40

Implementing Exceptions in C

by Eric S. Roberts

March 21, 1989

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Implementing Exceptions in C

Eric S. Roberts

March 21, 1989

Author's abstract

Traditionally, C programmers have used specially designated return codes to indicate exception conditions arising during program execution. More modern languages offer alternative mechanisms that integrate exception handling into the control structure. This approach has several advantages over the use of return codes: it increases the likelihood that programming errors will be detected, makes it easier to structure the specification of an abstraction, and improves the readability of the implementation by providing better syntactic separation between handling of conventional and exceptional cases. This paper describes a set of language extensions to support exception handling in C, and a preprocessor-based implementation of those extensions that demonstrates both the feasibility and the portability of this approach.

Eric S. Roberts

1. Introduction

In designing an abstraction, it is important to define how it behaves, not only under "normal" conditions, but also when unusual or exceptional conditions occur. For example, a file handling package that implements functions like **open**, **read**, and **write** must define the semantics of those routines, not only when all is well, but also in response to such conditions as file-not-found and data-error. Some conditions represent errors; others represent events that are expected to happen, but that are nonetheless distinct from the "main-line" behavior of the abstraction, such as end-of-file during a call to **read**. Collectively, both types of conditions are called *exceptions*.

Historically, C programmers adhere to the Unix tradition of reporting exceptions via specially designated return codes. For example, the **fopen** function in the standard I/O library [Kernighan78] returns **NULL** if the requested file cannot be opened. Similarly, the **getc** function indicates the end-of-file condition by returning the special value **EOF**. This approach, however, has several deficiencies and has been described as "perhaps the most primitive form of exceptional condition handling mechanism" [Levin77]. Various authors [Levin77, Goodenough75, Yemini85] have described higher-level constructs for reporting and handling exceptions that address these shortcomings.

This paper describes a general exception facility for use with C. This facility provides significantly increased functionality and makes it possible to separate, syntactically as well as logically, the specification of the usual behavior of an abstraction from the exception conditions that might arise. It is based on the exception-handling mechanisms in Modula-2+ and Modula-3 [Rovner85, Cardelli88] and is historically related to similar mechanisms in Ada and CLU [ADA82, Liskov84]. The paradigm itself is not new; the main contribution of this paper lies in demonstrating that this mechanism can be implemented in C without requiring language changes or otherwise sacrificing portability.

Although its genesis is independent, this work is also similar to an earlier mechanism reported by Eric Allman and David Been at the 1985 USENIX conference [Allman85]. It shares the overall goal of providing a portable implementation of exceptions in C; moreover, both packages use the C preprocessor to achieve that portability, although the earlier package requires some system-specific assembly code. The work described in this paper makes additional contributions in four areas: (1) no assembly code is required, (2) the syntactic presentation emphasizes the connection between exception-handling code and the program region over which that exception-handling code is active, (3) exception handlers can access local variables in the scope of the exception body, and (4) this facility includes a mechanism for specifying "finalization" actions (see section 3.3).

In this paper, section 2 outlines the deficiencies of the return-code mechanism traditionally used in C, and section 3 introduces an alternative paradigm. Section 4 provides a syntactic definition of the mechanism. Section 5 discusses the implementation of the exception facility. To achieve portability, this implementation is based on the C preprocessor and makes minimal assumptions about the operating environment. In a particular environment, considerable improvements in efficiency can be obtained by incorporating this syntax directly into the compiler, as discussed in section 5.2. As long as these extended compilers remain compatible with the preprocessor-based implementation, programmers can rely on that preprocessor implementation to maintain portability.

2. Weaknesses in the return-code paradigm

As a general technique, return codes have several deficiencies. First, it is often difficult to find an appropriate return code to indicate an exception when the function also returns data. In some cases, the need to find such a value leads to a counterintuitive weakening of the type system. One would expect, for example, that a function named **getc** would return a value of type **char**; to accommodate the end-of-file code, **getc** is defined to return an **int**.

