1.4 Error Handling: Wrapper Functions

In any real-world program, it is essential to check *every* function call for an error return. In Figure 1.5, we check for errors from `socket`, `inet_pton`, `connect`, `read`, and `fputs`, and when one occurs, we call our own functions, `err_quit` and `err_sys`, to print an error message and terminate the program. We find that most of the time, this is what we want to do. Occasionally, we want to do something other than terminate when one of these functions returns an error, as in Figure 5.12, when we must check for an interrupted system call.

Since terminating on an error is the common case, we can shorten our programs by defining a *wrapper function* that performs the actual function call, tests the return value, and terminates on an error. The convention we use is to capitalize the name of the function, as in

```
sockfd = Socket(AF_INET, SOCK_STREAM, 0);
```

Our wrapper function is shown in Figure 1.7.

Figure 1.7 Our wrapper function for the `socket` function.

lib/wrapsock.c

```
236 int
237 Socket(int family, int type, int protocol)
238 {
239     int     n;

240     if ( (n = socket(family, type, protocol)) < 0)
241         err_sys("socket error");
242     return (n);
243 }
```

> *Whenever you encounter a function name in the text that begins with an uppercase letter, that is one of our wrapper functions. It calls a function whose name is the same but begins with the lowercase letter.*
>
> *When describing the source code that is presented in the text, we always refer to the lowest level function being called (e.g., `socket`), not the wrapper function (e.g., `Socket`).*

While these wrapper functions might not seem like a big savings, when we discuss threads in Chapter 26, we will find that thread functions do not set the standard Unix `errno` variable when an error occurs; instead, the `errno` value is the return value of the function. This means that every time we call one of the `pthread_` functions, we must allocate a variable, save the return value in that variable, and then set `errno` to this value before calling `err_sys`. To avoid cluttering the code with braces, we can use C's comma operator to combine the assignment into `errno` and the call of `err_sys` into a single statement, as in the following:

```
int    n;

if ( (n = pthread_mutex_lock(&ndone_mutex)) != 0)
    errno = n, err_sys("pthread_mutex_lock error");
```

Alternately, we could define a new error function that takes the system's error number as an argument. But, we can make this piece of code much easier to read as just

```
Pthread_mutex_lock(&ndone_mutex);
```

by defining our own wrapper function, as shown in Figure 1.8.

Figure 1.8 Our wrapper function for `pthread_mutex_lock`.

lib/wrappthread.c

```
72 void
73 Pthread_mutex_lock(pthread_mutex_t *mptr)
74 {
75     int     n;

76     if ( (n = pthread_mutex_lock(mptr)) == 0)
77         return;
78     errno = n;
79     err_sys("pthread_mutex_lock error");
```

```
80 }
```

With careful C coding, we could use macros instead of functions, providing a little run-time efficiency, but these wrapper functions are rarely the performance bottleneck of a program.

Our choice of capitalizing the first character of a function name is a compromise. Many other styles were considered: prefixing the function name with an "e" (as done on p. 182 of [Kernighan and Pike 1984]), appending "_e" to the function name, and so on. Our style seems the least distracting while still providing a visual indication that some other function is really being called.

This technique has the side benefit of checking for errors from functions whose error returns are often ignored: `close` and `listen`, for example.

Throughout the rest of this book, we will use these wrapper functions unless we need to check for an explicit error and handle it in some way other than terminating the process. We do not show the source code for all our wrapper functions, but the code is freely available (see the Preface).

Unix `errno` Value

When an error occurs in a Unix function (such as one of the socket functions), the global variable `errno` is set to a positive value indicating the type of error and the function normally returns −1. Our `err_sys` function looks at the value of `errno` and prints the corresponding error message string (e.g., "Connection timed out" if `errno` equals `ETIMEDOUT`).

The value of `errno` is set by a function only if an error occurs. Its value is undefined if the function does not return an error. All of the positive error values are constants with all-uppercase names beginning with "`E`," and are normally defined in the `<sys/errno.h>` header. No error has a value of 0.

Storing `errno` in a global variable does not work with multiple threads that share all global variables. We will talk about solutions to this problem in Chapter 26.

Throughout the text, we will use phrases such as "the `connect` function returns `ECONNREFUSED`" as shorthand to mean that the function returns an error (typically with a return value of −1), with `errno` set to the specified constant.