Error handling

Raul P. Pelaez

September 22, 2025

Contents

1 Introduction 1

2 The software development cycle 1

3 Error handling in C 2
- 3.1 Assertions 3
  - 3.1.1 Static assertions 4
- 3.2 Error flow control 5
  - 3.2.1 Return codes 6
  - 3.2.2 Errno 8
  - 3.2.3 Program termination 10
  - 3.2.4 Exit hooks 10
  - 3.2.5 About Exit codes 12
- 3.3 General advice 12

4 Exercises 14
- 4.1 The error handler 14

1 Introduction

Today, we will cover some basic notions of the software development cycle, focusing on how to handle our programs failing (unexpectedly or not). We will cover errors that happen at runtime, not at compile time (those are handled by the compiler). Compilation/linking errors are of a different nature, as the program cannot be run until they are fixed and the compiler will usually provide good enough (even if obtuse) information to fix them.

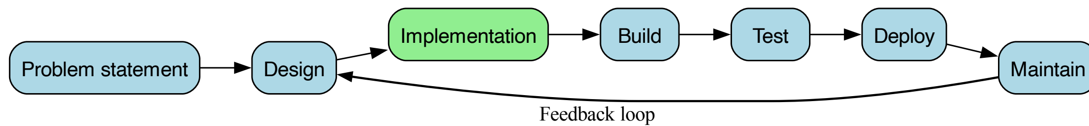
> Info
>
> **Error handling**
> The process of responding to and managing errors that occur during the execution of a program. This includes detecting errors, reporting them, and taking appropriate actions to recover from or mitigate their effects.

2 The software development cycle

Writing useful software goes beyond producing single-file source code. A typical project will consist of dozens of files, of which a few will be source code files, but many others will be configuration files, build scripts, documentation, tests, etc. We refer to anything related to software development that is not writing code as software engineering.

> Info
>
> **Software engineering**
> Anything related to the development of software that is not writing code. This includes project management, version control, testing, documentation, build systems, deployment, maintenance, etc.

The software development cycle consists of several stages, which are usually iterated over constantly while the software is being developed and maintained. These stages are:



After a first conceptualization of the problem, we design a solution, implement it, build it (compile and link), test it, and deploy it (make it available to users). Once it is out there, users will start to report errors and bugs or will request features (the maintenance stage). This cycle is iterated over constantly while the software is being developed and maintained.

The first thing to note here is that implementation only starts after a design is in place. Not a single line of code has been written before this planning stage. As you become more experienced and tackle ever more complex problems, you will realize that coding is far from the bottleneck. A solid design (aka architecture) is extremely hard to achieve, and it is the skill that will take you the most time to master.

> Advice
>
> **Design before you code**
> Always plan and architect your solution before writing code. A solid design will save you time and effort in the long run.

With a solid design in mind, implementation (the actual writing of the code) can start, and will usually be the fastest stage of the cycle. It is easy to modify the code to fix a bug or add a new feature, but a change in the design will usually cause a ripple effect that will require changes in many parts of the codebase.

The rest of the lesson today (and most of this course) will focus on this stage. You will learn about stages like deployment and maintenance in courses like DevOps.

After executing the plan by implementing the code, we need to build it (compile and link). This stage is usually automated by a build system (like Make, CMake, Meson, etc.) that will take care of compiling and linking the code. The build system will also take care of managing dependencies (as we are doing with conda).

Once the code is built, we need to test it. Testing ensures that the code works up to specification. There are many types of tests (unit tests, integration tests, system tests, etc.) that can be used to test different aspects of the code. We will cover testing in more detail in a later lesson.

3 Error handling in C

We will cover two main aspects of error handling. The first, assertions, deals with ensuring that the state of the program is as expected. The second, error flow control, is about how to take control back when the program has already failed.

3.1 Assertions

Info

Assertions
A statement that checks if a condition is true. If the condition is false, the program will terminate with an error message. Assertions are used to catch programming errors and to ensure that the program is in a valid state.

Assertions simply check (a.i. *assert*) that a condition is true. If it is not, the program will terminate. This is useful to catch programming errors and to ensure that the program is in a valid state. Assertions are used to check for conditions that should never happen (e.g., null pointers, out-of-bounds access, etc.). Assertions are not meant to handle runtime errors (e.g., file not found, out of memory, etc.) as these are expected to happen and should be handled gracefully by the program. Assertions are meant to catch programming errors that should never happen if the code is correct.