Second, using a single return code to indicate an exception often means that any additional data providing the specific details that gave rise to the exception must be passed outside of the return code mechanism. In the standard Unix libraries, this is usually done with the special variable **errno**. Unfortunately, this strategy is not reentrant and complicates the design of packages that support multiple threads of control in a single address space.

Third, and most importantly, indicating exception conditions via return codes makes them too easy for programmers to ignore. This is particularly true in the development of hierarchical software packages, since each layer must check for the condition explicitly, deciding whether to handle it internally or to propagate it back to its caller. If a layer in the abstraction hierarchy fails to check for the condition, any trace of the exception is lost. This problem lies at the root of many hard-to-detect bugs in traditional C code.

3. Exceptions viewed as control structure

An alternative paradigm is to consider exception handling as part of the control structure. When an exception condition is detected, the program indicates that event by transferring control to a dynamically enclosing section of code specifically designated to react to that condition. The transfer of control is called *raising an exception*, and the code that detects and responds to the condition is called an *exception handler.*

For concreteness, imagine that a new file management package has been designed using this control-based exception paradigm. A client might then open the file **test.dat** using the following code:

```
TRY
    file = newopen("test.dat", "r");
EXCEPT(OpenError)
    printf("Cannot open the file test.dat\n");
ENDTRY;
```

The statement forms themselves are described in the the next section, but the example illustrates the mechanism. If the file **test.dat** exists and is readable, **newopen** returns normally and passes back a handle which is assigned to **file**. If, however, some problem is detected, the implementation of **newopen** can raise the **OpenError** exception. This causes control to be passed immediately to the **printf** in the **EXCEPT** handler.

Note that the statement which raises the **OpenError** exception can be nested arbitrarily deeply within the implementation of the file management package. When the exception is raised, the control stack is unwound to the level of the exception handler, and control proceeds from there. Finding no appropriate handler when an exception is raised constitutes a fatal error. This means that conditions that may safely be ignored must be specified explicitly in the code, thereby reducing the likelihood of error through a careless oversight.

As described below, the package offers considerably more functionality than is illustrated by the simple example above. For example, a **TRY–EXCEPT** statement can specify several independent exceptions and is not limited to a single handler. Each exception has its own handler body, so that the response to each condition is clearly indicated in the code. When an exception is raised, it is possible to pass additional data back to the handler. Thus, in the example above, the file handling package could pass an indication of the error type, so that the client could differentiate conditions like nonexistent file, protection violations, and so forth. This is handled in a reentrant fashion without requiring the use of a global variable such as **errno**. The package also provides a mechanism for specifying a "finalization" action for those cases in which an intervening software layer needs to ensure that some actions are performed even when an exception causes control to pass through this layer. This is discussed in more detail below under the description of the **TRY–FINALLY** statement.

4. Syntactic forms

This section describes the new "statement forms" that are defined as preprocessor macros in order to implement the exception facility in C. To use the package, it is necessary to read in the exception header file by including the line

```
#include "exception.h"
```

4.1 Declaring exceptions

In the exception package, a new exception is declared by using the type definition **exception**, as in

exception *name*;

Like any other variable declaration in C, the scope of an exception may be restricted by the use of the keyword **static**, or imported into another module by using **extern**. An exception is uniquely identified by its address; its value and the concrete type actually used are irrelevant to the operation of the package. In a typical implementation, the concrete type used for an exception will be a **struct** to ensure that **lint** will detect type errors in the use of exceptions.

4.2 TRY–EXCEPT statements

Once the exceptions are declared, a **TRY–EXCEPT** statement is used to associate one or more exception handlers with a sequence of statements. The **TRY–EXCEPT** statement has the form

```
TRY
    statements
EXCEPT(name-1)
    statements
EXCEPT(name-2)
    statements
...
ENDTRY
```

where any number of **EXCEPT** clauses may appear (up to an implementation-determined maximum).

