

Beyond `errno` Error Handling in C

David Svoboda

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213



Software Engineering Institute

Carnegie Mellon University

© 2016 Carnegie Mellon University

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

REV-03.18.2016.0

Copyright 2016 Carnegie Mellon University and IEEE

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

[Distribution Statement A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

CERT® is a registered mark of Carnegie Mellon University.

DM-0004177

Beyond `errno`: Error Handling in C



Terms and Definitions

1. Prevention
2. Termination
3. Global Error Indicator
4. Local Error Indicator
5. Return Values
6. Assertions
7. Signals
8. Goto chains
9. Non-local Jumps
10. Error Callbacks
11. Exceptions

Summary

Beyond `errno`: Error Handling in C

Terms and Definitions



Error Handling

Systems are invariably subject to stresses that are outside the bounds of **normal operation** such as those caused by

- attack
- erroneous or malicious inputs
- hardware or software faults
- unanticipated user behavior
- unexpected environmental changes

These systems must continue to deliver essential services in a timely manner, safely and securely.

Terminology

fail safe [[IEEE Std 610.12 1990](#)]

Pertaining to a system or component that automatically places itself in a safe operating mode in the event of a failure

- A traffic light that reverts to blinking red in all directions when normal operation fails.

fail soft [[IEEE Std 610.12 1990](#)]

Pertaining to a system or component that continues to provide partial operational capability in the event of certain failures

- A traffic light that continues to alternate between red and green if the yellow light fails.
- GCC: `-fwrapv` assume that signed overflow of addition, subtraction, multiplication wraps

fail hard, also known as **fail fast** or **fail stop**.

The reaction to a detected fault is to immediately halt the system.

- GCC: `-ftrapv` generates traps for signed overflow on addition, subtraction, multiplication

Error Detection & Recovery

Error handling is often separated into detection and recovery:

- *Detection*: discovering that an error has occurred.
- *Recovery*: determining how to handle the error.

C error-handling routines vary in whether they separate detection from recovery.

Exceptions specifically allow detection and recovery to be handled by different components of the program.

Resource Cleanup

All programs use resources that are limited by the operating system:

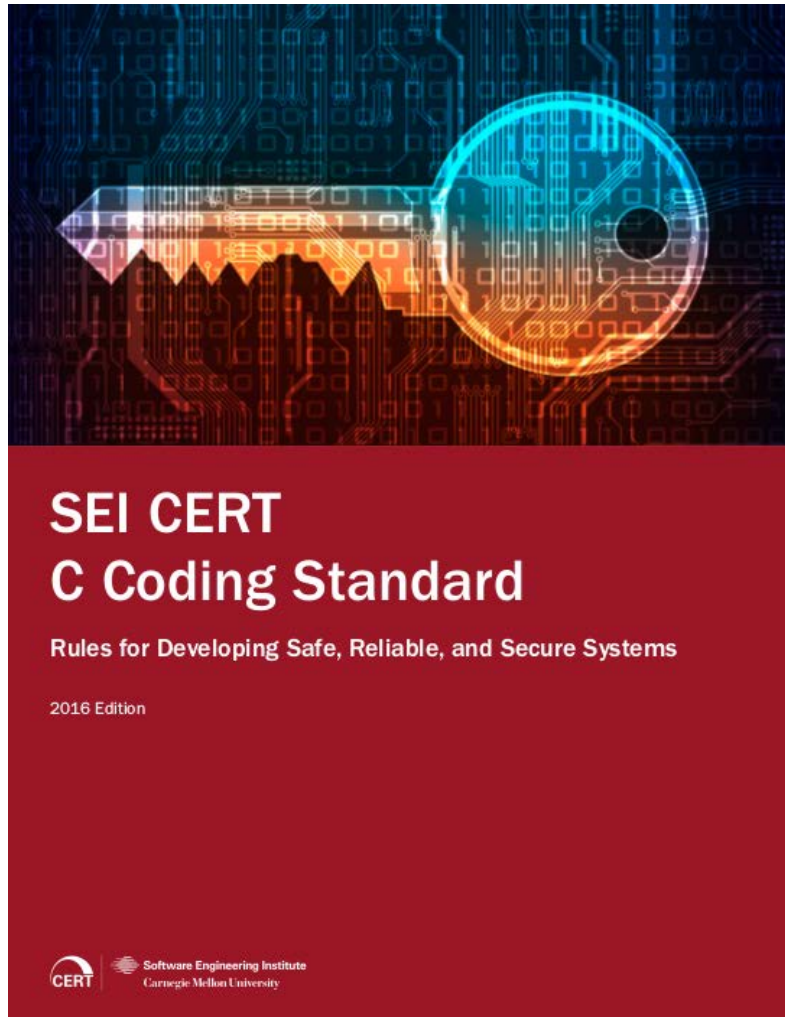
- Dynamic Memory
- Open files
- Thread locks

Exhausting these resources will cause the platform to refuse to provide more resources

- This constitutes denial-of-service!

Many programs clean up resources properly upon normal behavior, but fail to clean up resources if an error occurs!

SEI CERT Coding Standards



SEI CERT C Coding Standard

- [Version 3.0](#) published in 2016
- ISO/IEC TS 17961 C Secure Coding Rules Technical Specification
- Conformance Test Suite
- Version 2.0 (C11) published in 2014
- Version 1.0 (C99) published in 2009

SEI CERT C++ Coding Standard

- Version 1.0 under development

<https://www.securecoding.cert.org/confluence/x/BgE>



Beyond `errno`: Error Handling in C

Prevention



Prevention

When possible, preventing errors from occurring in the first place is the best approach.

- even if an error handling mechanism already exists
- [EXP34-C. Do not dereference null pointers](#)
- [INT33-C. Ensure that division and remainder operations do not result in divide-by-zero errors](#)

No support from the library is required.

Then again, some functions make prevention impractical:

- The `tan(x)` function can overflow for values of `x` near $n * \pi / 2$
 - where `n` is any odd integer.

Beyond `errno`: Error Handling in C

Termination



C/C++ Termination Functions

Function	Closes Open Streams	Flushes Stream Buffers	Removes Temporary Files	Destroys Static Objects (C++)	Calls Exit Handlers	Program Termination
<code>abort()</code>	Implementation -defined Yes in Linux	Implementation -defined Yes in Linux	Implementation -defined	No	No	Abnormal
<code>_Exit()</code>	Implementation -defined Yes in Linux	Implementation -defined No in POSIX	Implementation -defined	No	No	Normal
<code>quick_exit()</code>	Implementation -defined	Implementation -defined	Implementation -defined	No	<code>at_quick_exit()</code>	Normal
<code>exit()</code>	Yes	Yes	Yes	Yes	<code>atexit()</code>	Normal
Return from <code>main()</code>	Yes	Yes	Yes	Yes	<code>atexit()</code>	Normal

Application-independent Code

Each program has its own policy for recovering from errors

Termination is *fail-hard*.