The C standard library provides the `assert` macro to implement assertions. It is defined in the `assert.h` header file. The syntax is as follows:

```
#include <assert.h>

assert(condition);
```

Where `condition` is a runtime boolean expression that should be true.

Advice

Assertions are not error handling
Assertions are not meant to handle runtime errors (e.g., file not found, out of memory, etc.) as these are expected to happen and should be handled gracefully by the program. Assertions are meant to catch programming errors that should never happen if the code is correct. This is why an assertion failure will terminate the program unavoidably.

Advice

Assertions are a kind of documentation
Assertions also help in making the restrictions and assumptions of the code explicit. For example, if a function takes a pointer as an argument, we can assert that the pointer is not null. This way, anyone reading the code will know that the function expects a valid pointer and will not have to guess or read the documentation to find out.
Other languages allow to express these restrictions via the type system itself (e.g. templates in C++, typehints in Python, nullable types in Kotlin, etc.). C does not have a rich type system, so assertions are a way to express these restrictions and assumptions explicitly in the code. They also work essentially identical in, e.g. Python.

Advice

Assert every runtime properties
Asserts can be turned off in production code by defining `NDEBUG` during compilation, so they are virtually free in terms of performance. Use them liberally to prevent bugs. For example, if you have a function that takes a pointer as an argument, you can assert that the pointer is not null:

```
#include <assert.h>
#include <stdio.h>
#define MAX_ALLOWED_SIZE 8

void foo(int *ptr, size_t size) {
    assert(ptr != NULL);
    assert(size < MAX_ALLOWED_SIZE);
    // Do something with ptr
}
```

3.1.1 Static assertions

Static assertions are a way to check for conditions at compile time. They are useful to check for conditions that can be determined at compile time (e.g., size of types, alignment, etc.).

Advice

Assert during compile-time when possible
Compilation errors are easy to deal with, as the program will not run until they are fixed.

Example

Sometimes, we will need to act upon the bit representation of data. For example, when dealing with low-level protocols or file formats. In these cases, we need to ensure that the size of the types we are using is as expected. We can use static assertions to check for this:

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

static_assert(sizeof(int) == 4, "int must be 4 bytes");

void print_bits(int value) {
    for (int i = 31; i >= 0; i--) {
        putchar((value & (1 << i)) ? '1' : '0');
        if (i % 8 == 0) putchar(' ');
    }
    putchar('\n');
}

int main() {
    int num = 42;
    print_bits(num);
    return 0;
}
```

Note that in this particular case, the "right" solution would be to use fixed-width types like `int32_t`, which C provides in the `stdint.h` header and is guaranteed to be 4 bytes on any platform.

Insight

When built in types do not have the expected size
Some platforms may have different sizes for built-in types. For example, `int` may be 2 bytes on some embedded systems. The C standard only guarantees minimum sizes for built-in types, not exact sizes. For instance, this code will probably fail to compile in Windows, while succeeding in OSX:

```
#include <assert.h>
static_assert(sizeof(long) == 8, "long must be 8 bytes");
int main() {
    return 0;
}
```

Insight

Assertions in other languages
Virtually every programming language has some form of assertions. In Python, you can use the `assert` statement in a very similar way to C:

```
from typing import List
def foo(lst: List):
    assert isinstance(lst, list), "Input must be a list"
    assert len(lst) > 0, "List must not be empty"
    # Do something with lst
```

C++ inherited the `assert` macro from C, then invented `static_assert` for compile-time assertions, which C later adopted in C11. In Java, you can use the `assert` statement as well:

```
public void foo(List lst) {
    assert lst != null : "Input must not be null";
    assert !lst.isEmpty() : "List must not be empty";
    // Do something with lst
}
```

The concept of assertions is universal due to the importance of having a mechanism to perform sanity checks in code such as type checking, boundary checking, and validating assumptions.

3.2 Error flow control

Assertions help in ensuring the state of the program at certain times. However, they do not help when the program has already failed for reasons outside of our control. For example, if our program tries to allocate more memory than the system can provide, or if a file we are reading gets deleted. In these cases, we need to have a way to handle these errors gracefully and take control back from the system. We employ error handling for two main purposes:

1. To provide useful feedback to the user about what went wrong.
2. To allow the program to recover from the error and continue running if possible.

There are several ways to handle errors in C. The most common ones are:

- Return codes
- Errno
- Exit hooks

Advice

Just let the program crash?
Sometimes, the error our code encounters is *unrecoverable*, meaning that there is no way to return to a valid state. For instance, running out of memory (`malloc` returning `NULL`) is usually one of these. Failure is always an option and, in fact, we should fail fast, early, and often.
More importantly, we should strive to *control* the failure, not just let it happen. It is much more helpful to have a program fail with a message like "Could not allocate memory in function FOO" than to let it crash and see "Segmentation Fault" in the terminal. Moreover, controlling the failure allows us to have a chance to end the program gracefully, such as saving some state to a file or freeing resources.

Advanced

Exceptions

Many languages have a built-in mechanism for error handling called exceptions. Exceptions allow us to throw an error when something goes wrong and catch it in a different part of the code. This way, we can separate the error handling code from the main logic of the program. For example, in Python, you can use the `try` and `except` keywords to handle exceptions:

```
def foo():
    try:
        # Do something that may raise an exception
        x = 1 / 0
    except ZeroDivisionError as e:
        print(f"Caught an exception: {e}")
```

C, in its simplicity, does not have a mechanism like that. That does not mean that we should ignore error handling, though. It means that we must be more disciplined and consistent in how we handle errors.

3.2.1 Return codes

The simplest way to handle errors in C is to use return codes. Functions can return an integer value that indicates whether the function succeeded or failed. By convention, a return value of 0 indicates success, while any non-zero value indicates failure. The specific non-zero values can be used to indicate different types of errors. For example, the `fopen` function returns a pointer to a `FILE` object if it succeeds, or `NULL` if it fails. We can check the return value to see if the file was opened successfully:

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

void process_file(const char *filename) {
  FILE *file = fopen(filename, "r");
  if (file == NULL) {
    fprintf(stderr, "Error: Could not open file %s\n", filename);
    return;
  }
  // Do something with the file
  fclose(file);
}

int main() {
  process_file("non_existent_file.txt");
  return 0;
}
```

Many standard library functions use this convention, typically when they return a pointer as a result, which is easy to hijack to indicate an error (by returning `NULL`). Functions that return integers can also use negative values to indicate errors, as in the case of `strcmp`, which returns a negative value if the first string is less than the second, a positive value if the first string is greater than the second, and 0 if they are equal. Examples of functions that use return codes for error handling include `malloc`, `free`, `fopen`, `fclose`, `read`, `write`, etc.

Info

The `NULL` pointer

`NULL` is a special pointer value that indicates that the pointer does not point to any valid memory location. `NULL` also transforms to `0` in integer contexts, so it is common to see checks like `if (ptr)` or `if (!ptr)` to check if a pointer is valid or not. `NULL` is often used to indicate errors when a function returns a pointer. For example, `malloc` returns `NULL` if it fails to allocate memory.

One downside of return codes is that they tend to capitalize the return value. This means that if a function needs to return something besides the error, it needs to use other mechanisms, such as

output parameters (pointers passed as arguments to the function that the function can modify to return additional information). When designing the API of a function, we have to consider how error reporting will impact its usability.

Example

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

int divide(int a, int b, int *result) {
  if (b == 0) {
    return -1; // Error: division by zero
  }
  *result = a / b;
  return 0; // Success
}

int main() {
  int result;
  if (divide(10, 0, &result) != 0) {
    fprintf(stderr, "Error: Division by zero\n");
    return EXIT_FAILURE;
  }
  printf("Result: %d\n", result);
  return EXIT_SUCCESS;
}
```

Info

fprintf and streams
The `fprintf` function is similar to `printf`, but it allows us to specify a stream to write to. In this case, we are writing the error message to the standard error stream (`stderr`) instead of the standard output stream (`stdout`, default for `printf`). This is useful for separating normal output from error messages. There is also `stdlog` for logging messages.

Advice

Always check for the `NULL` pointer
In C, any function that takes a pointer as an argument is subject to receiving a `NULL` pointer. Always check for it before dereferencing the pointer.
Example

```c
void foo(int *ptr) {
    if (ptr == NULL) {
        fprintf(stderr, "Error: ptr is NULL\n");
        return;
    }
    // Do something with ptr
}
```

This function treats a `NULL` pointer as a potentially recoverable error, prints an error message, and returns early. If, on the other hand, we consider this function receiving a `NULL` pointer something that should never happen, we can use an assertion instead:

```c
#include <assert.h>
void foo(int *ptr) {
    assert(ptr != NULL);
    // Do something with ptr
}
```

One alternative to overcome the limitation of return codes is to use `errno`, which we will cover next.

3.2.2 Errno

errno is a global variable defined in the errno.h header file that is set by system calls and some library functions in the event of an error to indicate what went wrong. The value of errno is an integer that corresponds to a specific error code. The C standard library provides a set of macros that define these error codes, such as EINVAL (invalid argument), ENOMEM (out of memory), EIO (input/output error), etc. To use errno, we need to include the errno.h header file and check its value after a function call that may fail. If the function fails, we can use the perror function to print a human-readable error message based on the value of errno. For example, the open function returns -1 if it fails to open a file and sets errno to indicate the error:

```c
#include <errno.h>
#include <fcntl.h> // For open
#include <stdio.h> // For perror
#include <string.h> // For strerror
#include <unistd.h> // For close

void process_file(const char *filename) {
  int fd = open(filename, O_RDONLY);
  if (fd == -1) {
    perror("Error opening file");
    // Alternatively, we can use strerror to get the error message
    fprintf(stderr, "Error code: %d (%s)\n", errno, strerror(errno));
    close(fd);
    errno = 0;
    return;
  }
  // Do something with the file descriptor
  close(fd);
}
int main() {
  process_file("non_existent_file.txt");
  return 0;
}
```

Info

open vs fopen
The open function is a low-level system call that opens a file and returns a file descriptor (an integer) that can be used to read from or write to the file. It is defined in the fcntl.h header file. The fopen function, on the other hand, is a higher-level function that opens a file and returns a pointer to a FILE object that can be used with other standard library functions like fread, fwrite, fprintf, etc. It is defined in the stdio.h header file. The fopen function internally uses open to open the file.

errno is an remnant of an ancient time, but some functions in the standard use it to report errors, so we are stuck with it. If the error is recoverable (e.g., file not found), we can handle it gracefully and continue running the program. In this cases, we must remember to reset errno to 0 after handling the error, as it is not reset automatically by the system.

Info

perror
The `perror` function prints a human-readable error message to the standard error stream based on the value of `errno`. It takes a single argument, which is a string that will be printed before the error message. If the argument is `NULL`, only the error message will be printed.
strerror
The `strerror` function returns a pointer to a string that describes the error code passed as an argument. This is useful if we want to include the error message in a custom error message or log it to a file.

Other functions, such as `strtol`, use `errno` to report errors while still returning a valid value. In this case, we need to check both the return value and `errno` to determine if an error occurred.

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

int main() {
  long val = strtol("a123abc", NULL, 10);
  if (errno == ERANGE) {
    // Handle out-of-range error
    fprintf(stderr, "Value out of range\n");
  } else if (errno == EINVAL) {
    // Handle invalid input error
    fprintf(stderr, "Invalid input\n");
  } else {
    // Successful conversion
    printf("Converted value: %ld\n", val);
  }
  errno = 0; // Reset errno for future operations
  return EXIT_SUCCESS;
}
```

Advice

Avoid global variables
`errno` is a global variable (also called a state variable), which means that it can be modified by any function. Global variables have a lot of downsides beyond readability (look at the example above and tell me that you did not wonder where `errno` is defined). For instance, global variables are awkward to deal with in multi-threaded programs, as they can be modified by any thread at any time. Many functions in the standard library use `errno` for two reasons: historical and convenience. However, using global variables is generally considered bad practice.
You should avoid global variables in your code, which break encapsulation and scope and make reasoning about the code harder. Do not design your APIs to use global variables like `errno` for error reporting.

Error reporting via `errno` or return codes also pose a discoverability problem. Knowing which mechanism a function uses to report error is not obvious, and the precise meaning of the error code is also not self-evident. This is why modern languages have adopted exceptions as the primary mechanism for error reporting, as they are more explicit and easier to use. In C, we must keep the standard library documentation at hand to know how to handle errors properly. See for instance the documentation for `fopen`: `https://en.cppreference.com/w/c/io/fopen.html` . Go down to "Return value", which describes how to handle errors for this function.

Sometimes, detecting errors is not enough or it is simply outside of our control. In this situations program termination is inevitable, but we can still take control of it and end the program gracefully. For example, we may want to save some state to a file or free resources before exiting.

Let us see how we can influence termination in C.

