ZED SHAW'S HARD WAY SERIES

Learn C the HARD WAY

Practical Exercises on the Computational Subjects You Keep Avoiding (Like C)

ZED A. SHAW

LEARN C
THE HARD WAY

```
    return 1;

error:
    return -1;
}

int test_check_debug()
{
    int i = 0;
    check_debug(i != 0, "Oops, I was 0.");

    return 0;
error:
    return -1;
}

int main(int argc, char *argv[])
{
    check(argc == 2, "Need an argument.");

    test_debug();
    test_log_err();
    test_log_warn();
    test_log_info();

    check(test_check("ex19.c") == 0, "failed with ex19.c");
    check(test_check(argv[1]) == -1, "failed with argv");
    check(test_sentinel(1) == 0, "test_sentinel failed.");
    check(test_sentinel(100) == -1, "test_sentinel failed.");
    check(test_check_mem() == -1, "test_check_mem failed.");
    check(test_check_debug() == -1, "test_check_debug failed.");

    return 0;

error:
    return 1;
}
```

Pay attention to how check is used, and when it's `false`, it jumps to the `error:` label to do a cleanup. The way to read those lines is, "check that A is true, and if not, say M and jump out."

What You Should See

When you run this, give it some bogus first parameter to see this:

Exercise 19 Session

```
$ make ex19
cc -Wall -g -DNDEBUG    ex19.c   -o ex19
$ ./ex19 test
```

```
[ERROR] (ex19.c:16: errno: None) I believe everything is broken.
[ERROR] (ex19.c:17: errno: None) There are 0 problems in space.
[WARN] (ex19.c:22: errno: None) You can safely ignore this.
[WARN] (ex19.c:23: errno: None) Maybe consider looking at: /etc/passwd.
[INFO] (ex19.c:28) Well I did something mundane.
[INFO] (ex19.c:29) It happened 1.300000 times today.
[ERROR] (ex19.c:38: errno: No such file or directory) Failed to open test.
[INFO] (ex19.c:57) It worked.
[ERROR] (ex19.c:60: errno: None) I shouldn't run.
[ERROR] (ex19.c:74: errno: None) Out of memory.
```

See how it reports the exact line number where the `check` failed? That's going to save you hours of debugging later. Also, see how it prints the error message for you when `errno` is set? Again, that will save you hours of debugging.

How the CPP Expands Macros

It's now time for you to get a short introduction to the CPP so that you know how these macros actually work. To do this, I'm going to break down the most complex macro from `dbg.h`, and have you run `cpp` so you can see what it's actually doing.

Imagine that I have a function called `dosomething()` that returns the typical 0 for success and -1 for an error. Every time I call `dosomething`, I have to check for this error code, so I'd write code like this:

```
int rc = dosomething();

if(rc != 0) {
    fprintf(stderr, "There was an error: %s\n", strerror());
    goto error;
}
```

What I want to use the CPP for is to encapsulate this `if-statement` into a more readable and memorable line of code. I want what you've been doing in `dbg.h` with the `check` macro:

```
int rc = dosomething();
check(rc == 0, "There was an error.");
```

This is *much* clearer and explains exactly what's going on: Check that the function worked, and if not, report an error. To do this, we need some special CPP tricks that make the CPP useful as a code generation tool. Take a look at the `check` and `log_err` macros again:

```
#define log_err(M, ...) fprintf(stderr,\
    "[ERROR] (%s:%d: errno: %s) " M "\n", __FILE__, __LINE__,\
    clean_errno(), ##__VA_ARGS__)
#define check(A, M, ...) if(!(A)) {\
    log_err(M, ##__VA_ARGS__); errno=0; goto error; }
```

The first macro, `log_err`, is simpler. It simply replaces itself with a call to `fprintf` to `stderr`. The only tricky part of this macro is the use of `...` in the definition `log_err(M, ...)`. What this does is let you pass variable arguments to the macro, so you can pass in the arguments that should go to `fprintf`. How do they get injected into the `fprintf` call? Look at the end for the ##__VA_ARGS__, which is telling the CPP to take the args entered where the `...` is, and inject them at that part of the `fprintf` call. You can then do things like this:

```
log_err("Age: %d, name: %s", age, name);
```

The arguments `age, name` are the `...` part of the definition, and those get injected into the fprintf output:

```
    fprintf(stderr, "[ERROR] (%s:%d: errno: %s) Age %d: name %d\n",
        __FILE__, __LINE__, clean_errno(), age, name);
```

See the `age, name` at the end? That's how `...` and ##__VA_ARGS__ work together, which will work in macros that call other variable argument macros. Look at the `check` macro now and see that it calls `log_err`, but `check` is *also* using the `...` and ##__VA_ARGS__ to do the call. That's how you can pass full `printf` style format strings to `check`, which go to `log_err`, and then make both work like `printf`.