- Useful when a function detects that the program must not continue.
 - It detects a bug in itself.
 - It detects a hacking attempt

When an application-independent (aka library) function can *detect* an error that does not mandate termination, it cannot *recover* from the error because it doesn't know how the application's error recovery policy.

Rather than respond at the point where the error was detected, an application-independent function can only indicate that the error has occurred and leave the error recovery to some other function further up the call chain.

Beyond `errno`: Error Handling in C
Global Error Indicator



Global Error Indicator

A static variable (or data structure) indicating the error status of a function call or object.

- `errno`
- `feraiseexcept()` (floating-point error indicator)
- MSVC's `GetLastError()`

These indicators are *fail-soft*, because the program can continue in spite of these errors

- Which is both a blessing and a curse!

errno

errno is an example of a global error indicator.

- It used to be a static (global) variable, in C11 it is thread-local.

Set **errno** to zero before calling a function, and use it only after the function returns a value indicating failure

The value of **errno** is zero at program startup, but is never set to zero by any library function.

The value of **errno** may be set to nonzero by a library function call whether or not there is an error, provided the use of **errno** is not documented in the description of the function in the C standard.

It is only meaningful for a program to inspect the contents of **errno** after an error has been reported.

More precisely, **errno** is only meaningful after a library function that sets **errno** on error has returned an error code.

Incorrect Use of `errno`

```
unsigned long number;
char *string = /* initialized */;
char *endptr;
/* ... */
number = strtoul(string, &endptr, 0);
if ( (endptr == string) ||
     (errno == ERANGE) ) {
    /* handle the error */
} else {
    /* computation succeeded */
}
```

This code fails to zero `errno` before invoking `strtoul()`.

Any error detected in this manner may have occurred earlier in the program, or may not represent an actual error.

Disadvantages of using `errno`

Before C11, `errno` was a global variable, with all the inherent disadvantages:

- Later system calls overwrote earlier system calls
- Global map of values to error conditions (`ENOMEM`, `ERANGE`, etc)
- Behavior is underspecified in ISO C and POSIX
- Technically `errno` is a “modifiable lvalue” rather than a global variable, so expressions like `&errno` may not be well-defined.
- Thread-unsafe

In C11, `errno` is thread-local, so it is thread-safe.

Beyond `errno`: Error Handling in C

Local Error Status



Non-global Error Indicators

These are (non-global) variables that indicate an error state.

C11, section 7.21.1, paragraph 2 describes the **FILE** type:

which is an object type capable of recording all the information needed to control a stream, including ... An error indicator that records whether a read/write error has occurred, and an end-of-file indicator that records whether the end of the file has been reached.

and provides several functions for manipulating these indicators:

- **feof()**
- **ferror()**
- **clearerr()**

Beyond `errno`: Error Handling in C

Function Return Values



Function Return Values

A function can indicate an error by its return value. Many standard functions do this:

- `getc()` returns an `int`, which is either a valid `unsigned char` or `EOF`
- `malloc()` returns a pointer to newly allocated memory or `NULL`
- `atexit()` returns 0 if successful, nonzero otherwise

Caller functions are responsible for checking for error conditions.

- [ERR33-C. Detect and handle standard library errors](#)

errno_t

C11 Annex K introduced the new type `errno_t`

The `errno_t` type (an integer) should be used as the type of an object that may contain only values that might be found in `errno`.

If your platform does not support Annex K, you can define it yourself with the following:

```
#ifndef __STDC_LIB_EXT1__
    typedef int errno_t;
#endif
```


In-Band Error Indicators

A result value that either:

- Indicates an error, or
- Conveys information beyond indicating a lack of error.

Typically returned by functions.

- but could be offered by function argument, or static variable
- `getc()` returns an `int`, which is either a valid `unsigned char` or `EOF`
- `malloc()` returns a pointer to allocated memory or `NULL`
- `atexit()` returns 0 if successful; nonzero if it fails

Which of these functions return in-band error indicators?

In-Band Error Indicators

A result value that either:

- Indicates an error, or
- Conveys information beyond indicating a lack of error.

Typically returned by functions.

- But could be offered by function argument, or static variable
- `getc()` returns an `int`, which is either a valid `unsigned char` or `EOF`
- `malloc()` returns a pointer to allocated memory or `NULL`
- `atexit()` returns 0 if successful; nonzero if it fails

Disadvantages of Function Return Values

Functions that return error indicators cannot use return value for other uses.

Checking every function call for an error condition increases code by 30-40% [[Tratt 2009](#)]

But function return values are too easy to ignore or discard.

- Impossible for library function to enforce that callers check for error condition.

Once ignored, a value in an in-band error indicator can create havoc, including malicious code execution.

- CERT Vulnerability [VU#159523](#) describes a vulnerability in Adobe Flash Player that arose partially because it ignored the return value of `calloc()`.

Ignoring Function Return Values

ignore_return.c in the exercises

```
#define SIG_DESC_SIZE 32
typedef struct {
    char sig_desc[SIG_DESC_SIZE];
} signal_info;
```

Attacker can control all inputs

```
void create_sigs(size_t sig_num, size_t temp, const char *desc) {
    signal_info *sigs = calloc(sig_num, sizeof(signal_info));
```

```
/* ... */
```

Return value of `calloc()` unchecked!

```
    signal_info *point = sigs + temp - 1;
    memcpy(point->sig_desc, desc, SIG_DESC_SIZE);
    point->sig_desc[SIG_DESC_SIZE - 1] = '\0';
```

Program lets attacker write string of their choosing to memory of their choosing!

```
/* ... */
```

```
    free(sigs);
```

```
}
```

In-Band Error Indicator Wrap-up

Do not create new data types with in-band error indicators.

- [ERR02-C. Avoid in-band error indicators](#)

Use pre-existing data types carefully:

- **NULL** pointers
- **Infinity**, **-Infinity**, **NaN** in floating-point arithmetic

Beyond `errno`: Error Handling in C

Assertions



What `assert ()` does

The C Standard, subclause 7.2.1.1, defines `assert ()` to:

- Take a single integer expression argument, and evaluate it
- If it is 0:
 - Prints a message to standard error, containing:
 - File name of source code containing assertion
 - Line number of assertion
 - Function containing assertion
 - Then calls `abort ()`

When to use `assert ()`

Assertions are primarily intended for use during debugging and are generally turned off before code is deployed by defining the `NDEBUG` macro (typically as a flag passed to the compiler).

