Exercise 19. Zed’s Awesome Debug Macros

There’s a reoccurring problem in C that we’ve been dancing around, but I’m going to solve it in this exercise using a set of macros I developed. You can thank me later when you realize how insanely awesome these macros are. Right now, you don’t know how awesome they are, so you’ll just have to use them, and then you can walk up to me one day and say, “Zed, those debug macros were the bomb. I owe you my firstborn child because you saved me a decade of heartache and prevented me from killing myself more than once. Thank you, good sir, here’s a million dollars and the original Snakehead Telecaster prototype signed by Leo Fender.”

Yes, they are that awesome.

The C Error-Handling Problem

Handling errors is a difficult activity in almost every programming language. There are entire programming languages that try as hard as they can to avoid even the concept of an error. Other languages invent complex control structures like exceptions to pass error conditions around. The problem exists mostly because programmers assume errors don’t happen, and this optimism infects the types of languages they use and create.

C tackles the problem by returning error codes and setting a global `errno` value that you check. This makes for complex code that simply exists to check if something you did had an error. As you write more and more C code, you’ll write code with this pattern:

- Call a function.
- Check if the return value is an error (and it must look that up each time, too).
- Then, clean up all the resources created so far.
- Lastly, print out an error message that hopefully helps.

This means for every function call (and yes, *every* function), you are potentially writing three or four more lines just to make sure it worked. That doesn’t include the problem of cleaning up all of the junk you’ve built to that point. If you have ten different structures, three files, and a database connection, you’d have 14 more lines when you get an error.

In the past, this wasn’t a problem because C programs did what you’ve been doing when there was an error: die. No point in bothering with cleanup when the OS will do it for you. Today, though, many C programs need to run for weeks, months, or years, and handle errors from many different sources gracefully. You can't just have your Web server die at the slightest touch, and you definitely can't have a library that you've written nuke the program it's used in. That's just rude.

Other languages solve this problem with exceptions, but those have problems in C (and in other languages, too). In C, you only have one return value, but exceptions make up an entire stack-based return system with arbitrary values. Trying to marshal exceptions up the stack in C is difficult, and no other libraries will understand it.

The Debug Macros

The solution I've been using for years is a small set of debug macros that implements a basic debugging and error-handling system for C. This system is easy to understand, works with every library, and makes C code more solid and clearer.

It does this by adopting the convention that whenever there's an error, your function will jump to an `error:` part of the function that knows how to clean up everything and return an error code. You can use a macro called `check` to check return codes, print an error message, and then jump to the cleanup section. You can combine that with a set of logging functions for printing out useful debug messages.

I'll now show you the entire contents of the most awesome set of brilliance you've ever seen.

`dbg.h`

[Click here to view code image]

```
#ifndef __dbg_h__
#define __dbg_h__

#include <stdio.h>
#include <errno.h>
#include <string.h>

#ifdef NDEBUG
#define debug(M, ...)
#else
#define debug(M, ...) fprintf(stderr, "DEBUG %s:%d: "
M "\n",\
        __FILE__, __LINE__, ##__VA_ARGS__)
#endif

#define clean_errno() (errno == 0 ? "None" :
strerror(errno))

#define log_err(M, ...) fprintf(stderr,\
        "[ERROR] (%s:%d: errno: %s) " M "\n",
__FILE__, __LINE__,\
        clean_errno(), ##__VA_ARGS__)

#define log_warn(M, ...) fprintf(stderr,\
        "[WARN] (%s:%d: errno: %s) " M "\n",\
        __FILE__, __LINE__, clean_errno(),
##__VA_ARGS__)

#define log_info(M, ...) fprintf(stderr, "[INFO]
(%s:%d) " M "\n",\
        __FILE__, __LINE__, ##__VA_ARGS__)

#define check(A, M, ...) if(!(A)) {\
    log_err(M, ##__VA_ARGS__); errno=0; goto error; }

#define sentinel(M, ...) { log_err(M,
##__VA_ARGS__);\
    errno=0; goto error; }

#define check_mem(A) check((A), "Out of memory.")

#define check_debug(A, M, ...) if(!(A)) { debug(M,
##__VA_ARGS__);\
    errno=0; goto error; }

#endif
```