The next thing to study is how `check` crafts the `if-statement` for the error checking. If we strip out the `log_err` usage, we see this:

```
if(!(A)) { errno=0; goto error; }
```

Which means: If A is false, then clear `errno` and `goto` the `error` label. The `check` macro is being replaced with the `if-statement`, so if we manually expand out the macro `check(rc == 0, "There was an error.")`, we get this:

```
    if(!(rc == 0)) {
        log_err("There was an error.");
        errno=0;
        goto error;
    }
```

What you should be getting from this trip through these two macros is that the CPP replaces macros with the expanded version of their definition, and it will do this *recursively*, expanding all of the macros in macros. The CPP, then, is just a recursive templating system, as I mentioned before. Its power comes from its ability to generate whole blocks of parameterized code, thus becoming a handy code generation tool.

That leaves one question: Why not just use a function like `die`? The reason is that you want `file:line` numbers and the goto operation for an error handling exit. If you did this inside a function, you wouldn't get a line number where the error actually happened, and the goto would be much more complicated.

Another reason is that you still have to write the raw `if-statement`, which looks like all of the other `if-statements` in your code, so it's not as clear that this one is an error check. By wrapping the `if-statement` in a macro called `check`, you make it clear that this is just error checking, and not part of the main flow.

Finally, CPP has the ability to *conditionally compile* portions of code, so you can have code that's only present when you build a developer or debug version of the program. You can see this already in the `dbg.h` file where the debug macro only has a body if the compiler asks for it. Without this ability, you'd need a wasted `if-statement` that checks for debug mode, and then wastes CPU capacity doing that check for no value.

Extra Credit

- Put `#define NDEBUG` at the top of the file and check that all of the debug messages go away.
- Undo that line, and add `-DNDEBUG` to `CFLAGS` at the top of the `Makefile`, and then recompile to see the same thing.
- Modify the logging so that it includes the function name, as well as the `file:line`.

This page intentionally left blank

Advanced Debugging Techniques

I've already taught you about my awesome debug macros, and you've been using them. When I debug code I use the `debug()` macro almost exclusively to analyze what's going on and track down the problem. In this exercise, I'm going to teach you the basics of using GDB to inspect a simple program that runs and doesn't exit. You'll learn how to use GDB to attach to a running process, stop it, and see what's happening. After that, I'll give you some little tips and tricks that you can use with GDB.

This is another video-focused exercise where I show you advanced debugging tricks with my technique. The discussion below reinforces the video, so watch the video first. Debugging will be much easier to learn by watching me do it first.

Debug Printing versus GDB

I approach debugging primarily with a "scientific method" style: I come up with possible causes and then rule them out or prove that they cause the defect. The problem many programmers have with this approach is that they feel like it will slow them down. They panic and rush to solve the bug, but in their rush they fail to notice that they're really just flailing around and gathering no useful information. I find that logging (debug printing) forces me to solve a bug scientifically, and it's also just easier to gather information in most situations.

In addition, I have these reasons for using debug printing as my primary debugging tool:

- You see an entire tracing of a program's execution with debug printing of variables, which lets you track how things are going wrong. With GDB, you have to place `watch` and debug statements all over the place for everything you want, and it's difficult to get a solid trace of the execution.
- The debug prints can stay in the code, and when you need them, you can recompile and they come back. With GDB, you have to configure the same information uniquely for every defect you have to hunt down.
- It's easier to turn on debug logging on a server that's not working right, and then inspect the logs while it runs to see what's going on. System administrators know how to handle logging, but they don't know how to use GDB.
- Printing things is just easier. Debuggers are always obtuse and weird with their own quirky interfaces and inconsistencies. There's nothing complicated about `debug("Yo, dis right? %d", my_stuff);`.