- [MSC11-C. Incorporate diagnostic tests using assertions](#)
- [ERR06-C. Understand the termination behavior of `assert\(\)` and `abort\(\)`](#)

Use assertions to protect against incorrect programmer assumptions at run time.

When not to use `assert()`

The following code verifies that there are no padding bits in the structure:

```
#include <assert.h>

struct timer {
    unsigned char MODE;
    unsigned int DATA;
    unsigned int COUNT;
};

int func(void) {
    assert(sizeof(struct timer)
        == sizeof(unsigned char)
        + sizeof(unsigned int)
        + sizeof(unsigned int));
}
```

If compiled without debugging,
assertion never checked

If `func()` is never executed,
assertion never tested

If `func()` is executed
many times, assertion runs
many times, which is slow

Is there a better way?

Use `static_assert()`

```
#include <assert.h>
```

```
struct timer {  
    unsigned char MODE;  
    unsigned int DATA;  
    unsigned int COUNT;  
};
```

New to C11

```
static_assert(sizeof(struct timer)  
    == sizeof(unsigned char)  
    + sizeof(unsigned int)  
    + sizeof(unsigned int),  
    "Structure must not have any padding");
```

Always runs exactly once, at compile time

If fails, warning output by compiler

Other limitations of `assert()`

Assertions should never be used to verify the absence of runtime (as opposed to logic) errors, such as

- Invalid user input (including command-line arguments and environment variables)
- File errors (for example, errors opening, reading or writing files)
- Network errors (including network protocol errors)
- Out-of-memory conditions (for example, `malloc()` or similar failures)
- System resource exhaustion (for example, out-of-file descriptors, processes, threads)
- System call errors (for example, errors executing files, locking or unlocking mutexes)
- Invalid permissions (for example, file, memory, user)

Beyond `errno`: Error Handling in C

Signals



Signals

A signal is an interrupt that is used to notify a process that an event has occurred so that the process can then respond to that event accordingly.

Signals are handled by a process by registering a signal handler using the `signal()` function, which is specified as:

```
void (*signal(int sig, void (*func)(int)))(int);
```

The `raise()` function invokes the signal handler for the signal `sig`.

```
int raise(int sig);
```

If a signal handler is called, the `raise` function does not return until after the signal handler returns (if it returns).

The `raise()` function returns zero if successful, nonzero if unsuccessful.

Recommendations for Signals

Don't use signals to implement normal functionality. Signal handlers are severely limited in the actions they can perform in a portable and secure manner.

Ideally, signal handlers should do no more than read or write a flag (of type `volatile sig_atomic_t` and return.

- Programs must then poll `static sig_atomic_t` values to discover if a signal was called, and act accordingly.

Don't use signal handlers at all if the platform resets signals after invoking each signal handler. (eg Windows)

Beyond `errno`: Error Handling in C

Goto Chains



Managing Resources

cleanup1.c in the exercises

Consider the following function, which must open two files and allocate space:

```
1 void do_something(void) {
2     FILE *fin1, *fin2;
3     object_t *obj;
4
5     fin1 = fopen("file1.txt", "r");
6     fin2 = fopen("file2.txt", "r");
7     obj = malloc(sizeof(object_t));
8
9     /* ... Work with resources ... */
10
11     fclose(fin1);
12     fclose(fin2);
13     free(obj);
14 }
```

These functions return **NULL** on failure.

No error checking...will probably dereference **NULL** here.

`fclose()` doesn't accept **NULL** gracefully.

Sufficient Error Checking? 1

cleanup2.c in the exercises

```
1  errno_t do_something(void) {
2      FILE *fin1, *fin2;
3      object_t *obj;
4      errno = 0 ;
5      fin1 = fopen("file1.txt", "r");
6      if (fin1 == NULL) {
7          return errno;
8      }
9
10     fin2 = fopen("file2.txt", "r");
11     if (fin2 == NULL) {
12         return errno;
13     }
14 }
```

file1 not closed
if this call fails

file1 and file2 not
closed if this call fails

```
15     obj = malloc(sizeof(object_t));
16     if (obj == NULL) {
17         return errno;
18     }
19
20     /* ... Work with resources ... */
21
22     fclose(fin1);
23     fclose(fin2);
24     free(obj);
25     return NOERR;
26 }
```

Sufficient Error Checking? 2

cleanup3.c in the exercises

```
1  errno_t do_something(void) {
2      FILE *fin1, *fin2;
3      object_t *obj;
4      errno = 0;
5      fin1 = fopen("file1.txt", "r");
6      if (fin1 == NULL) {
7          return errno;
8      }
9
10     fin2 = fopen("file2.txt", "r");
11     if (fin2 == NULL) {
12         fclose(fin1);
13         return errno;
14     }
15
```

Sufficient, but cleanup distributed throughout code!

Does not scale!
(Imagine if there were three more files to open!)

```
16     obj = malloc(sizeof(object_t));
17     if (obj == NULL) {
18         fclose(fin1);
19         fclose(fin2);
20         return errno;
21     }
22
23     /* ... Work with resources ... */
24
25     fclose(fin1);
26     fclose(fin2);
27     free(obj);
28     return NOERR;
29 }
```

Resource Acquisition Is Initialization (RAII)

Design principle from C++

Any important resource should be controlled by an object that links the resource's lifetime to the object's lifetime.

- Every resource allocation should occur in its own statement (to avoid sub-expression evaluation order and sequence point issues).
- The object's constructor immediately puts the resource in the charge of a resource handle.
- The object's destructor frees the resource.
- Copying and heap allocation of the resource handle object are carefully controlled or outright denied.

No inherent support in C, nor does C provide any automatic cleanup.

So is there a way to properly indicate errors in this code while properly cleaning up resources?

Goto Chain

cleanup4.c in the exercises


```
1  errno_t do_something(void) {
2      FILE *fin1, *fin2;
3      object_t *obj;
4      errno_t ret_val = NOERR;
5      errno = 0;
6      fin1 = fopen("file1.txt", "r");
7      if (fin1 == NULL) {
8          ret_val = errno;
9          goto FAIL_FIN1;
10     }
11
12     fin2 = fopen("file2.txt", "r");
13     if (fin2 == NULL) {
14         ret_val = errno;
15         goto FAIL_FIN2;
16     }
17
```

```
18     obj = malloc(sizeof(object_t));
19     if (obj == NULL) {
20         ret_val = errno;
21         goto FAIL_OBJ;
22     }
23
24     /* ... Work with resources ... */
25
26     SUCCESS:
27         free(obj);
28     FAIL_OBJ:
29         fclose(fin2);
30     FAIL_FIN2:
31         fclose(fin1);
32     FAIL_FIN1:
33         return ret_val;
34 }
```

Cleanup relegated
to end of function.