The semantics of the **TRY–EXCEPT** statement are as follows. First, the statements associated with the **TRY** body are evaluated. If no exception conditions are encountered before control reaches the end of that statement sequence, the exception scope is exited and control passes to the end of the entire block. If, however, any of these statements (or any statement of a procedure dynamically nested within this block) raises an exception, control is passed immediately to the statements in the **TRY–EXCEPT** handler with a matching exception name.

Exceptions are raised by calling

RAISE(*name*, *value*);

where *name* is a declared exception and *value* is an **int** to be communicated back to the handler scope. When the **RAISE** statement is encountered, the dynamic stack of **TRY** scopes is searched for the innermost **TRY–EXCEPT** that declares a handler for this exception, or any handler for the predeclared exception **ANY**, which matches any exception. If none is found, an error exit occurs. If an appropriate handler is found, control returns to this stack context, and the statements associated with the appropriate handler are executed. These statements are executed outside of the local exception scope, so that any **RAISE** statements inside a handler propagate the exception back to the next higher level.

In a handler, the argument value passed to **RAISE** may be retrieved by using the name **exception_value**, which is automatically declared as an **int** in the scope of the exception handler. In most cases, this value will not be needed. Even so, the value argument must still be specified in **RAISE**, since **RAISE** is implemented as a macro and not as a procedure.

4.3 TRY–FINALLY statements

A second use of the **TRY** facility is to associate finalization code with a statement sequence in order to ensure that this code is executed even if the statement body terminates abnormally through an exception condition. This is accomplished using the **TRY–FINALLY** statement, which has the form

```
TRY
    statements
FINALLY
    statements
ENDTRY
```

In this form, the standard operation is to execute the **TRY** statement body followed by the statements in the **FINALLY** clause. The **FINALLY** body is also executed if the statements in the **TRY** body generate an exception that would pass control through this scope. When this occurs, the **FINALLY** clause is executed, and the exception is again raised, so that it is eventually caught by the appropriate handler.

For example, assume that **acquire(res)** and **release(res)** acquire and release some resource which requires exclusive access. The code fragment

```
acquire(res);
TRY
    ... code for accessing the resource ...
FINALLY
    release(res);
ENDTRY
```

guarantees that the resource is released, even if the accessing code raises an exception. It is the programmer's responsibility to ensure that this does not violate data integrity conditions in the data structures that manage the resource.

Note that explicit transfers of control out of a **TRY–FINALLY** statement cannot be intercepted without changing the compiler; only exceptions are correctly handled here. For example, it would be very convenient to write

```
acquire(res);
TRY
    return (res->data);
FINALLY
    release(res);
ENDTRY
```

in the hopes that returning from the **TRY–FINALLY** body would invoke the **FINALLY** clause (this is indeed the semantics in Modula-2+ and Modula-3), but this is not feasible in a portable implementation. Instead, it is necessary to assign the result to a temporary variable and write the **return** statement outside the **TRY** body.

5. Implementation of the exception facility

As noted in the introduction, the main purpose of this implementation is to demonstrate that writing a portable exception-handling facility in C is feasible. This implementation uses the C preprocessor to replace the syntactic forms used for exceptions with the actual code required to register and respond to the exception condition. The code is reproduced in full at the end of this paper, but it is best illustrated using a simple example.

5.1 Example of preprocessor expansion

Consider the simple **TRY–EXCEPT** example below:

```
#include "exception.h"

exception e;

Test()
{
        TRY
              Body();
        EXCEPT(e)
              Handler();
        ENDTRY
}
```

The expanded form of the **Test** procedure is given at the top of page 6.

The expanded **TRY** body begins by declaring a local context block **_ctx** in the stack frame for the procedure, initializing the appropriate fields, and then linking this block into a chain of active exceptions. On the first pass, the variable _es is set to **ES_Initialize**. This variable can take on two other values: **ES_EvalBody** in the second iteration of the **while** loop, and **ES_Exception** if control returns to the **setjmp** via a call to **RAISE**.

