Except

Exception handling extensions for C
Manual version $Revision: 2.15 $
$Date: 1995/02/08 19:54:11 $

Bob Glickstein

This manual documents Except, a library package for adding exception handling constructs to C programs.

This software is based on original work by Michael Horowitz and Michael McInerny at the Information Technology Center of Carnegie Mellon University, Pittsburgh, PA 15213. Modifications by Bob Glickstein at Z-Code Software Corp.

Introduction

This manual describes Except, a library package for adding exception handling constructs to C programs.

1 Exception handling overview

Exception handling is a programming technique in which error-handling code is kept separate from code representing the normal flow of a program. Writing code in an exception-handling style results in smaller, more readable, more correct programs.

Error-handling in conventional C programs is cumbersome and error-prone itself! Suppose function *a* calls function *b* which calls function *c* ... which calls function *y* which calls function *z*. Now suppose *z* encounters an error. Typically, it will return a special "error" value to *y*, which will return a (possibly different) special error value to *x* ... which will return a (possibly different) special error value to *b* which will finally inform *a* of the error by returning a (possibly different) special error value. Every function is responsible for checking for errors and communicating that information to its caller, even though the only functions that really *care* about the error are the innermost one (the one that encounters the error) and the one, possibly many call frames higher, which intends to handle the error. If someone in the chain fails to check for an error condition, it will go completely unnoticed and will probably wind up in a software crash later on (because, for instance, the program didn't notice that `malloc` returned `NULL` and tried to use it as a valid address).

Here is a sample sequence of code in which the programmer has been fastidious about checking for error codes:

```
if ((ptr = malloc(n)) == NULL) {
    return (-1);
}
if ((fp = fopen("foo", "r")) == NULL) {
    free(ptr);
    return (-1);
}
if (unlink("bar") != 0) {
    free(ptr);
    fclose(fp);
    return (-1);
}
... other processing ...
```

The code is choked with `if` tests and cleanup cases, which are often highly redundant. The return values that mean "error" often change from one function to the next. Functions which do not produce a computationally meaningful result are nonetheless forced to have a return value to indicate success or failure. And functions which do yield usable values must reserve one such value as its error token, unnecessarily constraining its value range.

In exception handling programming, errors, or *exceptions*, cause an immediate jump[1] from the place where the error happened to the place which has previously assured that it can handle that kind of exception. In the process of finding the handler, it will execute any cleanup code the programmer has associated with aborting the intervening call frames. Different kinds of exceptions can have different names, and a function can say it's able to handle "out of memory" errors but not "no such file" errors (for instance). In any event,

[1] Using `setjmp` and `longjmp`.

it's impossible for an exception to get lost in the shuffle; *someone* will handle it (even if it's the default exception handler that's installed at the beginning of a program).

Here's how the above code fragment might look rewritten in an exception handling style:

```
TRY {
    ptr = e_malloc(n);
    fp  = e_fopen("foo", "r");
    e_unlink("bar");
    ... other processing ...
} EXCEPT(ANY) {
    if (ptr)
        free(ptr);
    if (fp)
        free(fp);
    PROPAGATE();
} ENDTRY;
```

The `TRY` statement encapsulates code where errors might occur. `EXCEPT` clauses contain cleanup code, or *handlers*, to be executed when particular exceptions arise. (Exceptions themselves are indicated with the `RAISE` function.)

The calls to `malloc`, `fopen`, and `unlink` have been replaced here with calls to `e_malloc`, `e_fopen`, and `e_unlink`. These are fictional functions with names that suggest that they work like the originals but when they encounter an error, instead of returning a special value, they raise one of several exceptions whose names are known.

This example says to perform the desired computation, and if anything goes wrong, make sure any allocated memory is released and any opened files are closed.

Incidentally, the definition of `e_malloc` would probably look something like this:

```
void *
e_malloc(n)
    int n;
{
    void *result = malloc(n);

    if (result == NULL)
        RAISE("out of memory", NULL);
    return (result);
}
```

2 The TRY statement

The syntax of the `TRY` statement is:

```
TRY {
    body of normal case
} [ EXCEPT(eid) {
    body of handler for exception case eid
} ] ... [ FINALLY {
    body of cleanup clause
} ] ENDTRY;
```

Square brackets denote optional sections. Each "body" is an arbitrary sequence of C statements (potentially including nested `TRY` statements). The "normal case" body is executed normally, stopping if an exception occurs. (The exception may be raised by a `RAISE` call many call frames down.) There may be any number of `EXCEPT(`*eid*`) { ... }` blocks (called *exception clauses*), including zero. Each *eid* is a null-terminated string naming the exception which the clause handles; this string is called the *exception ID*. An exception ID may be the constant `ANY`; such a clause will handle any raised exception that reaches it. An exception clause is executed when an exception is raised and the clause is an *appropriate handler* as described in Chapter 3 [The RAISE function], page 6. The `FINALLY { ... }` block (called the *finally clause*) is optional. Its body is executed at the end of a `TRY ... ENDTRY` block, whether or not an exception occurred, and whether or not any raised exception was handled by this particular `TRY` block.