All resources
cleaned properly
under any failure.

Goto Chain Pattern



```
if (op1_failure(...))
    goto OP1_CLEANUP;

if (op2_failure(...))
    goto OP2_CLEANUP;

if (op3_failure(...))
    goto OP3_CLEANUP;

/* ... Work with resources ... */

    op3_clean(...);
OP3_CLEANUP:

    op2_clean(...);
OP2_CLEANUP:

    op1_clean(...);
OP1_CLEANUP:

/* end */
```

Every operation that could fail is tested.

If error occurs, transfers to appropriate cleanup handler.

Goto still considered harmful!

Use structured loops over `goto` when possible!

C11, subclause 6.8.6.1 dictates:

A `goto` statement shall not jump from outside the scope of an identifier having a variably modified type to inside the scope of that identifier.

Do not use `goto` to transfer across functions.

- Use `return` or `longjmp()` instead.
- [MEM12-C. Consider using a goto chain when leaving a function on error when using and releasing resources](#)

Beyond `errno`: Error Handling in C

Non-local Jumps



setjmp() and longjmp()

C99 defines the `setjmp()` macro, `longjmp()` function, and `jmp_buf` type, which can be used to bypass the normal function call and return discipline.

The `setjmp()` macro saves its calling environment for later use by the `longjmp()` function.

The `longjmp()` function restores the environment saved by the most recent invocation of the `setjmp()` macro.

longjmp() Example

setjmp.c in the exercises

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

```
jmp_buf place;
```

Transfers control back to setjmp() invocation in main().

```
void func(void) {
    longjmp(place, 2);
    printf("unreachable code\n");
}
```

1st call returns 0, 2nd transfer from longjmp() returns 2

```
int main(void) {
    if (setjmp(place) != 0) {
        printf("Returned using longjmp\n");
        return 1;
    }
    func();
    printf("unreachable code\n");
}
```

This call does not return

setjmp() / longjmp() Recommendations

Do not invoke `longjmp()`:

- From a signal handler
- From an exit handler
- After the function with `setjmp()` has completed.

Even when used properly, `setjmp() / longjmp()` provides an additional **arbitrary write target**.

CERT provides these guidelines:

- [MSC22-C. Use the setjmp\(\), longjmp\(\) facility securely](#)
- [ERR52-CPP. Do not use setjmp\(\) or longjmp\(\)](#)

MISRA Guidelines for the Use of the C Language in Critical Systems contains the following rule:

- Rule 20.7 (required) The `setjmp` macro and the `longjmp` function shall not be used.

Beyond `errno`: Error Handling in C

Error Callbacks



Error Callbacks

This is simply a function pointer that is associated with code that promises to invoke the function whenever it encounters a type of error.

Typically this function pointer is a static variable, although it could be passed as a function argument.

Such code must consider the case where the callback is set to **NULL**.

Some code can accept a special callback value (such as **NULL**) that indicates that they should ignore the error.

Other code expects the callback to “fix” the error so that the code can try their operation again.

Can be *fail-hard* or *fail-soft* depending on callback function.

Example: C++ `new_handler` 1

C++ allows an error callback which is set with `std::set_new_handler()`

This callback is expected to

- Free up some memory,
- `abort()`
- `exit()`

or

- Throw an exception of type `std::bad_alloc`.

This callback must be of the standard type `new_handler`:

```
typedef void (*new_handler)();
```

Example: C++ `new_handlers` 2

`operator new` will call the new handler if it is unable to allocate memory. If the new handler returns, `operator new` will re-attempt the allocation.

```
extern void myNewHandler();

void someFunc() {
    new_handler oldHandler
        = set_new_handler(myNewHandler);
    // allocate some memory...
    // restore previous new handler
    set_new_handler(oldHandler);
}
```

Example: C11 Annex K “Bounds-checking interfaces”

The C11 Annex K functions were originally created by Microsoft to help retrofit its existing, legacy code base in response to numerous, well-publicized security incidents over the past decade.

These functions were subsequently proposed to the [ISO/IEC JTC1/SC22/WG14](#) for standardization.

These functions were published as ISO/IEC TR 24731-1 and then later incorporated in C11 in the form of a set of optional extensions specified in a conditionally-normative annex (Annex K).

Annex K Runtime Constraints

Most Annex K functions, upon detecting an error such as invalid arguments or not enough room in an output buffer, call a special **runtime-constraint** handler function.

This function might print an error message and/or abort the program.

The programmer can control which callback function is called via the **set_constraint_handler_s()** function, and can make the callback simply return if desired.

- If the callback simply returns, the function that invoked the callback indicates a failure to its caller using its return value.
- Programs that install a callback that returns must check the return value of each call to any of the bounds checking functions and handle errors appropriately.
- [ERR03-C. Use runtime-constraint handlers when calling the bounds-checking interfaces](#)

Reading input with Annex K `gets_s()` 1

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

```
void get_y_or_n(void) {
    char response[8];
    size_t len = sizeof(response);
    printf("Continue? [y] n: ");
    gets_s(response, len);
    if (response[0] == 'n')
        exit(0);
}
```

Without a custom constraint handler, if 8 characters or more are input, the behavior is implementation defined (typically `abort()`)

```
int main(void) {
    constraint_handler_t oconstraint =
        set_constraint_handler_s(ignore_handler_s);
    get_y_or_n();
}
```

This causes `gets_s()` to truncate the input and return normally.

Beyond `errno`: Error Handling in C

Exceptions



Exceptions

Feature of C++, not in standard C.

Separates *error discovery* from *error handling*.

Can be disabled in modern C++ compilers

- GCC/G++ & Clang: **-fno-exceptions**
- GCC: Code will [break](#) if any exception passes through a stack frame of a function compiled without exceptions.

C++ exceptions support RAI:

```
try {  
    std::unique_ptr<object_t> obj(new object_t);  
    /* work with obj */  
} catch (std::bad_alloc x) {  
    // handle error  
}
```

obj automatically freed
when try block exits.

Dynamic Exception Declarations

ISO C++ supported the dynamic `throw` function declaration, which indicates the types of exceptions a function can throw.

- This incurred a runtime overhead when entering functions, was never fully supported, has been deprecated, and removed from upcoming C++17 standard.

C++11 supports the `noexcept` keyword, which indicates that a function throws no exceptions.

