The Exception Handling Library

0.2.1

Generated by Doxygen 1.5.8

Mon Jan 24 01:12:35 2011

Contents

Chapter 1

C Basic Library: Exception Handling Library

Version:

0.2.1

Author:

Jun Woong (woong.jun at gmail.com)

Date:

last modified on 2011-01-24

1.1 Introduction

This document specifies the Exception Handling Library which belongs to the C Basic Library. The basic structure is from David Hanson's book, "C Interfaces and Implementations." I modified the original implementation to make it more appropriate for my other projects, to conform to the C standard and to enhance its readibility; for example a prefix is used more strictly in order to avoid the user namespace pollution.

Since the book explains its design and implementation in a very comprehensive way, not to mention the copyright issues, it is nothing but waste to repeat it here, so I finish this document by giving typical exception-handing constructs with short explanation and emphasis on crucial issues. Some improvements to support C99 is also explained. How to use the facilities is deeply explained in files that define them.

The Exception Handling Library reserves identifiers starting with `except_` and `EXCEPT_`, and imports the Assertion Library.

1.2 How to Use The Library

The followin constrcut shows how a typical TRY-EXCEPT statement looks.

```
EXCEPT_TRY
    S;
EXCEPT_EXCEPT(e1)
    S1;
EXCEPT_EXCEPT(e2)
    S2;
EXCEPT_ELSE
    S3;
EXCEPT_END;
```

`EXCEPT_TRY` starts a TRY-EXCEPT or TRY-FINALLY statement. The statements following `EXCEPT_TRY` (referred to as `S` in this example) are executed, and if an exception is occurred during the execution the control moves to one of EXCEPT clauses with a matching exception or the ELSE clause. The statements `Sn` under the matched EXCEPT clause or ELSE clause are executed and the control moves to the next statement (if any) to `EXCEPT_END`.

```
EXCEPT_TRY
    S
EXCEPT_ELSE
    S1
EXCEPT_END;
```

A constrcut without any EXCEPT clause is useful when catching all exceptions raised during execution of `S` in a uniform way. If other exception handers are established during execution of `S` only uncaught exceptions there move the control to the ELSE clause above. For example, any uncaught exception with no recovery (e.g., assertion failures or memory allocation failures) can be handled as follows in the `main` function.

```
int main(void)
{
    EXCEPT_TRY
        real_main();
    EXCEPT_ELSE
        fprintf(stderr,
                "An internal error occurred with no way to recover.\n"
                "Please report this error to somebody@somewhere.\n\n");
        EXCEPT_RERAISE;
    EXCEPT_END;

    return 0;
}
```

A TRY-FINALLY statement looks like:

```
EXCEPT_TRY
    S;
EXCEPT_FINALLY
    S1;
EXCEPT_END;
```

The statements following EXCEPT_FINALLY are executed regardless of occurrence of an exception, so if a kind of clean-up like closing open files or freeing allocated storages is necessary to be performed unconditionally, S1 is its right place. If the exception caught by a TRY-FINALLY statement needs to be also handled by a TRY-EXCEPT statement EXCEPT_RERAISE raises it again to give the previous handler (if any) a chance to handle it.

Note that each group of the statements, say, S, S1 and so on, constitutes an independent block; opening or closing braces are hidden in EXCEPT_TRY, EXCEPT_EXCEPT, EXCEPT_FINALLY and EXCEPT_END. Therefore variables declared in a block, say, S is not visible to another block, say, S1.

And even if not explicitly specified in Hanson's book, it is possible to construct an exception handling statement which has both EXCEPT and FINALLY clauses, which looks like:

```
EXCEPT_TRY
    S;
EXCEPT_EXCEPT(e)
    Se;
EXCEPT_FIANLLY
    Sf;
EXCEPT_END;
```

Looking into the implementation by combining those macros explains how it works. Finding when it is useful is up to its users.

1.2.1 Some Caveats

Exception handling mechanism given here is implemented using a non-local jump provided by <setjmp.h>. Thus every restriction applied to <setjmp.h> also applies to this library. For example, there is no guarantee that an updated auto variable preserves its last stored value if the update done between setjmp() and longjmp(). For example,

```
{
    int i;

    EXCEPT_TRY
        i++;
        S;
    EXCEPT_EXCEPT(e1)
        S1;
    EXCEPT_TRY;
}
```

If an exception e1 occurs, which moves the control to S1, it is unknown what the value of i is in the EXCEPT clause above. A way to walk around this problem is to declare i as volatile or static. (See the manpage for setjmp() and longjmp().)

