Implementing exceptions in the C programming language

Adomas Paltanavičius, Student of Vilnius Lyceum, Class 2B

Introduction

This document briefly introduces the reader to the method of implementing generalized exceptions in structural programming language C. Its superset, C++ already has this capability, yet it is hardwired into the language, and is therefore not interesting as a hack.
On the other hand, this project authored by Adomas Paltanavičius, is written without using any exception-specifically compiler's extensions; it's just a hack[1].
This document consists of the listing intermixed with detailed comments for inexperienced readers.

Introducing exceptions

Exceptions are objects, with capability to be raised at certain place in code, and to be handled at another place of code. In a nutshell, it produces alternative to standard signal handling and library error-function methods. Thus it is usually very convenient.

[1] Hack in this case means "Something that is done with big difficulty and surprises much."

Motivation

Some folks say C is dead. This is surely not a true. A quick look around tell the right situation. Many systems requiring highest efficiently (yet portability too) are writing in plain C. On the other hand, if you start learning from C, all other most popular today's languages look similar (and they really are) after that (that is, C++, C# and Java).

License

Source structure

The source is contained in one header file, `exception.h`. This file is prepared for multi-inclusion. Yet it hasn't updated to be shared among few source files efficiently (currently, each file inlines its own copy of static functions), though this is not a hard task.

Source prerequisites

For many reasons (it's not a place to list them here), many companies around the world use GNU[2] Compiler Collection (former GNU C Compiler), `gcc` in short. Also, this header file requires C preprocessor, of course. GNU's one will do.

Source summary

To understand all this, you should be experienced in C, and also know the `setjmp` interface. (Unixers do.)

What is Implemented

- `TRY` — a catcher block.
- `EXCEPT, EXPECT, CATCH` — a handler block.
- `RAISE` — raises an exception.
- `ON` — a handler.

IMPORTANT

[2] GNU means "GNU's Not Unix"; it is a project fighting for freedom.

`TRY` without `EXCEPT` does not work properly. This is impossible to fix, at least I don't see the way. Anyway, if you don't want to break the whole system, I see no other need for such a construct.

Syntax of TRY

The syntax is:

```
try { statements; } except { handler blocks; }
```

Where:

- `statements` — valid C statements;
- `handler blocks` — valid handler blocks (see Syntax of ON);

Syntax of EXCEPT/EXPECT/CATCH

These all three keywords are actually synonyms. They cannot be used alone. The syntax is:

```
except { handler blocks; }
```

Where:

- `handler blocks` — valid handler blocks.

Syntax of RAISE

The syntax is:

```
raise (params)
```

Where:

- `params` — depend on your local implementation. By default, it is a number (int).

Syntax of ON

The syntax is:

```
on (params) { statements; }
```

Where:

- `params` — valid C expression
- `statements` — valid C statements

Main features

- Nested catcher blocks.
- Raising an exception outside catcher block (this results in an unhandled exception.)
- Raising an exception in function called from catcher block.
- Different types for exception structures.
- Reporting file and line where exception was raised/handled.

Notes on memory management

Since customized exceptions usually allocate memory dynamically, it would be nice to have support for freeing function, which would be called after it's not needed anymore:

_EXC_FREE. I. e. after exception is handled. This is listed as to-do item, yet not necessary in the example implementation.

The source

After dealing with various concerns, it is finally the time to list source code. As said before, source code is rich with comments, yet reading it should be a pleasure even for a novice.

Main header file

This section lists the source of header file.

Avoiding multiple includes

This piece of code checks if the header file was already included and aborts in that case.

```
#if defined __EXCEPTION_H__
#  error Header file "exception.h" used twice.  This is the Wrong Thing.
#endif

#define __EXCEPTION_H__
```

Including necessary header files

Some other standard headers are necessary, most notably the `setjmp`.

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

Naming conventions

The default prefix is `` `__exc' `` for functions and `` `__EXC' `` for macros. Underscores mean you shouldn't touch them.

C syntax hacks

Oh, god, I felt like an inventor after writing these.
Clearly, code

```
for (start (), J = 1; J; end (), J = 0)
  code ();
```

Does this:

- executes `start`
- executes `code`
- executes `end`
- ...and terminates.

