Error handling design patterns in non-OOP languages

Namely in ISO C

Matjaž GUŠTIN

2019-01-09

Material

- Slides available on **matjaz.it/slides**
- Slides licensed under a Creative Commons Attribution 4.0 International License (CC BY 4.0)
- LaTeX source code available at github.com/TheMatjaz/c_error_handling_design_patterns

Overview

1. A brief recap over Exceptions
2. Return codes
   2.1 Boolean codes
   2.2 Error codes
   2.3 Error flags
   2.4 Status/state codes
3. Code style for return codes
4. Using the return value's domain
5. Embedded error indicator in data type
6. `<errno.h>`
7. `<setjmp.h>`

Note: it will be very code-based and development-focused

A brief recap over Exceptions

Exceptions

In OOP languages we commonly have the **Exception** classes used to handle:

- unexpected values or states
- special cases
- non-nominal situations
- ... something that cannot be handled the normal way

Control flow break

When an Exception object is raised/thrown:

- breaks the execution flow
- makes the current function return early
- repeats the same on every function in the stack towards the `main()`
- stops if caught by a `try-catch` block
- or makes the program crash

Advantages of exceptions

- Biggest advantage: break of the control flow
- Allows separation of:
  - nominal behaviour
  - error-handling behaviour
- It’s nicer to read

From *Clean Code* by Robert C. Martin

- *”Returning error codes from command functions is a subtle violation of command query separation.”*
- **Command query separation**: *”functions should either do something or answer something, but not both. Either your function should change the state of an object [command], or it should return some information about that object [query]. Doing both often leads to confusion.”*

```python
def update_software():
    completed = False
    while not completed:
        try:
            update = download_latest_software_update()
            update.check_integrity()
            update.install()
            completed = True
        except ConnectionError:
            log.error("Retrying download later.")
            time.sleep(3600)
        except IntegrityError:
            log.error("Downloaded file corrupted.")
        except InstallationError:
            log.error("No admin rights.")
            print("Please run the program with sudo.")
            completed = True
        finally:
            update.erase_temp_files()
```

But...

- Many programming languages were created before OOP was even a thing (duh...)
- Examples: Fortran, C, Cobol, ... or the more modern Rust
- Other patterns took place, still in use today
- Main difference: **the control flow is not broken!** Manual handling is required.
- *"If a tree falls in a forest and no one is around to hear it, does it make a sound?"*

Problem and forces

- Need a way to indicate to the function caller that something could not be done and why.
- Exceptions are not available.
- Must be simple, lightweight, efficient, easy to understand.

Overview of the patterns

1. Return codes
   1.1 Boolean codes
   1.2 Error codes
   1.3 Error flags
   1.4 Status/state codes
2. Using the return value’s domain
3. Embedded error indicator in data type
4. `<errno.h>`
5. `<setjmp.h>`

Return codes

Return codes

- The function's return value indicates its execution success or failure
- Different levels of detail in the output
- A potential way to categorize them[1]
    1. Boolean codes
    2. Error codes
    3. Error flags
    4. Status/state codes

---

[1] This is just my proposal, as often all these terms are used interchangeably

Boolean return code

```c
#include <stdbool.h>

bool receive_message(message_t* message);

// Alternate version without booleans
int receive_message(message_t* message);
```

`true` (or non-0) on success

`false` (or 0) on failure

Boolean return code: consequences

- Why did it fail?
- Can we retry or not? Maybe with different settings?
- Confusing: does `false` indicate ”no error” or ”no success”?

Boolean return code: usage examples

```c
message_t rx_message;
if (!receive_message(&rx_message)) {
    puts("Reception failure");
}
```

```c
message_t rx_message;
while (!receive_message(&rx_message)) {
    sleep(5);
}
```

Hard to remember if the negation is needed or not.

Error code

```c
typedef enum {
    RX_OK = 0,
    ERROR_TIMEOUT_NOTHING_RECEIVED,
    ERROR_BROKEN_CRC,
    ERROR_INCOMPLETE_MESSAGE,
    ERROR_ANTENNA_DISCONNECTED,
} rx_code_t;

rx_code_t receive_message(message_t* message);
```

RX_OK
: on success. OK is `false` to indicate the *absence* of errors.