Consider the following `TRY` statement:

```
TRY {
    a();
    b();
    c();
} EXCEPT("out of memory") {
    d();
} EXCEPT("file not found") {
    e();
} FINALLY {
    f();
    g();
} ENDTRY;
```

When control reaches this block, the body of the normal case is executed (a call to `a` followed by a call to `b` followed by a call to `c`). If all goes well, control then jumps to the finally clause, where `f` and then `g` are called.

If an "out of memory" exception occurs sometime during the call to `b` (say), then control jumps directly to the handler for the "out of memory" error, and `d` is executed. Similarly, if a "file not found" exception occurs during the call to `b`, then control jumps directly to the "file not found" handler, executing `e`. Execution in both exception cases resumes with `f` and `g` in the finally clause, then continues after the `TRY` block.

If an exception other than "out of memory" and "file not found" occurs during `a`, `b`, or `c`, then the finally clause is executed and control jumps to a call frame containing a handler for the raised exception. No statements after the `ENDTRY` are executed.

If there is no `EXCEPT` block in any pending call frame that will handle the raised exception, then the *uncaught exception handler* is invoked (see Chapter 6 [Miscellaneous functions], page 11).

It is sometimes the case that a section of exception-handling code needs to apply to all raised exceptions even when there is additional code that needs to apply to specific exceptions. This won't suffice:

```
TRY {
    ...
} EXCEPT(specific) {
    code for a specific exception
} EXCEPT(ANY) {
    code for all other exceptions
} ENDTRY;
```

because the "code for all other exceptions" won't be executed in the case of the *specific* exception. Use this instead:

```
TRY {
    ...
} EXCEPT(ANY) {
    if (!strcmp(except_GetRaisedException(), specific)) {
        code for a specific exception
    }
    code for all exceptions
} ENDTRY;
```

Generally speaking, a program will contain many more `TRY ... FINALLY ... ENDTRY` blocks than `TRY ... EXCEPT ... ENDTRY` blocks. This is because the exception handler for a particular kind of error only needs to be in a small number of places, while cleanup code for code blocks where failures can occur is much more prevalent. There is no reason to write a `TRY` block with no `EXCEPT` clauses and no `FINALLY` clause. This is a common idiom:

```
p = e_malloc(n);        /* or fp = e_fopen(...), or ... */
TRY {
    code that uses the value of p
} FINALLY {
    free(p);            /* or fclose(fp), or ... */
} ENDTRY;
```

This ensures that `p` is always freed when it's no longer needed, even if an exception occurs deep inside the "code that uses the value of `p`". Note that the call to `e_malloc` is outside the `TRY` block because if it fails, the `FINALLY` clause needn't be executed (no memory will have been allocated), and the "out of memory" exception raised by `e_malloc` will abort this frame and unwind the stack to search for an appropriate handler.

3 The RAISE function

The `RAISE` function raises a named exception and associates a piece of data with it.

`RAISE` (*const char* *`eid`, *void* *`edata`) [Function]

Raises exception *eid* (a null-terminated string). This function does not return. Instead, it causes a `longjmp` to the nearest appropriate enclosing handler for the exception. "The nearest appropriate" handler is the innermost pending `TRY` statement containing either a `FINALLY` clause or an `EXCEPT` clause matching *eid*. An `EXCEPT` clause "matches" *eid* if the `EXCEPT` clause's name compares the same as *eid* (using `strcmp`), or if it is an `EXCEPT(ANY)` clause.

If the nearest appropriate handler is a `FINALLY` clause, its body is executed and then the search for an appropriate handler continues as above. If the nearest appropriate handler is a matching `EXCEPT` clause, its body is executed, followed by the `FINALLY` clause (if present) of the same `TRY` statement, and then control proceeds normally following the `TRY` block.

Any handler for the raised exception may call the function `except_GetExceptionValue` to get the value of *edata*, which is an arbitrary datum that is associated with the exception.

If no appropriate handler for exception *eid* exists, the uncaught exception handler is called (see Chapter 6 [Miscellaneous functions], page 11).

4 Non-local exits

This implementation of exceptions for C is entirely based on macro definitions and library functions; i.e., it is not a part of the language *per se*. As such, it has one important limitation not found in languages with intrinsic exception-handling: `TRY` statements that can be aborted by non-local exits must be written specially.

