ERR05-C. Application-independent code should provide error detection without dictating error handling

Application-independent code includes code that is

- Shipped with the compiler or operating system
- From a third-party library
- Developed in-house

When application-specific code detects an error, it can immediately respond with a specific action, as in

```
if (something_really_bad_happens) {
  take_me_some_place_safe();
}
```

This response occurs because the application must both detect errors and provide a mechanism for handling errors. Application-independent code, by contrast, is not associated with any application, so it cannot handle errors. However, it must still detect errors and report them to an application so that the application may handle them.

Error detection and reporting can take several forms:

- A return value (especially of type `errno_t`)
- An argument passed by address
- A global object (such as `errno`)
- `longjmp()`
- Some combination of the above

Noncompliant Code Example

This noncompliant code example consists of two application-independent functions, `f()` and `g()`. The `f()` function is part of the external API for the module; the `g()` function is an internal function.

```
void g(void) {
  /* ... */
  if (something_really_bad_happens) {
    fprintf(stderr, "Something really bad happened!\n");
    abort();
  }
  /* ... */
}

void f(void) {
  g();
  /* ... Do the rest of f ... */
}
```

If `something_really_bad_happens` in `g()`, the function prints an error message to `stderr` and then calls `abort()`. The problem is that this application-independent code does not know the context in which it is being called, so it is erroneous to handle the error.

“Smart Libraries,” Practice 23 [Miller 2004], says:

> *When a library aborts due to some kind of anomaly, it is saying there is no hope for execution to proceed normally beyond the point where the anomaly is detected. Nonetheless, it is dictatorially making this decision on behalf of the client. Even if the anomaly turns out to be some kind of internal bug in the library, which obviously cannot be resolved in the current execution, aborting is a bad thing to do. The fact is, a library developer cannot possibly know the fault-tolerant context in which his/her library is being used. The client may indeed be able to recover from the situation even if the library cannot.*

It is equally bad to eliminate the call to `abort()` from `g()`. In this case, the calling function has no indication that an error has occurred.

Compliant Solution (Return Value)

One way to inform the calling function of errors is to return a value indicating success or failure. This compliant solution ensures each function returns a value of type `errno_t`, where 0 indicates that no error has occurred:

```
const errno_t ESOMETHINGREALLYBAD = 1;

errno_t g(void) {
  /* ... */
  if (something_really_bad_happens) {
    return ESOMETHINGREALLYBAD;
  }
  /* ... */
  return 0;
}

errno_t f(void) {
  errno_t status = g();
  if (status != 0) {
    return status;
  }

  /* ... Do the rest of f ... */

  return 0;
}
```

A call to `f()` returns a status indicator, which is 0 upon success and a nonzero value upon failure indicating what went wrong.

A return type of `errno_t` indicates that the function returns a status indicator (see DCL09-C. Declare functions that return errno with a return type of errno_t).

This error-handling approach is secure, but it has the following drawbacks:

- Source and object code can significantly increase in size, perhaps by as much as 30 to 40 percent [Saks 2007b].
- All function return values must be checked (see ERR33-C. Detect and handle standard library errors).
- Functions should not return other values if they return error indicators (see ERR02-C. Avoid in-band error indicators).
- Any function that allocates resources must ensure they are freed in cases where errors occur.

Compliant Solution (Address Argument)

Instead of encoding status indicators in the return value, each function can take a pointer as an argument, which is used to indicate errors. In the following example, each function uses an `errno_t\ *` argument to report errors:

```
const errno_t ESOMETHINGREALLYBAD = 1;

void g(errno_t * err) {
  if (err == NULL) {
    /* Handle null pointer */
  }
  /* ... */
  if (something_really_bad_happens) {
    *err = ESOMETHINGREALLYBAD;
  } else {
    /* ... */
    *err = 0;
  }
}

void f(errno_t * err) {
  if (err == NULL) {
    /* Handle null pointer */
  }
  g(err);
  if (*err == 0) {
    /* ... Do the rest of f ... */
  }
  return 0;
}
```

A call to `f()` provides a status indicator that is 0 upon success and a nonzero value upon failure, assuming the user provided a valid pointer to an object of type `errno_t`.

This solution is secure, but it has the following drawbacks:

- A return status can be returned only if the caller provides a valid pointer to an object of type `errno_t`. If this argument is `NULL`, there is no way to indicate *this* error.
- Source code becomes even larger because of the possibilities of receiving a null pointer.
- All error indicators must be checked after calling functions.
- Any function that allocates resources must ensure they are freed in cases where errors occur.
- Unlike return values, static analysis tools generally do not diagnose a failure to check error indicators passed as argument pointers.