At the first blush, this restriction seems too restrictive, but not quite. The restriction applies only to those non-volatile variables with automatic storage duration and belonged to the function containing EXCEPT_TRY (which has setjmp() in it), and only when they are modified between setjmp() (EXCEPT_TRY) and corresponding longjmp() (except_raise()).

One more thing to remember is that the ordinary `return` statement does not work in the statements `S` above because it does not know anything about maintaining the exception stack. Inside `S`, the exception frame has already been pushed to the exception stack. Returning from it without adjusting the stack by popping the current frame spoils the exception handling mechanism, which results in undefined behavior. `EXCEPT_RETURN` is provided for this purpose. It does the same thing as the ordinary `return` statement except that it adjusts the exception stack properly. Also note that `EXCEPT_RETURN` is not necessary in a EXCEPT, ELSE or FINALLY clause since entering those cluases entails popping the current frame from the stack.

In general, it is said that recovery from an erroneous situation gets easier if you have a way to handle it with an exception and its handler. In practice, with the implementation using a non-local jump facility like this library, however, that is not always true. If a program manages resources like allocated memory and open file pointers in a complicated fashion and an exception can be raised at a deeply nested level, it is very likely for you to lose control over them. In addition to it, keeping internal data structures consistent is also a problem. If an exception can be triggered during modifying fileds of an internal data structure, it is never a trivia to guarantee consistency of that. Therefore, an exception handling facility this library provides is in fact best suited for handling in one place various problematic circumstances and then terminating the program's execution almost immediately. If you would like your code to be tolerant to an exceptional case by, for example, making it revert to an inferior but reliable approach, you have to keep these facts in your mind.

1.2.2 Improvements

The diagnostic message printed when an assertion failed changed in C99 to include the name of a function in which it failed. This can be readily attained with a newly introduced predefined identifier `__func__`. To provide more useful information, if an implementation claims to support C99 by defining the macro `__STDC_VERSION__` properly, the library also includes the function name when making up the message output when an uncaught exception detected. For the explanation on `__func__` and `__STDC_VERSION__`, see ISO/IEC 9899:1999.

1.3 Boilerplate Code

To show a bolierplate code, suppose that a module named "mod" defines and may raise exceptions named `mod_exceptfail` and `mod_exceptmem`, and that code invoking the module is expected to install an exception handler for that exception. Now implementing the module "mod" looks in part like:

```
const except_t mod_exceptfail = { "Some operation failed" };
const except_t mod_exceptmem = { "Memory operation failed" };

...

int mod_oper(int arg)
{
    ...
```

```
    if (!p)
        EXCEPT_RAISE(mod_exceptfail);
    else if (p != q)
        EXCEPT_RAISE(mod_exceptmem);
    ...
}
```

where the names of exceptions and the contents of strings used as initializers are up to an user. The string is printed out when the corresponding exception is raised but not caught. By installing an exception handler with a TRY-EXCEPT construct, code that invokes mod_oper() can handle exceptions mod_oper() may raise:

```
EXCEPT_TRY
    result = mod_oper(value);
    ...
EXCEPT_EXCEPT(mod_exceptfail)
    fprintf(stderr, "program: some operation failed; no way to recover\n");
    EXCEPT_RERAISE;
EXCEPT_EXCEPT(mod_exceptmem)
    fprintf(stderr, "program: memory allocation failed; internal buffer used\n");
    ... prepare internal buffer and retry ...
EXCEPT_END
```

Note that exceptions other than `mod_exceptfail` and `mod_exceptmem` are uncaught by this handler and handed to an outer handler if any.

1.4 Future Directions

1.4.1 Stack Traces

The current implementation provides no information about the execution stack of a program when an exception occurred leads it to abnormal termination. This imposes a burden on programmers since they have to track function calls by themselves to pinpoint the problem. Thus, showing stack traces on an uncaught exception would be useful especially when they include enough information like callers' names, calling sites and arguments.

1.5 Contact Me

Visit http://project.woong.org to get the latest version of this library. Only a small portion of my homepage (http://www.woong.org) is maintained in English, thus one who is not good at Korean would have difficulty when navigating most of other pages served in Korean. If you think the information you are looking for is on pages written in Korean you cannot read, do not hesitate to send me an email asking for help.

Any comments about the library are welcomed. If you have a proposal or question on the library just email me, and then I will reply as soon as possible.