A *non-local exit* in C is a point in the program where control does not flow sequentially from one statement to the next.[1] In C, the following keywords all cause non-local exits:

- `break`
- `continue`
- `goto`
- `return`

To keep its internal state consistent, Except expects that a `TRY` block will always reach its `ENDTRY` (or will be exited via a raised exception). *It is an error to write a* `break`*,* `continue`*, or* `return` *statement inside a* `TRY` *block if it will cause control to leave the* `TRY` *block*! These control constructs can be replaced with the following:

- `EXC_BREAK`
- `EXC_CONTINUE`
- `EXC_RETURN`
- `EXC_RETURNVAL(t, v)`

Note that there is no `EXC_` equivalent for `goto`; this means it is an error to use `goto` to leave a `TRY` block. The `EXC_RETURN` form is used when returning from `void` functions that do not return a value. The `EXC_RETURNVAL` form is used when returning from functions that do need to return a value. It takes two arguments: t is the type of the value and v is the value itself.[2]

Note that a `break` statement in a `switch` statement does *not* get replaced by `EXC_BREAK`, but a `break` statement that exits from a loop construct does.

Each of these non-local exit keywords exits a particular kind of block non-locally. `return` exits an entire function, while `break` and `continue` each exits the innermost enclosing loop containing them. Whichever kind of block has had a non-local exit replaced with an `EXC_` form must *itself* be rewritten as follows:

- A function containing an `EXC_RETURN` or `EXC_RETURNVAL` must have its outermost braces surrounded with `EXC_BEGIN` and `EXC_END`, like so:

---

[1] Note that when control reaches the end of a loop construct such as a `while` loop and jumps back to the beginning, a non-local exit is *not* considered to have taken place.

[2] `EXC_RETURNVAL` is a macro that creates a temporary variable whose type is t, which must be a type descriptor that can be placed entirely to the left of a variable name in a variable declaration. If v has a more complicated type, use `typedef` to create a simple name for it. For instance, if v is a pointer to a function returning `int`, its declaration would normally be written as

```
int (*v)();
```

but here v does not lie entirely to the right of its type descriptor. Use something like

```
typedef int (*intfn_t)();
```

and then call `EXC_RETURNVAL(intfn_t, v)`.

```
char *
somefunc(a)
    char *a;
EXC_BEGIN
{
    ...
    TRY {
        ...
        EXC_RETURNVAL(char *, index(foo, bar));
        ...
    } ENDTRY;
    ...
} EXC_END;
```

However, `EXC_RETURN` and `EXC_RETURNVAL` only need to be used when returning from within a `TRY` block. It's okay to use an ordinary `return` when returning from a `EXC_BEGIN-EXC_END` wrapped function if not returning from within a `TRY` block.

- A loop containing an `EXC_BREAK` or `EXC_CONTINUE` must be rewritten to replace the `do`, `for`, or `while` with `EXC_DO`, `EXC_FOR`, or `EXC_WHILE`, respectively. Also, the body of the loop (whether a single statement with no surrounding braces, or a compound statement with surrounding braces) must be followed by an `EXC_END`. Only the loop affected by the non-local exit need be rewritten; if such a loop is contained within another loop, the outer loop need not be rewritten. Example: this function

```
char *
newfunc(a)
    char *a;
{
    ...
    for (i = 0; i < 17; ++i) {
        ...
        TRY {
            ...
            if (i == j)
                break;
            ...
        } ENDTRY;
        ...
    }
    ...
}
```

should be rewritten as

```
char *
newfunc(a)
    char *a;
{
    ...
    EXC_FOR (i = 0; i < 17; ++i) {
        ...
        TRY {
            ...
            if (i == j)
                EXC_BREAK;
            ...
        } ENDTRY;
        ...
    } EXC_END;
    ...
}
```

Only loops containing `TRY` statements containing non-local exits need be rewritten. A `TRY` statement which totally encloses a loop which contains a non-local exit does not need to be rewritten, since the non-local exit will not cause control to leave the `TRY` block.

Warning: All of the non-local exit constructs will bypass the `FINALLY` clause of any `TRY` block which they exit!

5 Signal handling

A variation of the `TRY` statement allows the programmer to treat Unix signals as exceptions. The `TRYSIG` statement is used to enclose a block containing exception handlers for certain signals. The syntax of a `TRYSIG` block is parallel to the syntax for a `TRY` block:

```
TRYSIG((sig1, sig2, ..., 0)) {
    body of normal case
} [ EXCEPTSIG(sig1) {
    body of handler for signal number sig1
} ] ... [ FINALLYSIG {
    body of cleanup clause
} ] ENDTRYSIG;
```

Square brackets denote optional sections, and there may be any number of `EXCEPTSIG(`*sig*`) { ... }` blocks. Note the double-parentheses required in `TRYSIG`'s parameter list, and also note the required terminating `0`.