Others
: on failure. The actual value indicates the exact reason.

Error code: consequences

- Cleaner setup
- We can handle different failure cases in different ways
- Easy to remember which value is the success: the `false` value
- Can also be used in binary/encoded/serialized messages

```c
typedef struct {
    uint32_t error_code; // Indicate if the value makes sense
    uint32_t value;      // Value to transmit
} packed_value_t;
```

- **Arguably the most common pattern** outside of `libc`
- Longer code handling the cases

Usage examples: error code

```c
message_t rx_message;
rx_code_t error;

error = receive_message(&rx_message);
if (error != RX_OK) {  // Simply: if (error) {...}
    puts("Reception failure");
}
```

```c
message_t rx_message;
rx_code_t rx_code;

if ((rx_code = receive_message(&rx_message)) != RX_OK) {
    puts("Reception failure");
}
```

```c
message_t rx_message;
rx_code_t rx_code;
bool keep_receiving = true;

do {
    rx_code = receive_message(&rx_message);
    switch (rx_code) {
        case (RX_OK): {
            process_message(&rx_message);
            break;
        }
        case (ERROR_TIMEOUT_NOTHING_RECEIVED) {
            sleep(5);
            break;
        }
        case (ERROR_ANTENNA_DISCONNECTED) {
            puts("Please connect the antenna to the PC");
            keep_receiving = false;
            break;
        }
        default: {} // Do nothing, just retry the reception.
    }
} while (keep_receiving);
```

Usage examples: process exit status

- Also known as *exit code* or *error level*.
- Value that a process returns after its termination to the parent process.
- Usually `int32` where 0 means OK: process completed successfully.
- Non-zero values are not standardized: every OS has a different list of recommended/preferred interpretations.
- In C: the `int` value returned by `main()`. Alternatively the argument of `exit()`.

Error flags

```c
typedef enum {
    // Bit flags, each value on different bit
    RX_OK = 0x00,
    ERROR_TIMEOUT_NOTHING_RECEIVED = 0x01,
    ERROR_BROKEN_CRC = 0x02,
    ERROR_INCOMPLETE_MESSAGE = 0x04,
    ERROR_ANTENNA_DISCONNECTED = 0x08,
} rx_flag_t;
typedef uint8_t rx_code_t;

rx_code_t receive_message(message_t* message);
```

`RX_OK` on success. No flags or `false` to indicate the *absence* of errors.

Any flag on failure. Each bit expresses one reason. More than one reason possible **simultaneously**.

Error flags: consequences

In addition to the consequences of Error codes:

- Useful if multiple failures can happen simultaneously
- Can still be used if only one failure at the time
- Easy to remember which value is the success: the `false` value
- *N* bits indicate only *N* errors. Error codes indicate $2^N - 1$. Bigger integer types may be needed.
- Even longer code handling the cases (e.g. a series of `if-if-if` but not `switch-case`)
- Macros are required to group same-category flags

Status code

```c
typedef enum {
    STATUS_OK_DETERMINISTIC = 0,
    STATUS_OK_PROBABILISTIC_HIGH_PROB,
    STATUS_OK_PROBABILISTIC_LOW_PROB,
    STATUS_ERROR_ILLEGAL_HEADER,
    STATUS_ERROR_ILLEGAL_CONTENT,
    STATUS_ERROR_ILLEGAL_ENCODING,
} status_code_t;
#define STATUS_IS_OK(s) ((s) == STATUS_OK_DETERMINISTIC || (s) ==
   STATUS_OK_PROBABILISTIC_HIGH_PROB || (s) ==
   STATUS_OK_PROBABILISTIC_LOW_PROB )

status_code_t process_message(message_t* message);
```

Multiple ways to complete a task (e.g. Strategy) and we known which one was chosen.

