf__, x)

#define FOO_EXCEPTION (1)
#define BAR_EXCEPTION (2)
#define BAZ_EXCEPTION (3)

int
main(int argc, char** argv)
{
   TRY
   {
      printf("In Try Statement\n");
      THROW( BAR_EXCEPTION );
      printf("I do not appear\n");
   }
   CATCH( FOO_EXCEPTION )
   {
      printf("Got Foo!\n");
   }
   CATCH( BAR_EXCEPTION )
   {
      printf("Got Bar!\n");
   }
   CATCH( BAZ_EXCEPTION )
   {
      printf("Got Baz!\n");
   }
   FINALLY
   {
      printf("...et in arcadia Ego\n");
   }
   ETRY;

   return 0;
}
```

The following represents the compilation and execution steps of the previous example.

```
[nids@vultus ttc]% gcc ex3.c
[nids@vultus ttc]% ./a.out
In Try Statement
Got Bar!
...et in arcadia Ego
[nids@vultus ttc]%
```

The Need for the FINALLY Block (Modified on December 9th, 2014)

A reader of this document, made me notice that the code as it is right now — after the introduction of the FINALLY construct — forces the user to introduce always a FINALLY block. E.g.,

```
#include <stdio.h>
#include <setjmp.h>

#define TRY do{ jmp_buf ex_buf__; switch( setjmp(ex_buf__) ){ case 0: while(1){
#define CATCH(x) break; case x:
#define FINALLY break; } default:
#define ETRY } }while(0)
#define THROW(x) longjmp(ex_buf__, x)

#define FOO_EXCEPTION (1)
#define BAR_EXCEPTION (2)
#define BAZ_EXCEPTION (3)

int
main(int argc, char** argv)
{
   TRY
   {
      printf("In Try Statement\n");
      THROW( BAR_EXCEPTION );
      printf("I do not appear\n");
   }
   CATCH( FOO_EXCEPTION )
   {
      printf("Got Foo!\n");
   }
   CATCH( BAR_EXCEPTION )
   {
      printf("Got Bar!\n");
   }
   CATCH( BAZ_EXCEPTION )
   {
      printf("Got Baz!\n");
   }
   ETRY;

   return 0;
}
```

Just removing the FINALLY block results in a compilation error:

```
C:\Dev\Scratch\ttc>gcc -o ttc.exe ttc.c
ttc.c: In function 'main':
ttc.c:42:1: error: expected 'while' at end of input
 }
 ^
ttc.c:42:1: error: expected declaration or statement at end of input

C:\Dev\Scratch\ttc>
```

The same reader, not only found the error, the same person provided the fix! Here it is, in all its simplicity and elegence:

```
...
#define TRY do{ jmp_buf ex_buf__; switch( setjmp(ex_buf__) ){ case 0: while(1){
#define CATCH(x) break; case x:
#define FINALLY break; } default: {
#define ETRY } } }while(0)
#define THROW(x) longjmp(ex_buf__, x)
...
```

With the fix applied, the code works without the FINALLY block

```
C:\Dev\Scratch\ttc>gcc -o ttc.exe ttc.c
```

```
C:\Dev\Scratch\ttc>ttc.exe
In Try Statement
Got Bar!

C:\Dev\Scratch\ttc>
```

And it works also with the FINALLY block

```
C:\Dev\Scratch\ttc>gcc -o ttc.exe ttc.c

C:\Dev\Scratch\ttc>ttc.exe
In Try Statement
Got Bar!
...et in arcadia Ego

C:\Dev\Scratch\ttc>
```

A BAZ... Bug in the FINALLY Implementation (Modified on April 7th, 2015)

Cristiano Pedro da Silva, a Ph.D. student of the Laboratório de Transferência de Calor of the Universidade Federal de Itajubá using this code in hit thesis, pointed out that the code above has a problem. In the case without the FINALLY, if instead of throwing an exception of type BAR_EXCEPTION, the code throws a BAZ_EXCEPTION one, the program ends up in a infinite loop. This because at the closure of the block of the last CATCH, there is no **break** statement!

Fortunately, the fix is easy.

We need to make sure that at the end of the TRY...ETRY we always perform an exit from the infinite loop that we generate at the end of the expansion of the TRY. the code with the fix follows:

```
...
#define TRY do{ jmp_buf ex_buf__; switch( setjmp(ex_buf__) ){ case 0: while(1){
#define CATCH(x) break; case x:
#define FINALLY break; } default: {
#define ETRY break; } } }while(0)
#define THROW(x) longjmp(ex_buf__, x)
...
```

We simply introduced an extra **break** inside the scope defined by the **while(1){**, which means just before all the three closed curly braces at the beginning of the ETRY expansion.

The code has been tested with and without FINALLY and with and without CATCH blocks and it seems to work fine.

The downloadable code and the code in the section below have been updated accordingly.

Complete Code and License Terms (Updated April 7th, 2015)

After the publishing of this document, I received quite a bit of questions about how the code, if it can be possible to use it and what license covers the code.

For the license, I choose the MIT license and the full text of the "library" is reported below:

```
/* Copyright (C) 2009-2015 Francesco Nidito
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this software and associated documentation files (the "Software"), to deal
 * in the Software without restriction, including without limitation the rights to
 * use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies
 * of the Software, and to permit persons to whom the Software is furnished to do
 * so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in all
 * copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 */

#ifndef _TRY_THROW_CATCH_H_
#define _TRY_THROW_CATCH_H_

#include <stdio.h>
#include <setjmp.h>

/* For the full documentation and explanation of the code below, please refer to
 * http://www.di.unipi.it/~nids/docs/longjump_try_trow_catch.html
 */

#define TRY do { jmp_buf ex_buf__; switch( setjmp(ex_buf__) ) { case 0: while(1) {
#define CATCH(x) break; case x:
#define FINALLY break; } default: {
#define ETRY break; } } }while(0)
#define THROW(x) longjmp(ex_buf__, x)

#endif /*!_TRY_THROW_CATCH_H_*/
```

If you prefer, the code can be downloaded from this link .

A Note on ETRY (Updated July 2nd, 2016)

A reader pointed out the **ETRY** is not compliant with the C standard as every identifier or macro starting with capital E and followed by another capital letter or number is actually reserved for the standard <errno.h> include file. Fortunately, ETRY seems not to be used by any current <errno.h> that I was able to check.

On a side note, I will **not** fix this in the document or in the header file that you can download.

Conclusions

In this documents we learned something about **setjmp** and **longjmp**. These two functions are seen, by the largest part of programmers, as esoteric, dangerous and difficult to use.

We, also, learned how to use that functions to provide a simple *try-throw-catch* system. Our system has some limitations but can be useful and it helped us to learn something. This last thing could be considered enough by itself.

To implement the *try-throw-catch* system we stressed the C language weaving together **switch** statement and **while** statement in a non conventional way.

But the must important thing is that we had a lot of **fun**!

For any feedback, suggestion or comment feel free to mail to me at the address: francesco [dot] nidito [at] gmail [dot] com