It also works if nested (think why yourself.) For our purposes we define two macros:

```
/* Execute START, then block after the macro, and finally END. */

#define __EXC_BLOCK(start, end)              \
  for (start, __exc_block_pass = 1;          \
       __exc_block_pass;                     \
       end, __exc_block_pass = 0)

/* Likewise, but START block is empty. */

#define __EXC_END(end)                       \
  for (__exc_block_pass = 1;                 \
       __exc_block_pass;                     \
       end, __exc_block_pass = 0)
```

Getting name of function we’re in

Standard C predefines `__LINE__`, `__FILE__` are not enough for error reporting. We’d like to include function name too. GCC includes things which expand to the name of current function's name.

```
#if (!defined (__GNUC__) || __GNUC__ < 2 || \
     __GNUC_MINOR__ < (defined (__cplusplus) ? 6 : 4))
   /* Otherwise stick to unknown. */
#  define __EXC_FUNCTION               (char *) 0
#else
#  define __EXC_FUNCTION         __PRETTY_FUNCTION__
#endif
```

The exception value

You'll want to make local changes to these. For example, to use your own exception structure. Exception is by default an `int`. Anyway, it can be anything from string to some structure.
Whatever the implementation you choose, type name should be defined as `__EXC_TYPE`. The `RAISE` (and `ON`) macro accepts as many arguments, as it is given, so your function may use all the power of argument passing. Define your function's name as `__EXC_MAKE`. Exceptions are compared in ON macro. You should define comparing function as `__EXC_EQ`.
For example, if you'd like to use strings in place of numbers, use this snippet:

```
#define __EXC_TYPE              char *
#define __EXC_EQ(s1, s2)        (strcasecmp (s1, s2) == 0)
#define __EXC_PRINT(e, stream)  fprintf (stream, "%s", e)
```

The default implementation follows:

```
#ifndef __EXC_TYPE
#  define __EXC_TYPE               int

/* Include the default __EXC_PRINT. */
#  define __EXC_TYPE_DEFAULT
#endif

#ifndef __EXC_MAKE
#  define __EXC_MAKE(code...)      code
#endif

#ifndef __EXC_EQ
#  define __EXC_EQ(c1, c2)         ((c1) == (c2))
#endif
```

There is also an optional exception printer. This is used for debugging purposes only. Define your self's one as `__EXC_PRINT`. Arguments are exception of type `__EXC_TYPE` and stream to print to. Default printer follows:

```
#if !defined (__EXC_PRINT) && defined (__EXC_TYPE_DEFAULT)
#  define __EXC_PRINT(e, stream)             \
     fprintf (stream, "%d", e)
#endif
```

Controlling variables

This part of code contains all variables used for handling exceptions etc. All variables are declared volatile to force non-optimization.
This counter is used by __EXC_BLOCK. It works well even if nested.

```
static volatile int __exc_block_pass;
```

Flag to be set by ON after exception is handled.

```
static volatile int __exc_handled;
```

For indexing every call to `TRY`.

```
static volatile unsigned __exc_tries;
```

These identify the raised exception. File, function, line and the exception itself.

```
static char *__exc_file;
static char *__exc_function;
static unsigned __exc_line;
static __EXC_TYPE __exc_code;
```

Stack is actually a linked list of catcher cells.

```
struct __exc_stack
{
  unsigned num;
  jmp_buf j;
  struct __exc_stack *prev;
};
```

This is the global stack of catchers.

```
static struct __exc_stack *__exc_global;
```

Debugging of exceptions

Code in this section generates many (really) messages telling what is going to happen. In order to work with it successfully, you should define `__EXC_PRINT` (see above.)

```
#ifdef __EXC_DEBUG
#  include <stdarg.h>
#  ifndef __EXC_STREAM
```

I often redirect debugging information to a file, though printing to screen is also useful:

```
#    define __EXC_STREAM                     stdout
#  endif
```

This function prints error message:

```
static void
__exc_debug (char *fmt, ...)
```

```
{
  va_list ap;

  fprintf (__EXC_STREAM, "__EXC: ");
  va_start (ap, fmt);
  vfprintf (__EXC_STREAM, fmt, ap);
  va_end (ap);
  fprintf (__EXC_STREAM, "\n");
}
```