Status code: consequences

- We can handle different success cases in different ways
- Easily confusing
- Often macros are used to group ”OK”-states, which results in a regular error code

Status code split setup

```c
typedef enum {
    STATUS_OK = 0,
    STATUS_ERROR_ILLEGAL_HEADER,
    STATUS_ERROR_ILLEGAL_CONTENT,
    STATUS_ERROR_ILLEGAL_ENCODING,
} error_code_t;
typedef enum {
    STATUS_EXTRA_DETERMINISTIC = 0,
    STATUS_EXTRA_PROBABILISTIC_HIGH_PROB,
    STATUS_EXTRA_PROBABILISTIC_LOW_PROB,
} status_extra_t;

error_code_t process_message(message_t* message,
                             status_extra_t* extra);
```

Status code split setup: consequences

- We separated the execution success/failure indicator from the additional information
- Now we need to handle two values instead of one
- What is the value of `extra` if an error happens during processing?
- What happens if I pass a `NULL` pointer for `extra`?

Nested return codes

```c
rx_code_t receive_message(message_t* message) {
    rx_code_t reception_error = RX_OK;
    encoding_code_t encoding_error = ENC_OK;

    encoding_error = prepare_message(message);
    if (encoding_error) {
        return ???;  // Which error code should we return?
    }
    ...
}
```

Nested return codes problem (cont.)

- Returning the inner error code `encoding_error` breaks abstraction layers
- Returning the outer error code `reception_error` may hide details
- Using only one `enum` for everything: very big enum?
- Combining the error code into a concatenated integer leads to confusion

Nested return codes problem (cont.)

Common solutions: break abstraction layers

- Either one huge `enum` containing every possible error
  - Each function inside the library uses the same error code data type
  - Just returning it to the upper layer
- Or combined error code
  - Example: ISO/IEC 7816 for smart cards uses `uint16`
  - High byte for category (where failed)
  - Low byte for reason (what failed)
  - 0x9000 = OK
    0x69xx = Something with the command
    0x6900 = Command not allowed
    0x6981 = Command not compatible with data structure

Code style for return codes

Stopping flow with multiple return

```
rx_code_t receive_message(message_t* message) {
    receiver_config_t config;  rx_code_t error_code;

    error_code = load_receiver_config_from_storage(&config);
    if (error_code) return error_code;  // Bad, missing brackets
    error_code = receiver_enable_peripheral();
    if (error_code) { return error_code; }  // Always use brackets
    error_code = receiver_configure(&config);
    if (error_code) { return error_code; }
    error_code = receiver_receive(message);
    return error_code;
}
```

With do−while(0) is nicer

```c
rx_code_t receive_message(message_t* message) {
    receiver_config_t config;  rx_code_t error_code;

    do {
        error_code = load_receiver_config_from_storage(&config);
        if (error_code) { break; }
        error_code = receiver_enable_peripheral();
        if (error_code) { break; }
        error_code = receiver_configure(&config);
        if (error_code) { break; }
        error_code = receiver_receive(message);
    } while (0);
    return error_code;
}
```

Loops inside do−while(0) don’t work

```c
rx_code_t receive_message(message_t* message) {
    receiver_config_t config;  rx_code_t error_code;
    do {
        error_code = load_receiver_config_from_storage(&config);
        if (error_code) { break; }
        for (int i = 0; i < 10; i++) {
            // Activate 10 peripherals
            error_code = receiver_enable_peripheral(i);
            if (error_code) { break; }
            // The break exits the for loop, not the while(0)
        }
        error_code = receiver_configure(&config);
        if (error_code) { break; }
        error_code = receiver_receive(message);
    } while (0);
    return error_code;
}
```

With `goto` works always

```c
rx_code_t receive_message(message_t* message) {
    receiver_config_t config;  rx_code_t error_code;
    error_code = load_receiver_config_from_storage(&config);
    if (error_code) { goto terminate; }
    for (int i = 0; i < 10; i++) {
        // Activate 10 peripherals
        error_code = receiver_enable_peripheral(i);
        if (error_code) { goto terminate; }
    }
    error_code = receiver_configure(&config);
    if (error_code) { goto terminate; }
    error_code = receiver_receive(message);

    terminate: {
        return error_code;
    }
}
```

My personal rules for `goto`

1. Avoid it, if you can.
2. Use only **one label per function** with a clear name:
   - Good: `goto termination`, `goto error_handling`
   - Bad: `goto failure` (Does it jump to a failing point or handles a failure?)
3. Only jump **downwards in the code** (i.e. skip some instruction, don't cycle them). Better if only to the function's end (to the `return` call).