If a `noexcept` function throws an exception, the program terminates.

```
void do_something(void) noexcept {
    try {
        std::unique_ptr<object_t> obj(new object_t);
        /* ... Work with obj ... */
    } catch (std::bad_alloc x) {
        // handle error
    }
}
```

Beyond `errno`: Error Handling in C

Conclusion



Multiple Approaches

Many systems use multiple approaches to error handling.

C's floating point operations use several mechanisms to indicate errors:

- **Global Error Indicator:** `feraiseexcept()`
- **Return Value:** `Infinity -Infinity NaN`
- **Signals:** `SIGFPE`

The degree of support for each approach is platform-dependent, and C provides several flags indicating what is supported.

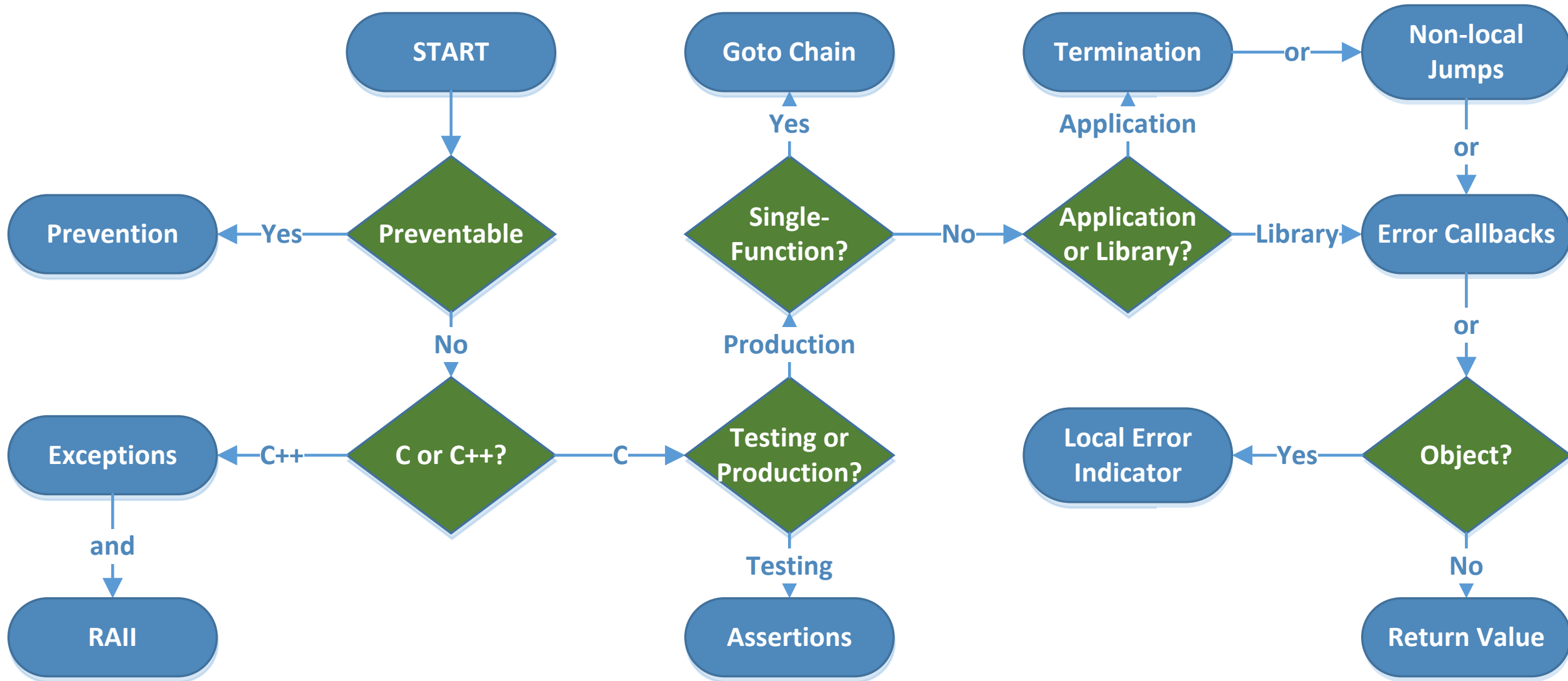
Error Handling Strategy Reference

<u>Technique</u>	<u>Code Construct</u>	<u>C11</u>	<u>CERT</u>
<i>Prevention</i>			
<i>Termination</i>	<code>abort()</code> <code>exit()</code> <code>_Exit()</code>	7.22.4.1	ERR04-C
<i>Global Error Indicator</i>	<code>errno</code>	7.5	ERR30-C ERR32-C
<i>Local Error Indicator</i>	<code>ferror()</code>	7.21.10	ERR01-C
<i>Return Value</i>	<code>return</code>	6.8.6.4	ERR02-C ERR33-C
<i>Assertions</i>	<code>assert()</code>	7.2	ERR06-C MSC11-C
<i>Signals</i>	<code>signal()</code> <code>raise()</code>	7.14.1	SIG chapter
<i>Goto Chains</i>	<code>goto</code>	6.8.6.1	MEM12-C
<i>Non-local Jumps</i>	<code>setjmp()</code> <code>longjmp()</code>	7.13	MSC22-C
<i>Error Callbacks</i>		K.3.6	ERR03-C
<i>Exceptions</i>	<code>try</code> <code>catch</code> <code>throw</code>		C++ ERR chapter