3.2.3 Program termination

Besides returning a value from `main`, C offers several functions that will terminate the program that can be called from anywhere:

- `[[noreturn]] void exit(int status)`: Equivalent to returning `status` from `main`.
- `[[noreturn]] void abort(void)`: Abnormal termination of the program. This function will terminate without any cleanup (no `atexit` functions will be called, no buffers will be flushed, etc.). It is usually used to indicate a fatal error.
- `[[noreturn]] void _Exit(int status)`: Similar to `abort`, but it allows us to specify an exit status. This function will also terminate without any cleanup.
- `[[noreturn]] void quick_exit(int status)`: Similar to `exit`, but it will only call functions registered with `at_quick_exit`. This function is not part of the C standard, but it is available in some implementations (e.g., glibc).

> Advice
>
> **Just use exit**
> Default to calling `exit` (or return from main) unless you know pretty well what you are doing.

> Advanced
>
> **The `noreturn` attribute**
> This attribute means that the function will not return to the caller. This allows the compiler to optimize the code better and avoid warnings about control reaching the end of a non-void function.
> Example:

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

[[noreturn]] void fatal_error(const char *message) {
    fprintf(stderr, "Fatal error: %s\n", message);
    exit(EXIT_FAILURE);
}

int main() {
    fatal_error("Something went terribly wrong");
    // The compiler knows that this point is never reached
    return 0;
}
```

3.2.4 Exit hooks

At this stage, we have seen how to detect errors and how to terminate the program. However, we have not seen how to take control of the termination process and end the program gracefully. This is where exit hooks come into play.

> Info
>
> **Hooks**
> In programming, a hook is a mechanism that allows us to register a function to be called at a specific point in the program's execution. Hooks are often used to allow customization or extension of the program's behavior without modifying its source code.

C provides two functions to register hooks that will be called when the program is terminating:

- `int atexit(void (*func)(void))`: Registers a function to be called when the program is terminating via `exit` or returning from `main`. Multiple functions can be registered, and they will be called in reverse order of registration (last registered, first called).

- `int at_quick_exit(void (*func)(void))`: Registers a function to be called when the program is terminating via `quick_exit`. Multiple functions can be registered, and they will be called in reverse order of registration (last registered, first called).

Example

```c
#include <stdio.h>
#include <stdlib.h>
void goodbye_message() {
  printf("------- Thanks for using this program -------\n");
}

void cleanup2() {
  printf("Cleanup 2\n");
}

int main() {
  atexit(goodbye_message);
  atexit(cleanup2);
  // Do something
  printf("Exiting program\n");
  return EXIT_SUCCESS;
}
```

Output

```
Exiting program
Cleanup 2
------- Thanks for using this program -------
```

Info

Function parameters
In C, functions can take other functions as parameters. This is done by specifying the function's signature (return type and parameter types) in the parameter list.
Example

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

int operate(float x){
  return (int)(x * 2);
}

int safe_operate(float x){
  if (x < 0) {
    fprintf(stderr, "Error: x must be non-negative\n");
    exit(EXIT_FAILURE);
  }
  return operate(x);
}

void apply_and_print(float x, int (*func)(float)){
  int result = func(x);
  printf("Result: %d\n", result);
  fflush(stdout);
}

int main() {
  apply_and_print(3.5, operate);
  apply_and_print(3.5, safe_operate);

  apply_and_print(-3.5, safe_operate); // This will terminate the program
  return 0;
}
```

Output

```
Result: 7
Result: 7
Error: x must be non-negative
```

3.2.5 About Exit codes

The value returned by `main`, or the argument to `exit`, are referred to as *exit codes*. The exit code is communicated to the terminal executing the command, we can inspect the exit code of the last command with an environment variable called `$?` in UNIX systems. In Windows, the equivalent is `$LASTEXITCODE`.

```
% echo $? # In OSX or Linux
PS > echo $LASTEXITCODE # In Windows Powershell
```

Try it out! Run a program that returns `EXIT_SUCCESS` (0) and another that returns `EXIT_FAILURE` (1) and check the value of `$?` or `$LASTEXITCODE` right after executing it.