Comparing various code styles

- Multiple `return`: clear when writing, confusing to read. Not allowed by certain industry standards (e.g. automotive).
- `do-while(0)` is clear for simple cases after getting used to it
- `goto` is clear only when used correctly and only for this kind of handling

Using the return value's domain

Return values outside the domain

- The function returns a value, not a return code.
- The value has a limited domain.
- When value out of bounds, indicates an error.

Example: writing formatted strings to a file.

```
int fprintf ( FILE * stream, const char * format, ... );
```

Returns the amount of characters written: 0 or more.
Negative on failure.

Outside the domain: consequences

- No need for additional enums
- Easy to understand if something is wrong (e.g. negative length does not make sense)
- Easy to forget to check and use error value as a good result
- Must read documentation of function in detail
- Not possible if no value outside the domain exists

Return values inside the domain

- Also known as *semipredicate problem*
- how do I distinguish between a valid output and an error indicator?

Example: parsing a string for an integer value.

```
long int strtol (const char* str, char** endptr, int base);
```

Returns the converted integer.
If no valid conversion could be performed, **returns 0**.

Return values inside the domain (cont.)

What happens if the input string contains a zero digit?

```c
strtol("12", NULL, 10); // returns 12

strtol("0", NULL, 10);  // returns 0

strtol("fdjkfnskxg", NULL, 10);  // returns 0 as well!
```

But: `strtol()` sets second parameter `endptr` to first char after the parsed number. On failed parsing `str == endptr`.

Inside the domain: consequences

- Extremely confusing
- Just never ever do that

Embedded error indicator in data type

Nullable types

The language's type system supports every value to be either `NULL`-like indicating missing data or a value.

- In Python anything can be `None`
- In SQL anything can be `NULL`
- In Java non-primitives only (`int` no, `Integer` yes): *Null object pattern*
- In C works only with pointers

Nullable types: example from SQL

ISO SQL (structured query language) used to interrogate RDBMS:

- every value can also be NULL indicating missing data
- functions often return either the value or NULL
- three-valued logic with NULL as ”undefined”:
  - `NULL AND TRUE → NULL`
  - `NULL AND FALSE → FALSE`
  - `NULL AND NULL → NULL`
  - `NULL OR NULL → NULL`

A simple pointer

A pointer to something may be NULL (have the value 0) to indicate the broken link. Otherwise it can be dereferenced.

```c
#include <stdio.h>
#include <stdint.h>

int main() {
    uint8_t* buffer = malloc(50);
    if (buffer != NULL) {
        puts("Malloc succeeded and I have a buffer");
    } else {
        puts("Malloc failed");
    }
    return 0;
}
```

A simple pointer: consequences

- Embedded in most programming languages
- Every programmer understands: cannot work on `NULL`
- No need for checking a separate error code
- Very easy to forget to check and dereference `NULL` (auch!)
- Dereferencing may be too slow
- In embedded environments may be possible work due to missing `malloc()`
- May make the code less readable (`int*` vs `int`)

Option(al) types

Polymorphic type representing a value that may or may not have meaning. Mostly functional languages.

- In Rust: `enum Option<T> { None, Some(T) }`
- In Haskell: `data Maybe a = Nothing | Just a`
- In C#: `Nullable<T>` or `T?`
- In C: manually...

```c
struct optional_uint32 {
    uint8_t error_code;  // Indicate if the value makes sense
    uint32_t value;      // Actual value
};
```

IEEE 754 floating-point values

The IEEE encoding of the real numbers has embedded handling of special cases.

- $+\infty$
  - division by (positive) zero: `1.0/0.0`
  - overflows: `pow(10.0, 500.0)`
  - always bigger than any other value
  - to check: `isinf(value) && value > 0`
- $-\infty$
  - division of negative value by zero: `-1.0/0.0`
  - or by negative zero: `1.0/-0.0`
  - underflows: `pow(-10.0, 501.0)`
  - always smaller than any other value
  - to check: `isinf(value) && value < 0`

IEEE 754 floating-point values (cont.)