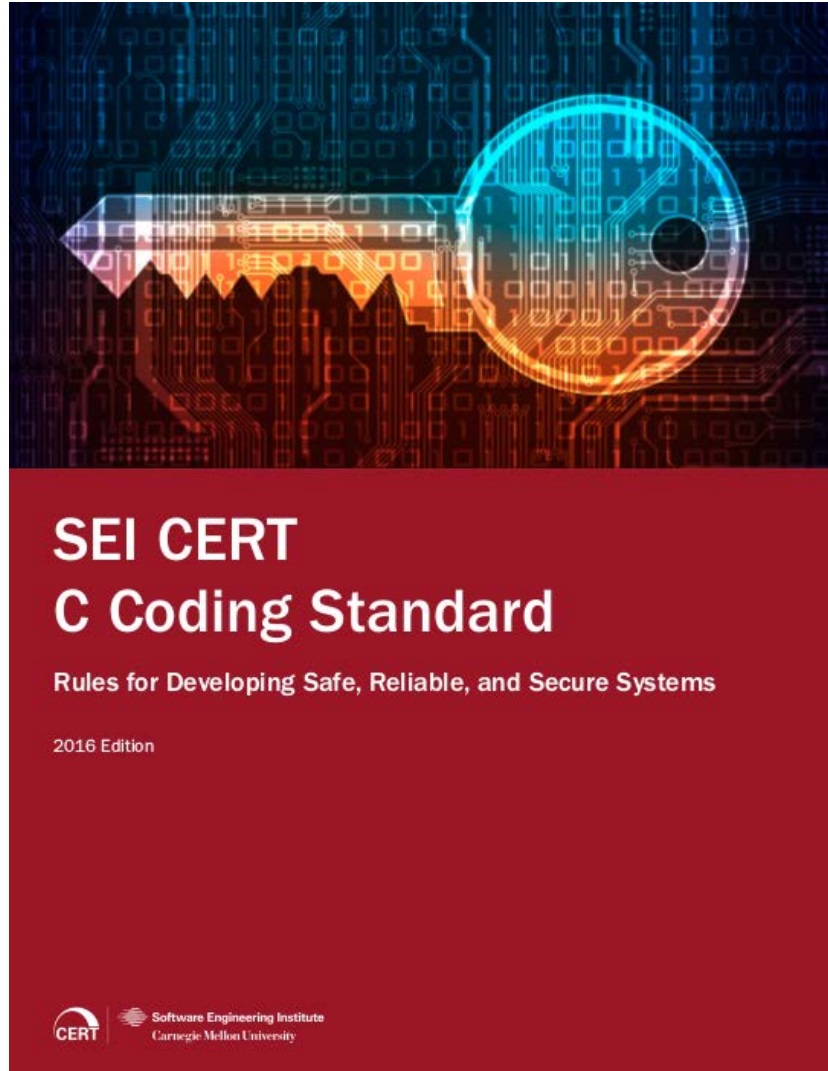
Error Handling Gotchas

<u>Technique</u>	<u>Not Ignorable?</u>	<u>Recoverable?</u>	<u>Other Problems</u>	<u>Error-Handling</u>
<i>Prevention</i>	Yes	Yes	Occasionally Infeasible	Yes
<i>Termination</i>	Yes	No	No Cleanup	Not for Libraries
<i>Global Error Indicator</i>	No	Yes	Underspecified / Global map	No
<i>Local Error Indicator</i>	No	Yes		Yes
<i>Return Value</i>	No	Yes	Precludes real return value	Yes
<i>Assertions</i>	Yes	No	No Cleanup / Disabled in Production	No
<i>Signals</i>	Yes	No	No Cleanup / Thread-Unsafe	No
<i>Goto Chains</i>	N/A	N/A	Single-Use / goto	Yes in functions
<i>Non-local Jumps</i>	Yes	No	No Cleanup / Brittle / Vul pointer	Not for Libraries
<i>Error Callbacks</i>	Yes	Yes	Cumbersome	Yes
<i>Exceptions</i>	Yes	Yes	Global Error Hierarchy / Not in C	Yes

Error Handling Decision Tree



For More Information



Visit CERT® websites:

<https://www.cert.org/secure-coding>

<https://www.securecoding.cert.org>

Contact Presenter

David Svoboda

svoboda@cert.org

(412) 268-3965

Contact CERT:

Software Engineering Institute

Carnegie Mellon University

4500 Fifth Avenue

Pittsburgh PA 15213-3890

USA

Questions?



Software Engineering Institute

Carnegie Mellon University

© 2016 Carnegie Mellon University

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Backup Slides

Backup Slides

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213



Software Engineering Institute

Carnegie Mellon University

© 2016 Carnegie Mellon University

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

REV-03.18.2016.0

Error Handling 2

To accomplish this, a system must exhibit the following system qualities

- availability
- reliability
- error tolerance
- fault tolerance
- performance
- robustness
- security

All of these system quality attributes depend upon a consistent and comprehensive error-handling policy that supports the goals of the overall system.

Terminology 1

availability [[IEEE Std 610.12 1990](#)]

The degree to which a system or component is operational and accessible when required for use. Often expressed as a probability.

reliability [[IEEE Std 610.12 1990](#)]

The ability of a system or component to perform its required functions under stated conditions for a specified period of time.

error tolerance [[IEEE Std 610.12 1990](#)]

The ability of a system or component to continue normal operation despite the presence of erroneous inputs.

fault tolerance [[IEEE Std 610.12 1990](#)]

The ability of a system or component to continue normal operation despite the presence of hardware or software faults.

The `exit()` function

Calling the Standard C `exit()` function is the “polite” way of terminating a program:

```
if (something_really_bad_happened)
    exit(EXIT_FAILURE);
```

Calling `exit()`

- flushes unwritten buffered data and closes all open files
- removes temporary files
- returns an integer exit status to the operating system.

The standard header `<stdlib.h>` defines the macro `EXIT_FAILURE` as the value indicating unsuccessful termination.

C++ programs also destroy statically-declared objects.

The `atexit()` Function

You can use `atexit()` to customize `exit()` to perform additional actions at program termination.

For example, calling:

```
atexit(turn_gizmo_off);
```

registers the `turn_gizmo_off()` function so that a subsequent call to `exit()` will invoke:

```
turn_gizmo_off();
```

as it terminates the program.

The C standard says that `atexit()` should let you register up to 32 functions.

Returning from `main()`

The C Standard, subclause 5.1.2.2.3 [[ISO/IEC 9899:2011](#)] says:

If the return type of the main function is a type compatible with `int`, a return from the initial call to the main function is equivalent to calling the `exit` function with the value returned by the main function as its argument; reaching the `}` that terminates the main function returns a value of 0. If the return type is not compatible with `int`, the termination status returned to the host environment is unspecified.

Some platforms implement this behavior as:

```
void _start(void) {  
    /* ... */  
    exit(main(argc, argv));  
}
```

The `abort()` Function

The most abrupt way to halt a program is by calling the Standard C `abort()` function.

The `abort()` function causes abnormal program termination to occur, unless the signal `SIGABRT` is being caught and the signal handler does not return.

Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed is **implementation-defined**.

An implementation-defined form of the status unsuccessful termination is returned to the host environment by means of the function call `raise(SIGABRT)`.

The `_Exit()` function

Calling the Standard C `_Exit()` function is ruder than `exit()` but less rude than `abort()`.

Calling `_Exit()`

- Does not invoke functions registered with `atexit()`
- Returns an integer exit status to the operating system.

C99 leaves these behaviors implementation-defined:

- Unwritten buffered data is flushed
- Open files are closed.
- Temporary files are removed

In C++11 adds the following about `_Exit()`

- The program is terminated without executing destructors for objects of automatic, thread, or static storage duration and without calling functions passed to `atexit()` (3.6.3).