Upon entry to a `TRYSIG` block, a signal handler is installed for each of the listed signal numbers which simply performs a `RAISE(strsignal(`*sig*`), NULL)` (where `strsignal(`*sig*`)` yields a string naming the signal numbered *sig*). This exception can be caught by a `EXCEPTSIG(`*sig*`)` clause. Upon (any kind of) exit from the `TRYSIG` block, the previously-installed handlers for the listed signals are restored.

Note that `EXCEPTSIG(`*sig*`)` is exactly equivalent to `EXCEPT(strsignal(`*sig*`))` and, in fact, ordinary `EXCEPT` clauses can be freely mixed among `EXCEPTSIG` clauses. However, if `EXCEPTSIG` is used, then `TRYSIG`, `FINALLYSIG`, and `ENDTRYSIG` must be used instead of their non-signal-related counterparts.

Finally, note that an `EXCEPT(ANY)` clause will catch all signal-generated exceptions as well as ordinary exceptions.

6 Miscellaneous functions

`ASSERT` (`e`, *const char* *`eid`, *void* *`edata`) [Macro]

Evaluates the arbitrary expression *e*. If it is zero, `ASSERT` raises the exception named *eid* (a string), with associated datum *edata* (see Chapter 3 [The RAISE function], page 6). If *e* is non-zero, `ASSERT` yields its value and takes no further action.

`PROPAGATE` () [Function]

Inside an `EXCEPT` clause, re-raises the same exception (i.e., the one being handled). Control immediately leaves the pending `TRY` block and searches for the next appropriate handler as described in Chapter 3 [The RAISE function], page 6. **Warning**: If `PROPAGATE` is called inside an `EXCEPT` clause, and there is an associated `FINALLY` clause awaiting execution, the `FINALLY` clause will be bypassed! A workaround for this unfortunate state of affairs is this: Instead of writing

```
TRY {
    ...
} EXCEPT("some error") {
    ...
    PROPAGATE();
    ...
} FINALLY {
    cleanup code
} ENDTRY;
```

(in which the "cleanup code" will never be executed if `PROPAGATE` is reached), write

```
TRY {
    TRY {
        ...
    } EXCEPT("some error") {
        ...
        PROPAGATE();
        ...
    } ENDTRY;
} FINALLY {
    cleanup code
} ENDTRY;
```

In this version, the call to `PROPAGATE` immediately aborts the inner `TRY`. In searching for a new handler for the raised exception, the outer `TRY` is encountered and its `FINALLY` clause is executed.

`const char * except_GetRaisedException` () [Function]

Returns the name (exception ID) of the latest exception raised.

`void except_SetExceptionValue` (*void* **v*) [Function]

Associates arbitrary datum *v* with the pending exception. *v* becomes the value returned by `except_GetExceptionValue`.

`void * except_GetExceptionValue ()` [Function]
Returns the datum that was associated with a raised exception by `RAISE` or a subsequent `except_SetExceptionValue`.

`void except_SetUncaughtExceptionHandler (`*void (*func*`)())` [Function]
Causes *func* to become the function that handles uncaught exceptions (exceptions which, when raised, can find no appropriate handler after checking the outermost enclosing `TRY` block). *func* is a void function of no arguments, and can access the name and data of the pending exception using the functions `except_GetRaisedException` and `except_GetExceptionValue`. It is expected that *func* doesn't return normally (i.e., it does a `longjmp` or it exits the application or somesuch); if it does return normally, control will resume after the outermost pending `TRY` block, or immediately following the pending call to `RAISE` if there are no pending `TRY` blocks.

The default uncaught exception handler generates a core-dumping signal that is guaranteed not to be caught.

`(void (*) ()) except_GetUncaughtExceptionHandler ()` [Function]
Returns the function that handles uncaught exceptions.

`DEFINE_EXCEPTION (`*name*, *string*`)` [Macro]
Define a string-valued variable and an exception identifier to serve as its value. The variable name should be used in `RAISE` and `EXCEPT` constructs rather than the string constant, to guard against typographical errors in duplicating the exception identifier. This macro expands to:

```
const char name[] = string
```

and must be followed by a semi-colon.

`DECLARE_EXCEPTION (`*name*`)` [Macro]
Declare the name of a variable whose string value is intended to be used as an exception identifier. This macro expands to:

```
extern const char name[]
```

and must be followed by a semi-colon.

7 Compiling

To use Except in a C program, include the header file `except.h` in any module that uses Except macros or functions. Link your object files with `libexcept.a`.

Function index

Table of Contents