Compliant Solution (Global Error Indicator)

Instead of encoding error indicators in the return value or arguments, a function can indicate its status by assigning a value to a global variable. In the following example, each function uses a static indicator called `my_errno`.

The original `errno` variable was the standard C library's implementation of error handling using this approach.

```
errno_t my_errno; /* Also declared in a .h file */
const errno_t ESOMETHINGREALLYBAD = 1;

void g(void) {
  /* ... */
  if (something_really_bad_happens) {
    my_errno = ESOMETHINGREALLYBAD;
    return;
  }
  /* ... */
}

void f(void) {
  my_errno = 0;
  g();
  if (my_errno != 0) {
    return;
  }
  /* ... Do the rest of f ... */
}
```

The call to `f()` provides a status indicator that is 0 upon success and a nonzero value upon failure.

This solution has many of the same properties as those observed with `errno`, including advantages and drawbacks.

- Source code size is inflated, though not by as much as in other approaches.
- All error indicators must be checked after calling functions.
- Nesting of function calls that all use this mechanism is problematic.
- Any function that allocates resources must ensure they are freed in cases where errors occur.
- In general, combining registries of different sets of errors is difficult. For example, changing this compliant solution code to use `errno` is difficult and bug-prone because the programmer must be precisely aware of when C library functions set and clear `errno` and also must be aware of all valid `errno` values before adding new ones.
- Calling `f()` from other application-independent code has major limitations. Because `f()` sets `my_errno` to 0, it may be overwriting a nonzero error value set by another application-independent calling function.

For these reasons, among others, this approach is generally discouraged.

Compliant Solution (`setjmp()` and `longjmp()`)

C provides two functions, `setjmp()` and `longjmp()`, that can be used to alter control flow. Using these functions, a user can ignore error values and trust that control flow will be correctly diverted in the event of error.

The following example uses `setjmp()` and `longjmp()` to ensure that control flow is disrupted in the event of error; it also uses the `my_errno` indicator from the previous example. See MSC22-C. Use the setjmp(), longjmp() facility securely for more information on `setjmp()` and `longjmp()`.

```
#include <setjmp.h>

const errno_t ESOMETHINGREALLYBAD = 1;

jmp_buf exception_env;

void g(void) {
  /* ... */
  if (something_really_bad_happens) {
    longjmp(exception_env, ESOMETHINGREALLYBAD);
  }
  /* ... */
}

void f(void) {
  g();
  /* ... Do the rest of f ... */
}

/* ... */
if (setjmp(exception_env) != 0) {
  /*
   * If we get here, an error occurred;
   * do not continue processing.
   */
}
/* ... */
f();
/* If we get here, no errors occurred */
/* ... */
```

Calls to `f()` will either succeed or divert control to an `if` clause designed to catch the error.

- The source code is not significantly larger because the function signatures do not change, and neither do functions that neither detect nor handle the error.
- Allocated resources must still be freed despite the error.
- The application must call `setjmp()` before invoking application-independent code.
- Signals are not necessarily preserved through `longjmp()` calls.
- The use of `setjmp()`/`longjmp()` bypasses the normal function call and return discipline.
- Any function that allocates resources must ensure they are freed in cases where errors occur.

Summary

The following table summarizes the characteristics of error-reporting and error-detection mechanisms.

Method	Code Increase	Manages Allocated Resources	Automatically Enforceable
Return value	Big (30–40%)	No	Yes
Address argument	Bigger	No	No
Global indicator	Medium	No	Yes
`longjmp()`	Small	No	n/a

Risk Assessment

Lack of an error-detection mechanism prevents applications from knowing when an error has disrupted normal program behavior.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
ERR05-C	Medium	Probable	High	**P4**	**L3**

Automated Detection

Tool	Version	Checker	Description

Compass/ROSE			Could detect violations of this rule merely by reporting functions that call `abort()`, `exit()`, or `_Exit()` inside an `if` or `switch` statement. This would also catch many false positives, as ROSE could not distinguish a library function from an application function
Parasoft C /C++test	10.4.2	**CERT_C-ERR05-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 ERR05-CPP. Application-independent code should provide error detection without dictating error handling

Bibliography

[Miller 2004]
[Saks 2007b]