The body of the **TRY** is not executed on the first iteration of the loop, which simply initializes the array of exceptions which are handled by this **TRY–EXCEPT**. The second pass evaluates the body. If **RAISE** is called, this is translated into a call on **_Raise_Exception**, which searches through the exception stack to find the appropriate handler and then uses **longjmp** to return to this context. When this occurs, _es is set to **ES_Exception** and the conditionals in the expanded **EXCEPT** clauses select the correct handler.

5.2 Preprocessor-based implementation *vs.* compiler support

The preprocessor-based implementation illustrated above does not generate particularly efficient code, mostly because the implementation relies on macro expansion and has no opportunity to perform context-sensitive expansion or to reorder the code. A compiler could do much better.

Implementing this facility directly within the compiler is certainly an option. A compiler can recognize this syntax and generate considerably more efficient code. In particular, a compiler can reduce significantly the overhead required to enter an exception scope by moving much of that work into the code for raising the exception; since most applications will use the exception for the less frequent cases, this tradeoff improves overall performance. By itself, a strategy that involves compiler changes sacrifices portability. However, as long as a preprocessor-based implementation exists, it is certainly reasonable to extend specific compilers to provide this facility with a much lower overhead.

```
Test()
{
        {
                context_block _ctx;
                int _es = ES_Initialize;
                _ctx.nx = 0;
                _ctx.link = NULL;
                _ctx.finally = 0;
                _ctx.link = exceptionStack;
                exceptionStack = &_ctx;
                if (setjmp(_ctx.jmp) != 0) _es = ES_Exception;
                while (1) {
                        if (_es == ES_EvalBody) {
                                Body();
                                if (_es == ES_EvalBody) exceptionStack = _ctx.link;
                                break;
                        }
                        if (_es == ES_Initialize) {
                                if (_ctx.nx >= MaxExceptionsPerScope)
                                        exit(ETooManyExceptClauses);
                                _ctx.array[_ctx.nx++] = &e;
                        } else if (_ctx.id == &e || &e == &ANY) {
                                int exception_value = _ctx.value;
                                exceptionStack = _ctx.link;
                                Handler();
                                if (_ctx.finally && _es == ES_Exception)
                                        _RaiseException(_ctx.id, _ctx.value);
                                break;
                        }
                        _es = ES_EvalBody;
                }
        }
}
```

Figure 1
Expansion of Test procedure

The overhead cost incurred by the preprocessor implementation is illustrated in the expanded form of the **Test** procedure given above. Much of the overhead arises because the preprocessor cannot take contextual information into account. For example, given a **TRY** statement with two **EXCEPT** clauses, a compiler would generate different code in each position, since the operations required at the end of the **TRY** body differ from those required at the end of the first **EXCEPT** clause. Unfortunately, the preprocessor cannot do this. All it can do is expand the **EXCEPT** macro in a context-independent way. This gives rise to redundant tests, like the

```
if (_es == ES_EvalBody)
```

conditional after the main body. If the code reaches here, the condition must be true, although it will be false when the same macro is expanded at the end of an **EXCEPT** clause. Integrating this syntax into the compiler would make it possible to eliminate such redundancy.

Another advantage of the compiler-based implementation is that it can certainly provide better checking for syntactic errors. Like most macro-based extensions, relying on the C preprocessor means that some syntactic errors will be able to pass through without detection and that any errors that are detected will be reported in the context of the expanded forms.

5.3 Implementation dependencies

The implementation depends on the existence of the library routines **setjmp** and **longjmp** to effect the actual transfer of control. These routines are supported in many non-Unix implementations of C, so that this dependency should not significantly reduce the portability of the package. In particular, no assumptions are made about the internal structure of a **jmp_buf** as defined in the header file **setjmp.h**.

