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

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

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.

Return codes

Return codes

- Also known as *status codes*
- The function's return value indicates its execution success or failure
- Different levels of detail
- A potential way to categorize them[1]
  1. Boolean codes
  2. Error codes
  3. Error flags

[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

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

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
- **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;
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
- *N* bits indicate only *N* errors. Error codes indicate $2^N - 1$. Bigger integer types may be needed.
- Even longer code handling the cases: need to handle all possible flags independently (e.g. a series of `if-if-if` but not `switch-case`)
- Sometimes macros are involved for operations on groups of flags

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

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

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

Code design choices with return codes

Nested return codes problem

```c
tx_code_t transmit_message(message_t* message) {
    tx_code_t tx_error = TX_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 `tx_error` may hide details

Nested return codes: solution 1

One huge enum containing every possible error
(Example: SQLite < v3.3.8)

```c
typedef enum {
    TX_OK = 0,
    ERROR_ENCODING_HEADER,
    ERROR_ENCODING_BODY,
    ERROR_WRONG_CONFIGURATION,
    ERROR_ANTENNA_DISCONNECTED,
} tx_code_t;

tx_code_t transmit_message(message_t* message) {
    tx_code_t tx_error = RX_OK;

    tx_error = prepare_message(message);
    if (tx_error) { return tx_error; }
    ...
```

Solution 1: consequences

- Every library function (inner and outer) returns the same data type
- Easy to write library: on error, just pass error code to caller
- Easy to write application: only one `enum` to handle
- Abstraction layers are broken
- Hard to understand which function from an API may return which codes

Nested return codes: solution 2

Combined error codes
(Example: ISO/IEC 7816 for smart cards)

```c
typedef enum {
    TX_OK = 0,
    ERR_ENCODING,
    ERR_ANTENNA,
} tx_categ_t;
```

```c
typedef enum {
    ENC_OK = 0,
    ENC_HEADER,
    ENC_BODY,
} tx_encoding_t;
```

```c
typedef enum {
    ANT_OK = 0,
    ANT_DISCONN,
} tx_antenna_t;
```

```c
uint16_t transmit_message(message_t* message) {
    uint8_t tx_error_low = 0;
    tx_error_low = prepare_message(message);
    if (tx_error_low) { return (ERR_ENCODING<<8) | tx_error_low; }
    ...
```

Solution 2: consequences

- Abstraction layers are ”less” broken: every byte is on its own layer
- Easy to understand which function from an API may return which codes
- Error codes need to be combined before returning
- Error codes need to be unpacked before inspection
- The application may ignore the ”more detailed byte”

Conclusion

Wrapping up

- If your programming language supports exceptions or 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