Yes, that's it, and here's a breakdown of every line:

dbg.h:1-2 The usual defense against accidentally including the file twice, which you saw in the last exercise.

dbg.h:4-6 `Include`s for the functions that these macros need.

dbg.h:8 The start of a `#ifdef` that lets you recompile your program so that all of the debug log messages are removed.

dbg.h:9 If you compile with `NDEBUG` defined, then "no debug" messages will remain. You can see in this case the `#define` `debug()` is just replaced with nothing (the right side is empty).

dbg.h:10 The matching `#else` for the above `#ifdef`.

dbg.h:11 The alternative `#define debug` that translates any use of `debug("format", arg1, arg2)` into an `fprintf` call to `stderr`. Many C programmers don't know this, but you can create macros that actually work like `printf` and take variable arguments. Some C compilers (actually CPP) don't support this, but the ones that matter do. The magic here is the use of `##__VA_ARGS__` that says "put whatever they had for extra arguments (...) here." Also notice the use of `__FILE__` and `__LINE__` to get the current `file:line` for the debug message. *Very* helpful.

dbg.h:12 The end of the `#ifdef`.

dbg.h:14 The `clean_errno` macro that's used in the others to get a safe, readable version of `errno`. That strange syntax in the middle is a ternary operator and you'll learn what it does later.

dbg.h:16-20 The `log_err`, `log_warn`, and `log_info`, macros for logging messages that are meant for the end user. They work like `debug` but can't be compiled out.

dbg.h:22 The best macro ever, `check`, will make sure the condition `A` is true, and if not, it logs the error `M` (with variable arguments for `log_err`), and then jumps to the function's `error:` for cleanup.

dbg.h:24 The second best macro ever, `sentinel`, is placed in any part of a function that shouldn't run, and if it does, it prints an error message and then jumps to the `error:` label. You put this in `if-statements` and `switch-statements` to catch conditions that shouldn't happen, like the `default:`.

dbg.h:26 A shorthand macro called `check_mem` that makes sure a pointer is valid, and if it isn't, it reports it as an error with "Out of memory."

dbg.h:28 An alternative macro, `check_debug`, which still checks and handles an error, but if the error is common, then it doesn't bother reporting it. In this one, it will use `debug` instead of `log_err` to report the message. So when you define `NDEBUG`, the check still happens, and the error jump goes off, but the message isn't printed.

Using dbg.h

Here's an example of using all of `dbg.h` in a small program. This doesn't actually do anything but demonstrate how to use each macro. However, we'll be using these macros in all of the programs we write from now on, so be sure to understand how to use them.

`ex19.c`

[Click here to view code image]

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

void test_debug()
{
    // notice you don't need the \n
    debug("I have Brown Hair.");

    // passing in arguments like printf
    debug("I am %d years old.", 37);
}

void test_log_err()
{
    log_err("I believe everything is broken.");
    log_err("There are %d problems in %s.", 0, "space");
}

void test_log_warn()
{
    log_warn("You can safely ignore this.");
    log_warn("Maybe consider looking at: %s.", "/etc/passwd");
}

void test_log_info()
{
    log_info("Well I did something mundane.");
    log_info("It happened %f times today.", 1.3f);
}

int test_check(char *file_name)
{
```

```c
    FILE *input = NULL;
    char *block = NULL;

    block = malloc(100);
    check_mem(block);                       // should work

    input = fopen(file_name, "r");
    check(input, "Failed to open %s.", file_name);

    free(block);
    fclose(input);
    return 0;

error:
    if (block) free(block);
    if (input) fclose(input);
    return -1;
}

int test_sentinel(int code)
{
    char *temp = malloc(100);
    check_mem(temp);

    switch (code) {
        case 1:
            log_info("It worked.");
            break;
        default:
            sentinel("I shouldn't run.");
    }

    free(temp);
    return 0;

error:
    if (temp)
        free(temp);
    return -1;
}

int test_check_mem()
{
    char *test = NULL;
    check_mem(test);

    free(test);
    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

Click here to view code image

```
$ 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:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
#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:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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`.