- NaN (Not a Number)
  - invalid operations
    - `0.0/0.0`
    - `Infinity * 0.0`
    - `sqrt(-1.0)`
  - propagating: operations on a NaN return NaN
  - comparing with a NaN always returns `false`
  - has a quiet (just returning) and signaling variant (FPU/floatlib indicates error to the system)
  - it even has free bits to encode type of error (rare)
  - to check: `isnan(value)`
- combined check: `isfinite(value)` returns `true` when the value is not NaN or $\pm\infty$

IEEE 754 floating-point values: consequences

- IEEE standard since decades, really every computer supports it
- Hardware accelerated
- No need for pointer dereferencing or checking a separate error code
- Easy to forget to check with `isfinite()`

`<errno.h>`

<errno.h>

A standard, glorified global error code.

```c
#include <stdio.h>
#include <errno.h>   // To access the integer `errno`
#include <string.h>  // To convert `errno` to a human-readable string with strerror()

int main() {
    printf("At startup: value=%d, string=%s\n",
    errno, strerror(errno));
    FILE *file = fopen("NON_existing_file.txt", "r");
    if (file == NULL) {
        printf("After fopen fails: value=%d, string=%s\n",
        errno, strerror(errno));
    } else { fclose(file); }
    return 0;
}
```

`<errno.h>`: consequences

- Part of C standard library
- Easy to lose track who set the `errno` variable in nested code
- Global variables are bad

`<setjmp.h>`

<setjmp.h>

```c
int setjmp (jmp_buf env);
```

- Fills `env` with the current state of the calling environment, so it can be restored later.
- Returns 0 on direct invocation (when state is saved).
- Otherwise returns the value (forcibly non-zero) passed by `longjmp()` (when state is restored).

<setjmp.h> (cont.)

```c
void longjmp (jmp_buf env, int val);
```

- Restores stored `env`.
- Transfers the control to the point where `setjmp()` was last used to fill the `env`.
- Makes `setjmp()` return `val`.
- This function never returns (jumping to `setjmp()` before that).

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

static jmp_buf state;
typedef enum {
    OK = 0,
    NEGATIVE_VALUE = 1,
    TOO_BIG_VALUE = 2,
} error_code_t;

int twice4(int value) {
    if (value < 0) {
        printf("(!) Negative value: %d\n", value);
        longjmp(state, NEGATIVE_VALUE); // Restore state, set code
    } else if (value > 100) {
        printf("(!) Too big value: %d\n", value);
        longjmp(state, TOO_BIG_VALUE);  // Restore state, set code
    } else { return 2 * value; }
}
```

```c
int twice3(int value) { return twice4(value); }
int twice2(int value) { return twice3(value); }
int twice(int value)  { return twice2(value); }

int main() {
    // Initially saves state and sets error_code to 0.
    // Jumped to using longjmp(state, new_value),
    // setting error_code to new_value.
    int error_code = setjmp(state);
    if (error_code == OK) {
        int input = -10;  // -10 or 1000 jumps to else branch
        int result = twice(input);
        printf("Twice of %d is %d\n", input, result);
    } else {
        printf("Error code %d\n", error_code);
    }
    return 0;
}
```

`<setjmp.h>`: consequences

- Breaking control flow
- Good performance (avoiding functions return calls)
- A way to implement exception-like behaviour
- Like `goto` but worse: may be **very** confusing
- Often readability is more important than premature optimization

Conclusion

Wrapping up

- If your programming language supports nullable/option types: use them
- Otherwise go with return error codes or flags: 0 for OK, other values for error cases
- Write the error handling code with care, focus on **readability**

Sources

- `http://www.cplusplus.com/reference/`
- `https://en.wikipedia.org/wiki/Exception_handling`
- `https://en.wikipedia.org/wiki/Exit_status`
- `https://en.wikipedia.org/wiki/Nullable_type`
- `https://en.wikipedia.org/wiki/Floating-point_arithmetic#Special_values`
- Robert C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, ISBN-10: 9780132350884

Material

- Slides available on **matjaz.it/slides**
- Slides licensed under a Creative Commons Attribution 4.0 International License (CC BY 4.0)
- LaTeX source code available at github.com/TheMatjaz/c_error_handling_design_patterns