The `quick_exit()` function

A hybrid of `exit()` and `_Exit()`, `quick_exit()`

- Calls handlers registered with `at_quick_exit()`
 - but not handlers registered with `atexit()`
- Then calls `_Exit()`

In C++:

- `exit()` calls destructors of static or thread-local objects
 - but `quick_exit()` does not.
 - Neither does `_Exit()`

False Errors 1

Some functions behave differently regarding **errno** in various standards.

When **fopen()** encounters an error:

- It returns **NULL**, according to the C standard.
- The C Standard makes no mention of **errno** when describing **fopen()**
- **errno** is set to a value indicating the error, according to POSIX.1 [[IEEE Std 1003.1-2013](#)].
- [MSVC](#) sets **errno** to **EINVAL** if the invalid parameter handler allows **fopen()** to proceed.

The implication is that a program conforming to C but not POSIX or Windows should not check **errno** after calling **fopen()**, but a POSIX / Windows program may check **errno** if **fopen()** returns a null pointer.

False Errors 2

```
FILE *fileptr;  
/* ... */  
errno = 0;  
fileptr = fopen( ... );  
if (errno != 0) {  
    /* handle error */  
}
```

Checking `errno` immediately after a call to `fopen()` may be incorrect on some platforms, because `fopen()` may set `errno`, even if no error occurred.

Compliant Solution ?

```
FILE *fileptr;
/* ... */
errno = 0;
fileptr = fopen( ... );
if (fileptr == NULL) {
    /* handle error */
    if (errno != 0) {
        /* errno useful for
           handling error */
    }
}
```

Only inspect `errno` after an error has already been detected by another means:

History of Signals

Different APIs for different systems

- **signal** (System V)
- **sigvec** (4BSD)

1990 POSIX 1003.1, aka “POSIX.1”, standardizes things somewhat

- **sigaction** (POSIX.1)

Windows POSIX subsystem capable of dealing with signals as well, but still primarily a UNIX feature.

Michal “Icamtuf” Zalewski publishes “*Delivering Signals for Fun and Profit*” (2001).

- First public discussion of the relevant vulnerabilities
- Credits OpenBSD’s Theo de Raadt with bringing them to his attention
 - OpenBSD does subsequent signal handling audit

Dowd, et. al “*The Art of Software Security Assessment*” (2006).

- Probably the most thorough treatment of the topic to date

Liabilities of Signals

Some APIs use signals for **error handling**.

Because one process can send signals to another process, signals are a form of **IPC** (Inter-Process Communication).

Because a signal handler can operate in the middle of another task, they are also a form of **concurrency**.

Finally, the different implementation details of signal handling means that signals also introduce **portability** concerns.

Therefore, signals have all of the security liabilities of **error handling**, **IPC**, **concurrency**, and **portability**.

Signal Handler Persistence

Many POSIX systems automatically reinstall signal handlers upon handler execution, meaning that the signal handler defined by the user is left in place until it is explicitly removed.

However, when a signal handler is installed with the `signal()` function in Windows and some POSIX systems, the default action is restored for that signal after the signal is triggered.

What happens if two `SIGQUIT` signals are sent to a Windows program?

- The first signal invokes the handler (if set)
- The second signal triggers the default behavior (termination), overriding a program's desire to handle the signal.

Many such programs attempt to prevent this by having the handler re-assert itself by calling `signal()`.

- But this produces a classic race condition...will the handler manage to re-assert itself before it is interrupted by the second signal?
- [SIG01-C. Understand implementation-specific details regarding signal handler persistence](#)

Function Calls in Signal Handlers

The C Standard, 7.14.1.1, paragraph 5, states that if the signal occurs other than as the result of calling the `abort()` or `raise()` function, the behavior is undefined if

...the signal handler calls any function in the standard library other than the `abort` function, the `_Exit` function, the `quick_exit` function, or the signal function with the first argument equal to the signal number corresponding to the signal that caused the invocation of the handler.

According to Section 7.14.1.1 of the C Rationale [[ISO/IEC 03](#)]:

*When a signal occurs, the normal flow of control of a program is interrupted. If a signal occurs that is being trapped by a signal handler, that handler is invoked. When it is finished, execution continues at the point at which the signal occurred. **This arrangement can cause problems if the signal handler invokes a library function that was being executed at the time of the signal.***

Many systems define an implementation-specific list of asynchronous-safe functions. Check your system's asynchronous-safe functions before using them in signal handlers.

- [SIG30-C. Call only asynchronous-safe functions within signal handlers](#)

Finally, C++ disallows any C++ features in signal handlers, except for plain lock-free atomic operations.

Accessing Objects in Signal Handlers

According to the “Signals and Interrupts” section of the C99 Rationale, other than calling a limited, prescribed set of library functions,

The C89 Committee concluded that about the only thing a strictly conforming program can do in a signal handler is to assign a value to a volatile static variable which can be written uninterruptedly and promptly return.

At the April 2008 meeting of ISO/IEC WG14, it was agreed that

- there are no known implementations in which it would be an error to read a value from a `volatile static` variable
- the original intent of the committee was that both reading and writing variables of `volatile sig_atomic_t` would be strictly conforming.

The type of `sig_atomic_t` is implementation-defined, although there are bounding constraints.

- only integer values from 0 through 127 can be assigned to a variable of type `sig_atomic_t` to be fully portable.
- the `volatile` qualifier is needed for data that cannot be cached.

Vulnerable Code

```
char *err_msg;
void handler(int signum) {
    strcpy(err_msg, "SIGINT encountered.");
}
```

err_msg is updated to indicate that the SIGINT signal was delivered.

```
int main(void) {
    signal(SIGINT, handler);
    err_msg = (char *) malloc(MAX_MSG_SIZE);
    if (err_msg == NULL) {
        /* Handle error condition */
    }
    strcpy(err_msg, "No errors yet.");
    /* Main code loop */
    return 0;
}
```

Undefined behavior occurs if a SIGINT is generated before the allocation completes.

Portably Secure Solution

```
volatile sig_atomic_t e_flag = 0;

void handler(int signum) { e_flag = 1; }

int main(void) {
    char *err_msg = (char *)malloc(MAX_MSG_SIZE);
    if (err_msg == NULL) { /* Handle error */ }
    signal(SIGINT, handler);
    err_msg strcpy(err_msg, "No errors yet.");
    while (true) {
        /* . . . */
        if (e_flag) strcpy(err_msg, "SIGINT received.");
        /* . . . */
    }
}
```

Portably, signal handlers can only unconditionally get or set a flag of type `volatile sig_atomic_t` and return.

setjmp() Undefined Behavior

Any implementation of `setjmp()` as a conventional called function cannot know enough about the calling environment to save any temporary registers or dynamic stack locations used part way through an expression evaluation.