3.3 General advice

Error-handling is a very nuanced topic. We have seen many choices that allow to deal with execution going wrong in a program. The boundaries between these methods are not always clear-cut and depend on the context. The most important thing is to design your code to reduce complexity. If you find yourself writing a lot of error-handling, it is often a sign of a flawed design. Mind you, this does not

mean at all that you should not handle errors in lieu of code simplicity. It means that you should try to *design errors out of existence.*

Advice

Design errors out of existence
This advice might sound a bit nebulous at first, and it is really hard to master. The idea is to design/redesign your code in a way that makes it impossible for a certain error to happen, leveraging the rules and syntax of the language. Lets see an example with a function that can only work for positive integers:

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

int factorial(int n) {
    if (n < 0) {
        fprintf(stderr, "Error: n must be non-negative\n");
        return -1; // Error
    }
    if (n == 0) return 1;
    return n * factorial(n - 1);
}

int main() {
    int n = 5;
    int result = factorial(m);
    printf("factorial(%d) = %d\n", m, result);
    return EXIT_SUCCESS;
}
```

In the `factorial` function, we have to check if the input is valid (non-negative) and handle the error if it is not. This adds complexity to the code and makes it harder to read and maintain. Instead, we can "design away" this input sanitation by using the type system:

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

unsigned long factorial(unsigned long n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
```

It is now, by design, impossible for the body of this function to receive a negative number. The API for this function makes it clear to the reader to just by looking at the signature and seeing `unsigned`. Granted, this is a bad example, since we should also check for overflow, but please bear with me :). The more you practice this, the more natural it will become to you.

Advanced

Calling yourself: Recursive functions
A function that calls itself is called a recursive function. Recursion is a powerful technique that can be used to solve problems that can be broken down into smaller subproblems. Recursive functions must have a base case that stops the recursion, otherwise they will call themselves indefinitely and eventually crash the program (stack overflow). We will learn more about recursion in a later lesson.

4 Exercises

4.1 The error handler

Goal

Add error handling to the following code, which has a series of potentially unsafe operations.

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

void print_array(int *arr, int size) {
  for (int i = 0; i < size; i++) {
    printf("%d ", arr[i]);
  }
  printf("\n");
}

void vectorized_divide(int *a, int *b, int *result, int size) {
  for (int i = 0; i < size; i++) {
    result[i] = a[i] / b[i];
  }
}

int main() {
  int numbers[5] = {1, 2, 3, 4, 20};
  int divisors[5] = {0, 1, 2, 3, 4};
  int results[5] = {0};
  vectorized_divide(numbers, divisors, results, 5);
  print_array(results, sizeof(results) / sizeof(results[0]));
  return EXIT_SUCCESS;
}
```

Milestone

Make sure that you are able to compile and run the code. As written, the code should compile and run without complaining, perhaps printing some garbage values when dividing by zero.
It is ok if you want to work on this exercise with an online compiler like godbolt. You can also use VSCode to edit it and compile it directly in your terminal:

```
% clang errors.c -o errors
# Or in Windows
PS > cl errors.c
```

However, I suggest you take this time to integrate it into the course's template project, at `https://github.com/RaulPPelaez/c-template-project`. This will give you practice with using CMake, git, conda and the terminal, which are essential tools for any software developer. In order to do that, you will need to:

1. Clone the repository with `git clone https://github.com/RaulPPelaez/c-template-project.git` . You can also just download the ZIP from the repo.
2. Open the terminal (or `Anaconda Powershell` in Windows) and navigate to the project folder using `cd`.
3. Create and activate the conda environment with:
   ```
   conda env update -n cdev
   conda activate cdev
   ```
4. Open the project folder in VSCode, using the `code .` command in the terminal.
5. Add your C file to the `src/homework` folder.
6. Add an executable target to the `CMakeLists.txt` file in the `src/homework` folder. Mimic how it is done in the `CMakeLists.txt` file in the `src/` folder.
7. Go to the project root in the terminal and run CMake to compile the project, which now includes your code
   ```
   cmake -B build
   cmake --build build
   ```
8. Run your program from the `build/src/homework` folder.
   ```
   % ./build/src/homework/errors
   ```

Milestone

Identify all potential error conditions in the `print_array` and `vectorized_divide` functions. These can be related to invalid input, memory allocation failures, division by zero, etc.

Milestone

For each identified error condition, decide on an appropriate error handling strategy. This could involve returning error codes, using assertions, or terminating the program gracefully.

Milestone

Implement the chosen error handling strategies in the code. Ensure that the program can handle errors gracefully and provides useful feedback to the user.

hint

- Focus on each function definition, not on how its called.
- There are many ways in which these functions can fail. The input might be invalid, some operation might fail, etc.

Advanced milestone

Register an exit hook to print a message at exit. Make it print the current value of `errno`.