For printing __exc_global.

```
static void
__exc_print_global ()
{
  struct __exc_stack *level = __exc_global;
  unsigned items;

  if (level == NULL)
    {
      fprintf (__EXC_STREAM, "Stack empty\n");
      return;
    }

  fprintf (__EXC_STREAM, "Current stack (from bottom to top):\n");
  for (items = 0; level; level = level->prev)
    {
      fprintf (__EXC_STREAM, "%c ", items == 0 ? '[' : ' ');
      fprintf (__EXC_STREAM, "%u", level->num);
      fprintf (__EXC_STREAM, " %c\n", level->prev ? ' ' : ']');
      items++;
    }

  fprintf (__EXC_STREAM, "Totally %u items.\n", items);
}

#else
#  define __exc_debug(args...)
#  define __exc_print_global()
#endif
```

Function below prints information about exception. Called in debug mode, or when no handler is found:

```
void
__exc_print (FILE *stream, char *file, char *function, unsigned line,
       __EXC_TYPE code)
{
  fprintf (stream, "Exception in file \"%s\", at line %u",
     file, line);
  if (function)
    {
      fprintf (stream, ", in function \"%s\"", function);
    }
  fprintf (stream, ".");

#ifdef __EXC_PRINT
```

```
  fprintf (stream, " Exception: ");
  __EXC_PRINT (code, stream);
#endif
  fprintf (stream, "\n");
}
```

Managing handlers stack

All handlers (their `jmp_buf`s, actually, are pushed onto the handlers list (`__exc_global`). The first function takes exception from stack, putting into `J` (if nonzero). If stack is empty, print error message and exit. Used in `EXCEPT`.

```
static void
__exc_pop (jmp_buf *j)
{
  register struct __exc_stack *stored = __exc_global;

  __exc_debug ("POP () to %p", j);

  if (stored == NULL)
    {
      __exc_debug ("Unhandled exception.");

      fprintf (stderr, "Unhandled exception:\n");
      __exc_print (stderr, __exc_file, __exc_function,
           __exc_line, __exc_code);

      exit (3);
    }

  __exc_global = stored->prev;

  if (j)
    {
```

This assumes that `jmp_buf` is a structure etc. and can be copied rawely. This is true in all architectures supported by GLIBC[3], as far as I know:

```
      memcpy (j, &stored->j, sizeof (jmp_buf));
    }

  __exc_debug ("Popped");
  __exc_print_global ();

  /* While with MALLOC, free.  When using obstacks it is better not to
     free and hold up. */
  free (stored);
}
```

Second function pushes `J` onto the stack, with `RETURNED` as value from `SETJMP`. Returns nonzero, if `RETURNED` is 0. If `RETURNED` is nonzero, returns 0. Used in `TRY`.

[3] GNU C Library, the most core element after the kernel of the Unix system. Here I talk about the GNU's implementation, which is used in Linux etc.

```
static int
__exc_push (jmp_buf *j, int returned)
{
  struct __exc_stack *new;

  __exc_debug ("PUSH (), %p, %d", j, returned);
```

`SETJMP` returns 0 first time, nonzero from `__EXC_RAISE`. Returning false-like value here (0) will enter the else branch (that is, EXCEPT.)

```
  if (returned != 0)
    {
      __exc_debug ("Returning from RAISE");
      return 0;
    }
```

Since this didn't come from RAISE, fine to increase counter:

```
  ++__exc_tries;
  __exc_debug ("This is PUSH () number %u", __exc_tries);
```

Using memcpy here is the best alternative:

```
  new = malloc (sizeof (struct __exc_stack));
  memcpy (&new->j, j, sizeof (jmp_buf));
  new->num = __exc_tries;
  new->prev = __exc_global;
  __exc_global = new;

  __exc_print_global ();

  return 1;
}
```

Exception raising code

This function raises an exception in `FILE` at `LINE`, with code `CODE`. Used in `RAISE`.

```
static void
__exc_raise (char *file, char *function, unsigned line, __EXC_TYPE code)
{
  jmp_buf j;

  __exc_debug ("RAISE ()");
#if defined __EXC_DEBUG
  __exc_print (__EXC_STREAM, file, function, line, code);
#endif
  __exc_file = file;
  __exc_function = function;
  __exc_line = line;
  __exc_code = code;
```

Pop for jumping:

```
  __exc_pop (&j);
  __exc_debug ("Jumping to the handler");
```

`LONGJUMP` to J with nonzero value.

```
    longjmp (j, 1);
  }
```

This function raises it in upper level of catcher blocks.

```
  static void
  __exc_reraise ()
  {
    jmp_buf j;

    __exc_debug ("RERAISE ()");
  #ifdef __EXC_DEBUG
    __exc_print (__EXC_STREAM, __exc_file, __exc_function,
           __exc_line, __exc_code);
  #endif

    __exc_pop (&j);
    longjmp (j, 1);
  }
```