Even so, it is important to be aware that some systems impose implementation-dependent restrictions on the use of **setjmp** and **longjmp**, particularly when the compiler tries to be too clever. If such restrictions exist, this exception package may not be usable. We consider such "implementations" of **setjmp** and **longjmp** to be buggy and do not believe that compiler and language changes should be introduced to accommodate these environments.

5.4 Implementation on a multiprocessor

The implementation described in this paper uses the global variable **exceptionStack** as a pointer to a chain of active exception blocks. This is appropriate in a conventional Unix envrionmnent, but would fail in a concurrent environment in which multiple independent lightweight processes (threads) share a single address space [Rovner85, Cooper88, Birrell89]. If threads are supported by the operating system, it is necessary to maintain separate copies of this pointer for each thread. If the implementation of concurrency provides for a thread-specific data area, the chain pointer would presumably be placed there. Otherwise, it will be necessary to simulate this by, for example, hashing on the thread id.

5.5 Other tradeoff considerations in the implementation

In this implementation, the **context_block** structure uses an array of exceptions rather than a linked list to avoid the cost of dynamic allocation when exceptions are registered. This has the negative effect of placing a fixed upper bound on the number of exceptions per scope, but this is not likely to become a serious problem in practice.

In the code for **_Raise_Exception**, two loops are executed: one to determine whether any exception handler exists, and a second to execute the **FINALLY** clauses. A single loop could be used here, but the two-loop strategy has the advantage that the **EUnhandledException** error occurs at the original stack context, making it easier to find unhandled exceptions in the debugger.

6. Conclusions

This package demonstrates that an exception-handling facility can be implemented in C without requiring extensions to the compiler or non-standard assumptions about the operating environment. This means that programs written to use exceptions will be portable to a wide variety of architectures, operating systems, and compilers. If greater efficiency is required, this syntax can be integrated into the compiler, maintaining portability through the preprocessor-based implementation.

7. Acknowledgments

I am indebted to Garret Swart for reading through several drafts of this paper and for providing the initial push that got me to write a report on this work.

8. References

[ADA82] United States Department of Defense. *Reference Manual for the Ada Programming Language*. AdaTEC, July 1982.

[Allman85] Eric Allman and David Been. "An Exception Handler for C," *Proceedings of the 1985 Summer USENIX Conference*. Portland, Oregon, 1985.

[Birrell89] Andrew Birrell. "An introduction to programming with threads," Research Report #35, Systems Research Center, January 6, 1989.

[Cardelli88] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow and Greg Nelson. "Modula-3 report," Research Report #31, Systems Research Center, August 25, 1988.

[Cooper88] Eric Cooper and Richard Draves. "C threads," Technical Report CMU-CS-88-154, Department of Computer Science, Carnegie Mellon University, June 1988.

[Goodenough75] John B. Goodenough. "Exception handling: issues and a proposed notation," *Communications of the ACM*. Vol. 18, no. 12, December 1975.

[Kernighan78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1978.

[Levin77] Roy Levin. "Program structures for exceptional condition handling," Department of Computer Science Technical Report, Carnegie-Mellon University, June 1977.

[Liskov84] Barbara Liskov, et al. *CLU Reference Manual*. Springer Verlag, 1984.

[Rovner85] Paul Rovner, Roy Levin and John Wick. "On extending Modula-2 for building large, integrated systems," Research Report #3, Systems Research Center, January 11, 1985.