C11, subclause 7.13.1.1 dictates that `setjmp()` cannot appear in a context other than

- the entire controlling expression of a selection or iteration statement
- one operand of a relational or equality operator with the other operand an integer
- constant expression, with the resulting expression being the entire controlling expression of a selection or iteration statement
- the operand of a unary `!` operator with the resulting expression being the entire controlling expression of a selection or iteration statement
- the entire expression of an expression statement (possibly cast to `void`).

```
jmp_buf env;  
int i;  
i = setjmp(env);
```

Undefined
behavior

State Following `longjmp()`

All accessible objects have values, and all other components of the abstract machine (for example, the floating-point status flags and the state of open files) have state as of the time the `longjmp()` function was called.

Objects of automatic storage duration have **indeterminate values** if they

- are local to the function containing the invocation of the corresponding `setjmp()` macro
- do not have volatile-qualified type
- have been changed between the `setjmp()` invocation and `longjmp()` call

Accessing an indeterminate value has **undefined behavior**

Other Limitations of `longjmp()`

Some implementations leave a process in a special state while a signal is being handled.

- Explicit reassurance must be given to the environment when the signal handler returns.
- To keep this job manageable, the C89 Committee agreed to restrict `longjmp()` to only one level of signal handling.
- Best not to call `longjmp()` from a signal handler at all

The `longjmp()` function should not be called in an exit handler either., that is, a function registered with the `atexit()` or `at_quick_exit()` functions because it might jump to code that is no longer in scope.

Exploiting `longjmp()`

The `jmp_buf` buffer must preserve the state between the `setjmp()` call and `longjmp()` call.

On IA-32/Linux, this buffer contains the saved values of the `SP` (stack pointer), `BP` (base pointer), and `PC` (program counter) registers.

To exploit a program, an attacker need merely overwrite the `PC` value with a pointer to their attack code.

A subsequent call to `longjmp()` transfers control to the attack code.

Application-independent Code

The announcement might appear as

- a return value
- an argument passed by address
- a global object (e.g., `errno`)
- some combination of the above

This is what most Standard C library functions do.

Application-independent Code

Although conceptually simple, returning error indicators can quickly become cumbersome.

For example, suppose your application contains a chain of calls in which `main()` calls `f()`, which calls `g()`, which calls `h()`.

Now, suppose reality intrudes and function `h()` has to check for a condition it can't handle.

Application-independent Code

In that case, you might rewrite `h()` so that it has a non-`void` return type, such as `errno_t`, and appropriate return statements for error and normal returns.

The function might look like:

```
errno_t h(void) {  
    if (something_bad_happened) return -1;  
    // do h  
    return 0;  
}
```

The function `h()` also needs to free any locally allocated resources.

Application-independent Code

Now `g()` is responsible to heed the return value of `h()` and act accordingly.

However, more often than not, functions in the middle of a call chain, such as `g()` and `f()`, aren't in the position to handle the error.

In that case, all they can do is look for error values coming from the functions they call and return them up the call chain.

This means you must rewrite both `f()` and `g()` to have non-`void` return types along with appropriate return statements.

These functions must also free locally allocated resources as they pass up the call chain.

Application-independent Code

```
errno_t g(void) {
    errno_t status;
    if ((status = h()) != 0)
        return status;
    // do the rest of g
    return 0;
}
errno_t f(void) {
    errno_t status;
    if ((status = g()) != 0)
        return status;
    // do the rest of f
    return 0;
}
```

Finally, the buck stops with main:

```
int main(void) {
    if (f() != 0)
        // handle the error
    // do the rest of main
    return 0;
}
```

Don't forget to
free resources!



Application-independent Code

Returning error codes via return values or arguments--effectively decouples error detection from error handling, but the costs can be high.

Passing the error codes back up the call chain increases the size of both the source code and object code and slows execution time.

- can increase the size of source/object code 30 to 40%.
- increases coding effort and reduces readability.

Application-independent Code

It's usually difficult to be sure that your code checks for all possible errors.

Static analyzers (such as Lint):

- can tell you when you've ignored a function's return value
- can't tell you when you've ignored the value of an argument passed by address.

Attributes can also be used to require that a return value be checked.

Problematic In-band Error Indicators

One example from the C standard of a troublesome in-band error indicator is `EOF`. See:

- [FIO34-C. Use int to capture the return value of character IO functions](#)
- [FIO35-C. Use feof\(\) and ferror\(\) to detect end-of-file and file errors when sizeof\(int\) == sizeof\(char\)](#)

Another problematic use of in-band error indicators from the C standard involving the `size_t` and `time_t` types is described by:

- [MSC31-C. Ensure that return values are compared against the proper type.](#)

The `sprintf()` Function

The `sprintf()` function returns the number of characters written in the array, not counting the terminating null character.

```
int i;
ssize_t count = 0;
for (i = 0; i < 9; ++i)
    count += sprintf(buf + count, "%02x ", slr);
count += sprintf(buf + count, "\n");
```

However, calls to `sprintf()` can (and will) return -1 on error conditions such as an encoding error.

If this happens on the first call (which is likely), the `count` variable, already at zero, is decremented. If this index is subsequently used, it will result in an out-of-bounds read or write.

Exceptions: ERR02-EX1

Null pointers are another example of an in-band error indicator.

Use of the null pointers is not quite as bad because it is supported for by the language.

According to C99 Section 6.3.2.3, "Pointers":

If a null pointer constant is converted to a pointer type, the resulting pointer, called a null pointer, is guaranteed to compare unequal to a pointer to any object or function.

Mitigation Strategies

A strategy for fault handling should be decided.

Consistency in fault handling should be the same with respect to critically similar parts.

A multi-tiered approach of fault prevention, fault detection, and fault reaction should be used.

Mitigation Strategies

System-defined components that assist in uniformity of fault handling should be used when available.

For one example, designing a “runtime constraint handler” (as described in ISO/IEC TR 24731-1) permits the application to intercept various erroneous situations and perform one consistent response, such as flushing a previous transaction and re-starting at the next one.

Error Handling in C

`errno` is a poor practice to emulate.

Returning integer error codes requires some registry of codes and their meanings (typically as message strings), but has the advantage that a common error printer can be conveniently used.

Another approach is to use nested exception handling, which allows procedures to recover at convenient places in the computation.

Unfortunately, there is no standard specification for this in C, but there are several implementations available, generally using `setjmp/longjmp`.