The `ON` implemented

This code implements `ON` on the low-level side.

```
  static int
  __exc_on (char *file, char *function, unsigned line, __EXC_TYPE code)
  {
    __exc_debug ("ON ()");
    __exc_debug ("Trying to handle in file \"%s\", at line %u", file,
  line);
  #ifdef __EXC_DEBUG
    if (function)
      {
        __exc_debug ("In function \"%s\".", function);
      }
  #endif

    if (__exc_handled == 1)
      {
        __exc_debug ("Exception already handled in this level, skip");
        return 0;
      }

    if (__EXC_EQ (code, __exc_code))
      {
        __exc_debug ("This handler FITS");

        __exc_handled = 1;
        return 1;
      }

    __exc_debug ("This handler DOESN'T FIT");
```

In case exception not matched, return zero:

```
    return 0;
  }
```

The magical macros

These macros are really magical. Though the modularity helps to keep them as simple, as possible. First, TRY is defined:

```
#define try                                 \
  if (({jmp_buf __exc_j;                    \
        int __exc_ret;                      \
        __exc_ret = setjmp (__exc_j);       \
        __exc_push (&__exc_j, __exc_ret);})) \
    __EXC_END(__exc_pop (0))
```

Then RAISE:

```
#define raise(code...)                      \
  __exc_raise (__FILE__, __EXC_FUNCTION,    \
         __LINE__, __EXC_MAKE (code))
```

Then EXCEPT:

```
#define except                              \
  else                                      \
    __EXC_BLOCK (__exc_handled = 0,         \
                 ({ if (__exc_handled == 0)  \
                     __exc_reraise (); }))
```

EXPECT and CATCH is an alias for EXCEPT:

```
#define expect                          except
#define catch                           except
```

And finally, ON is defined:

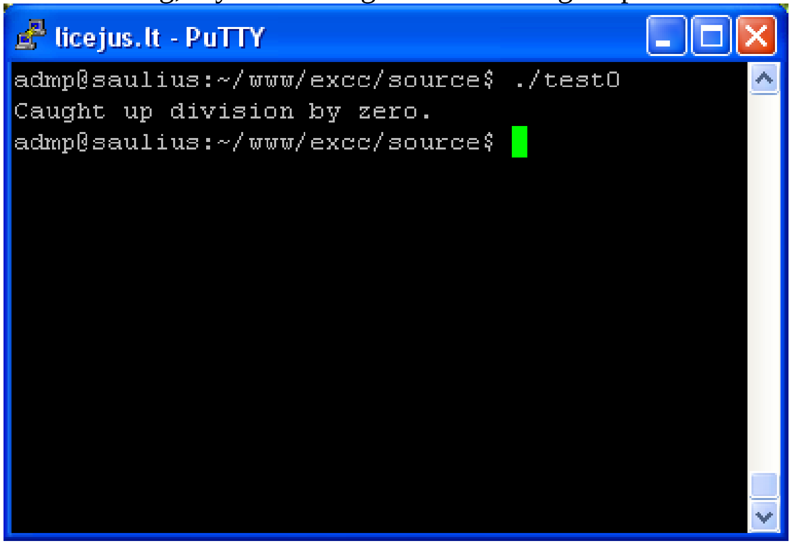
```
#define on(code...)                         \
  if (__exc_on (__FILE__, __EXC_FUNCTION,   \
        __LINE__, __EXC_MAKE (code)))
```

Example program

This section contains example program w/o any comments.

```
#include <stdio.h>
#include "exception.h"

#define DIVISION_BY_ZERO 1001

int divide (int a, int b) {
  if (b == 0) {
    raise (DIVISION_BY_ZERO);
  } else {
    Return a/b;
  }
}

int main (int argc, char **argv)
{
  int i, j;

  try {
    printf (“%d\n”, divide (100, 0));
  } except {
    on (DIVISION_BY_ZERO) {
      printf ("Caught up division by zero.");
      exit (0);
    }
  }
  return 0;
}
```

After running, it you should get the following output:



```
admp@saulius:~/www/excc/source$ ./test0
Caught up division by zero.
admp@saulius:~/www/excc/source$
```