[Yemini85] Shaula Yemini and Daniel Berry. "A modular verifiable exception-handling mechanism," *Transactions on Programming Languages and Systems*. Vol. 7, no. 2, April 1985.

```
/* Copyright 1989 Digital Equipment Corporation.                              */
/* Distributed only by permission.                                            */
/******************************************************************************/
/* File: exception.h                                                          */
/* Last modified on Wed Mar 15 16:40:41 PST 1989 by roberts                   */
/* ——————————————————————————————————————————————————————                     */
/* The exception package provides a general exception handling mechanism     */
/* for use with C that is portable across a variety of compilers and          */
/* operating systems. The design of this facility is based on the             */
/* exception handling mechanism used in the Modula-2+ language at DEC/SRC     */
/* and is described in detail in the paper in the documents directory.        */
/* For more background on the underlying motivation for this design, see      */
/* SRC Research Report #3.                                                    */
/******************************************************************************/

#include <setjmp.h>

#define MaxExceptionsPerScope 10
#define ETooManyExceptClauses 101
#define EUnhandledException 102

#define ES_Initialize 0
#define ES_EvalBody 1
#define ES_Exception 2

typedef struct { char *name } exception;

typedef struct _ctx_block {
        jmp_buf jmp;
        int nx;
        exception *array[MaxExceptionsPerScope];
        exception *id;
        int value;
        int finally;
        struct _ctx_block *link;
} context_block;

extern exception ANY;
extern context_block *exceptionStack;
extern void _RaiseException(/* e, v */);

#define RAISE(e, v) _RaiseException(&e, v)
```

```
#define TRY \
        { \
            context_block _ctx; \
            int _es = ES_Initialize; \
            _ctx.nx = 0; \
            _ctx.link = NULL; \
            _ctx.finally = 0; \
            _ctx.link = exceptionStack; \
            exceptionStack = &_ctx; \
            if (setjmp(_ctx.jmp) != 0) _es = ES_Exception; \
            while (1) { \
                if (_es == ES_EvalBody) {

#define EXCEPT(e) \
                    if (_es == ES_EvalBody) exceptionStack = _ctx.link; \
                    break; \
                } \
                if (_es == ES_Initialize) { \
                    if (_ctx.nx >= MaxExceptionsPerScope) \
                        exit(ETooManyExceptClauses); \
                    _ctx.array[_ctx.nx++] = &e; \
                } else if (_ctx.id == &e || &e == &ANY) { \
                    int exception_value = _ctx.value; \
                    exceptionStack = _ctx.link;

#define FINALLY \
                } \
                if (_es == ES_Initialize) { \
                    if (_ctx.nx >= MaxExceptionsPerScope) \
                        exit(ETooManyExceptClauses); \
                    _ctx.finally = 1; \
                } else { \
                    exceptionStack = _ctx.link;

#define ENDTRY \
                    if (_ctx.finally && _es == ES_Exception) \
                            _RaiseException(_ctx.id, _ctx.value); \
                    break; \
                } \
                _es = ES_EvalBody; \
            } \
        }
```

```
/* Copyright 1989 Digital Equipment Corporation.                              */
/* Distributed only by permission.                                            */
/******************************************************************************/
/* File: exception.c                                                          */
/* Last modified on Wed Mar 15 16:40:42 PST 1989 by roberts                   */
/* ------------------------------------------------------------------------   */
/* Implementation of the C exception handler. Much of the real work is        */
/* done in the exception.h header file.                                       */
/******************************************************************************/

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

context_block *exceptionStack = NULL;

exception ANY;

void _RaiseException(e, v)
exception *e;
int v;
{
	context_block *cb, *xb;
	exception *t;
	int i, found;

	found = 0;
	for (xb = exceptionStack; xb != NULL; xb = xb->link) {
		for (i = 0; i < xb->nx; i++) {
			t = xb->array[i];
			if (t == e || t == &ANY) {
				found = 1;
				break;
			}
		}
		if (found) break;
	}
	if (xb == NULL) exit(EUnhandledException);
	for (cb = exceptionStack; cb != xb && !cb->finally; cb = cb->link);
	exceptionStack = cb;
	cb->id = e;
	cb->value = v;
	longjmp(cb->jmp, ES_Exception);
}
```

by Eric S. Roberts

digital

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301