ERR04-C. Choose an appropriate termination strategy

Some errors, such as out-of-range values, might be the result of erroneous user input. Interactive programs typically handle such errors by rejecting the input and prompting the user for an acceptable value. Servers reject invalid user input by indicating an error to the client while at the same time continuing to service other clients' valid requests. All robust programs must be prepared to gracefully handle resource exhaustion, such as low memory or disk space conditions, at a minimum by preventing the loss of user data kept in volatile storage. Interactive programs may give the user the option to save data on an alternative medium, whereas network servers may respond by reducing throughput or otherwise degrading the quality of service. However, when certain kinds of errors are detected, such as irrecoverable logic errors, rather than risk data corruption by continuing to execute in an indeterminate state, the appropriate strategy may be for the system to quickly shut down, allowing the operator to start it afresh in a determinate state.

ISO/IEC TR 24772:2013, Section 6.39, "Termination Strategy [REU]," [ISO/IEC TR 24772:2013], says:

> *When a fault is detected, there are many ways in which a system can react. The quickest and most noticeable way is to fail hard, also known as fail fast or fail stop. The reaction to a detected fault is to immediately halt the system. Alternatively, the reaction to a detected fault could be to fail soft. The system would keep working with the faults present, but the performance of the system would be degraded. Systems used in a high availability environment such as telephone switching centers, e-commerce, or other "always available" applications would likely use a fail soft approach. What is actually done in a fail soft approach can vary depending on whether the system is used for safety-critical or security critical purposes. For fail-safe systems, such as flight controllers, traffic signals, or medical monitoring systems, there would be no effort to meet normal operational requirements, but rather to limit the damage or danger caused by the fault. A system that fails securely, such as cryptologic systems, would maintain maximum security when a fault is detected, possibly through a denial of service.*

And

> *The reaction to a fault in a system can depend on the criticality of the part in which the fault originates. When a program consists of several tasks, each task may be critical, or not. If a task is critical, it may or may not be restartable by the rest of the program. Ideally, a task that detects a fault within itself should be able to halt leaving its resources available for use by the rest of the program, halt clearing away its resources, or halt the entire program. The latency of task termination and whether tasks can ignore termination signals should be clearly specified. Having inconsistent reactions to a fault can potentially be a vulnerability.*

C provides several options for program termination, including `exit()`, returning from `main()`, `_Exit()`, and `abort()`.

`exit()`

Calling `exit()` causes normal program termination to occur. Other than returning from `main()`, calling `exit()` is the typical way to end a program. The function takes one argument of type `int`, which should be either `EXIT_SUCCESS` or `EXIT_FAILURE`, indicating successful or unsuccessful termination respectively. The value of `EXIT_SUCCESS` is guaranteed to be 0. The C Standard, subclause 7.22.4.4 [ISO/IEC 9899:2011], says, "If the value of status is zero or `EXIT_SUCCESS`, an implementation-defined form of the status successful termination is returned." The `exit()` function never returns.

```
#include <stdlib.h>
/* ... */

if (/* Something really bad happened */) {
  exit(EXIT_FAILURE);
}
```

Calling `exit()`

- Flushes unwritten buffered data.
- Closes all open files.
- Removes temporary files.
- Returns an integer exit status to the operating system.

The C Standard `atexit()` function can be used 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. C requires that `atexit()` can register at least 32 functions.

Functions registered by the `atexit()` function are called by `exit()` or upon normal completion of `main()`.

Note that the behavior of a program that calls `exit()` from an `atexit` handler is undefined. (See undefined behavior 182 in Annex J of the C Standard. See also ENV32-C. All exit handlers must return normally.)

`return` from `main()`

Returning from `main()` causes normal program termination to occur, which is the preferred way to terminate a program. Evaluating the `return` statement has the same effect as calling `exit()` with the same argument.

```
#include <stdlib.h>

int main(int argc, char **argv) {
  /* ... */
  if (/* Something really bad happened */) {
    return EXIT_FAILURE;
  }
  /* ... */
  return EXIT_SUCCESS;
}
```

The C Standard, subclause 5.1.2.2.3 [ISO/IEC 9899:2011], has this to say about returning from `main()`:

> *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.*

Consequently, returning from `main()` is equivalent to calling `exit()`. Many compilers implement this behavior with something analogous to

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

However, exiting from `main` is conditional on correctly handling all errors in a way that does not force premature termination. (See ERR00-C. Adopt and implement a consistent and comprehensive error-handling policy and ERR05-C. Application-independent code should provide error detection without dictating error handling.)