1.6 Copyright

Chapter 2

Todo List

Global except_raise Improvements are possible and planned:

- it would be useful to show stack traces when an uncaught exception leads to abortion of a program. The stack traces should include as much information as possible, for example, names of caller functions, calling sites (file name, function name and line number) and arguments.

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

File Documentation

4.1 except.c File Reference

Source for Exception Handling Library (CBL).

```
#include <stddef.h>
#include <setjmp.h>
#include <stdio.h>
#include <stdlib.h>
#include "cbl/assert.h"
#include "except.h"
```

Include dependency graph for except.c:

Functions

- void() except_raise (const except_t *e, const char *file, int line)
  raises an exception and *set its information properly.*

Variables

- except_frame_t * except_stack = NULL

stack for handling nested exceptions.

4.1.1 Detailed Description

Source for Exception Handling Library (CBL).

4.1.2 Function Documentation

4.1.2.1 void() except_raise (const except_t * *e*, const char * *file*, int *line*)

raises an exception *and* set its information properly.

`EXCEPT_RAISE` and `EXCEPT_RERAISE` macros call except_raise() with `__FILE__` and `__LINE__` predefined macros (and `__func__` if C99 supported) for the `file` and `line` parameters. So in general there is little chance to call except_raise() directly in application code.

Possible exceptions: assert_exceptfail

Unchecked errors: foreign data structure given for `e`

Parameters:

← ***e*** exception to raise

← ***file*** file name where exception occurred

← ***func*** function name where exception occurred (if C99 supported)

← ***line*** line number where exception occurred

Returns:

except_raise() cannot return anything

Todo

Improvements are possible and planned:

- it would be useful to show stack traces when an uncaught exception leads to abortion of a program. The stack traces should include as much information as possible, for example, names of caller functions, calling sites (file name, function name and line number) and arguments.

4.2 except.h File Reference

Documentation for Exception Handling Library (CBL).

`#include <setjmp.h>`

Include dependency graph for except.h:

This graph shows which files directly or indirectly include this file:

Data Structures

- struct **except_t**
- struct **except_frame_t**

Defines

- #define EXCEPT_RAISE(e) except_raise(&(e), __FILE__, __LINE__)

  raises exception `e`.

- #define EXCEPT_RERAISE except_raise(except_frame.exception, except_frame.file, except_frame.line)

  raises the exception again that has been raised most recently.

- #define EXCEPT_RETURN switch(except_stack=except_stack → prev, 0) default: return

  returns to the caller function within a TRY-EXCEPT statement.

- #define EXCEPT_TRY

  starts a TRY statement.

- #define EXCEPT_EXCEPT(e)

starts an EXCEPT(e) clause.

- #define EXCEPT_ELSE
  starts an ELSE clause.
- #define EXCEPT_FINALLY
  starts a FINALLY clause.
- #define EXCEPT_END
  ends a TRY-EXCEPT or TRY-FINALLY statement.

Enumerations

- enum { EXCEPT_ENTERED = 0, EXCEPT_RAISED, EXCEPT_HANDLED, EXCEPT_FINALIZED }

Functions

exception raising functions:

- void except_raise (const except_t *, const char *, int)
  raises an exception and *set its information properly.*

Variables

- except_frame_t * except_stack
  stack for handling nested exceptions.

4.2.1 Detailed Description

Documentation for Exception Handling Library (CBL).

Header for Exception Handling Library (CBL).

4.2.2 Define Documentation

4.2.2.1 #define EXCEPT_ELSE

Value:

```
if (except_flag == EXCEPT_ENTERED)        \
                        except_stack = except_stack->prev;    \
                } else {                                      \
                    except_flag = EXCEPT_HANDLED;
```

starts an ELSE clause.

If there is no matched EXCEPT clause for a raised exception the control moves to the statements following the ELSE clause.

4.2.2.2 #define EXCEPT_END

Value:

```
if (except_flag == EXCEPT_ENTERED)        \
                        except_stack = except_stack->prev;    \
                }                                             \
                if (except_flag == EXCEPT_RAISED)             \
                    EXCEPT_RERAISE;                           \
            }
```

ends a TRY-EXCEPT or TRY-FINALLY statement.

If a raised exception is not handled by the current handler, it will be handled by the previous handler if any.

4.2.2.3 #define EXCEPT_EXCEPT(e)

Value:

```
if (except_flag == EXCEPT_ENTERED)          \
                        except_stack = except_stack->prev;      \
                } else if (except_frame.exception == &(e)) {    \
                    except_flag = EXCEPT_HANDLED;
```

starts an EXCEPT(e) clause.

When an exception `e` is raised, its following statements are executed. Finishing them moves the control to the end of the TRY statement.

4.2.2.4 #define EXCEPT_FINALLY

Value:

```
if (except_flag == EXCEPT_ENTERED)        \
                        except_stack = except_stack->prev;    \
                }                                             \
                {                                             \
                    if (except_flag == EXCEPT_ENTERED)        \
                        except_flag = EXCEPT_FINALIZED;
```

starts a FINALLY clause.

It is used to construct a TRY-FINALLY statement, which is useful when some clean-up is necessary before exiting the TRY-FINALLY statement; the statements under the FINALLY clause are executed whether or not an exception occurs. `EXCEPT_RERAISE` macro can be used to hand over the not-yet-handled exception to the previous handler.

Warning:

Remember that, since raising an exception pops up the execption stack, re-raising an exception in a FINALLY clause has the effect to move the control to the outer (previous) handler. Also note that, even if not explicitly specified, a TRY-EXCEPT- FINALLY statement (there are both EXCEPT and FINALLY clauses) is possible and works as expected.

4.2.2.5 #define EXCEPT_RETURN switch(except_stack=except_stack → prev, 0) default: return

returns to the caller function within a TRY-EXCEPT statement.

In order to maintain the stack handling nested exceptions, the ordinary return statement should be avoided in statements (referred to as `S` below; see the explanation for `EXCEPT_TRY`) following `EXCEPT_TRY`. Because return has no idea about the exception frame, retruning without using `EXCEPT_RETURN` from `S` spoils the exception stack. `EXCEPT_RETURN` adjusts the stack properly by popping the current exception frame before returning to the caller.

Warning:

Note that the current exception frame is popped when an exception occurs during execution of `S` and before the control moves to one of EXCEPT, ELSE and FINALLY clauses, which means using `EXCEPT_RETURN` there is not allowed since it affects the previous, not the current, exception frame.

4.2.2.6 #define EXCEPT_TRY

Value:

```
{                                                   \
                volatile int except_flag;                       \
                /* volatile */ except_frame_t except_frame;     \
                except_frame.prev = except_stack;               \
                except_stack = &except_frame;                   \
                except_flag = setjmp(except_frame.env);         \
                if (except_flag == EXCEPT_ENTERED) {
```

starts a TRY statement.

Statements (referred to as `S` hereafter) whose exception is to be handled in EXCEPT, ELSE or FINALLY clause follow.

Warning:

Do not forget using `EXCEPT_RETURN` when returning from `S`. See `EXCEPT_RETURN` for more details. Besides, The TRY-EXCEPT/FINALLY statement uses the non-local jump mechanism provided by <setjmp.h>, which means any restriction applied to <setjmp.h> also applies to the TRY-EXCEPT/FINALLY statement. For example, the standard does not guarantee that an automatic non-volatile

variable belonging to the function which contains setjmp() preserves its last stored value when it is updated between a call to setjmp() and a call to longjmp() with the same `jmp_buf`.

4.2.3 Enumeration Type Documentation

4.2.3.1 anonymous enum

Enumerator:

EXCEPT_ENTERED exception handling started and no exception raised yet

EXCEPT_RAISED exception raised and not handled yet

EXCEPT_HANDLED exception handled

EXCEPT_FINALIZED exception finalized

4.2.4 Function Documentation

4.2.4.1 void except_raise (const except_t * *e*, const char * *file*, int *line*)

raises an exception *and* set its information properly.

`EXCEPT_RAISE` and `EXCEPT_RERAISE` macros call except_raise() with `__FILE__` and `__LINE__` predefined macros (and `__func__` if C99 supported) for the `file` and `line` parameters. So in general there is little chance to call except_raise() directly in application code.

Possible exceptions: assert_exceptfail

Unchecked errors: foreign data structure given for `e`

Parameters:

← ***e*** exception to raise

← ***file*** file name where exception occurred

← ***func*** function name where exception occurred (if C99 supported)

← ***line*** line number where exception occurred

Returns:

except_raise() cannot return anything

Todo

Improvements are possible and planned:

- it would be useful to show stack traces when an uncaught exception leads to abortion of a program. The stack traces should include as much information as possible, for example, names of caller functions, calling sites (file name, function name and line number) and arguments.

Index