`_Exit()`

Calling `_Exit()` causes normal program termination to occur. Like the `exit()` function, `_Exit()` takes one argument of type `int` and never returns. However, unlike `exit()`, whether `_Exit()` closes open streams, flushes stream buffers,[1] or deletes temporary files is implementation-defined. Functions registered by `atexit()` are not executed.

[1] Note that POSIX strengthens the specification for `_Exit()` by prohibiting the function from flushing stream buffers. See the documentation of the function in The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2013 Edition [IEEE Std 1003.1:2013].

```
#include <stdlib.h>
/* ... */

if (/* Something really bad happened */) {
  _Exit(EXIT_FAILURE);
}
```

The `_exit()` function is an alias for `_Exit()`.

`abort()`

Calling `abort()` causes abnormal program termination to occur unless the `SIGABRT` signal is caught and the signal handler calls `exit()` or `_Exit()`:

```
#include <stdlib.h>
/* ... */

if (/* Something really bad happened */) {
  abort();
}
```

As with `_Exit()`, whether open streams with unwritten buffered data are flushed,[2] open streams are closed, or temporary files are removed is implementation-defined. Functions registered by `atexit()` are not executed. (See ERR06-C. Understand the termination behavior of assert() and abort().)

[2] Unlike in the case of `_Exit()`, POSIX explicitly permits but does not require implementations to flush stream buffers. See the documentation of the function in The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2013 Edition [IEEE Std 1003.1:2013].

Summary

The following table summarizes the exit behavior of the program termination functions.

Function	Closes Open Streams	Flushes Stream Buffers	Removes Temporary Files	Calls `atexit()` Handlers	Program Termination
`abort()`	ℹ	ℹ [2]	ℹ	✗	Abnormal
`_Exit()`	ℹ	ℹ [1]	ℹ	✗	Normal
`exit()`	✓	✓	✓	✓	Normal
Return from `main()`	✓	✓	✓	✓	Normal

Table legend:

- ✓ – Yes. The specified action is performed.
- ✗ – No. The specified action is not performed.
- ℹ – Implementation-defined. Whether the specified action is performed depends on the implementation.

Noncompliant Code Example

The `abort()` function should not be called if it is important to perform application-specific cleanup before exiting. In this noncompliant code example, `abort()` is called after data is sent to an open file descriptor. The data may or may not be written to the file.

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

int write_data(void) {
  const char *filename = "hello.txt";
  FILE *f = fopen(filename, "w");
  if (f == NULL) {
    /* Handle error */
  }
  fprintf(f, "Hello, World\n");
  /* ... */
  abort(); /* Oops! Data might not be written! */
  /* ... */
  return 0;
}

int main(void) {
  write_data();
  return EXIT_SUCCESS;
}
```

Compliant Solution

In this compliant solution, the call to `abort()` is replaced with `exit()`, which guarantees that buffered I/O data is flushed to the file descriptor and the file descriptor is properly closed:

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

int write_data(void) {
  const char *filename = "hello.txt";
  FILE *f = fopen(filename, "w");
  if (f == NULL) {
    /* Handle error */
  }
  fprintf(f, "Hello, World\n");
  /* ... */
  exit(EXIT_FAILURE); /* Writes data and closes f */
  /* ... */
  return 0;
}

int main(void) {
  write_data();
  return EXIT_SUCCESS;
}
```

Although this particular example benefits from calling `exit()` over `abort()`, in some situations, `abort()` is the better choice. Usually, `abort()` is preferable when a programmer does not need to close any file descriptors or call any handlers registered with `atexit()`, for instance, if the speed of terminating the program is critical.

For more details on proper usage of `abort()`, see ERR06-C. Understand the termination behavior of assert() and abort().

Risk Assessment

As an example, using `abort()` or `_Exit()` in place of `exit()` may leave written files in an inconsistent state and may also leave sensitive temporary files on the file system.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
ERR04-C	Medium	Probable	High	P4	L3

Automated Detection

Tool	Version	Checker	Description
Parasoft C/C++test	10.4.2	**CERT_C-ERR04-a**	The library functions abort, exit, getenv and system from library stdlib.h shall not be used

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Related Guidelines

SEI CERT C++ Coding Standard	VOID ERR04-CPP. Choose an appropriate termination strategy
CERT Oracle Secure Coding Standard for Java	FIO14-J. Perform proper cleanup at program termination
ISO/IEC TR 24772:2013	Termination Strategy [REU]
MITRE CWE	CWE-705, Incorrect control flow scoping

Bibliography

[IEEE Std 1003.1:2013]	XSH, System Interfaces, `exit`
[ISO/IEC 9899:2011]	Subclause 5.1.2.2.3, "Program Termination" Subclause 7.22.4, "Communication with the Environment"