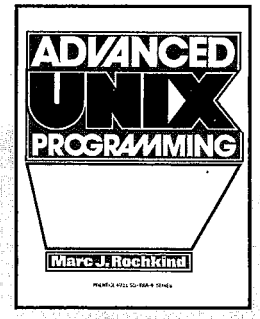
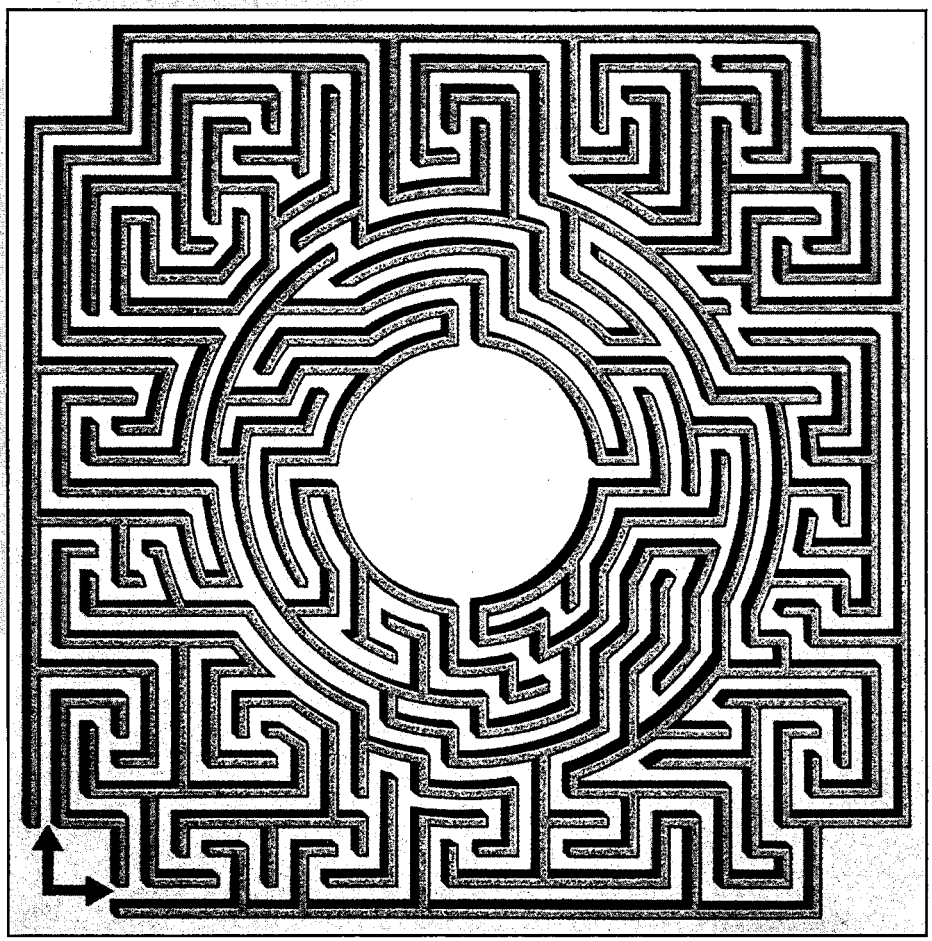
Exhibit D

▶Updated Classic!

Advanced UNIX® Programming



SECOND EDITION



MARC J. ROCHKIND

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

To sort things out, it's not enough to have complete documentation, just as the Yellow Pages isn't enough to find a good restaurant or hotel. You need a guide that tells you what's good and bad, not just what exists. That's the purpose of this book, and why it's different from most other UNIX programming books. I tell you not only how to use the system calls, but also which ones to stay away from because they're unnecessary, obsolete, improperly implemented, or just plain poorly designed.

Here's how I decided what to include in this book: I started with the 1108 functions defined in Version 3 of the Single UNIX Specification and eliminated about 590 Standard C and other library functions that aren't at the kernel-interface level, about 90 POSIX Threads functions (keeping a dozen of the most important), about 25 accounting and logging functions, about 50 tracing functions, about 15 obscure and obsolete functions, and about 40 functions for scheduling and other things that didn't seem to be generally useful. That left exactly 307 for this book. (See Appendix D for a list.) Not that the 307 are all good—some of them are useless, or even dangerous. But those 307 are the ones you need to know.

This book doesn't cover kernel implementation (other than some basics), writing device drivers, C programming (except indirectly), UNIX commands (shell, vi, emacs, etc.), or system administration.

There are nine chapters: Fundamental Concepts, Basic File I/O, Advanced File I/O, Terminal I/O, Processes and Threads, Basic Interprocess Communication, Advanced Interprocess Communication, Networking and Sockets, and Signals and Timers. Read all of Chapter 1, but then feel free to skip around. There are lots of cross-references to keep you from getting lost.

Like the first edition, this new book includes thousands of lines of example code, most of which are from realistic, if simplified, applications such as a shell, a full-screen menu system, a Web server, and a real-time output recorder. The examples are all in C, but I've provided interfaces in Appendices B and C so you can program in C++, Java, or Jython (a variant of Python) if you like.

The text and example code are just resources; you really learn UNIX programming by doing it. To give you something to do, I've included exercises at the end of each chapter. They range in difficulty from questions that can be answered in a few sentences to simple programming problems to semester-long projects.

I used four UNIX systems for nuts-and-bolts research and to test the examples: Solaris 8, SuSE Linux 8 (2.4 kernel), FreeBSD 4.6, and Darwin (the Mac OS X

way to do it. Also, it makes it easier later if you're looking for a bug and are trying to compare the code to the documentation. You don't have to solve a little puzzle in your head to compare the two.

1.4 Error Handling

Testing error returns from system calls is tricky, and handling an error once you discover it is even tricker. This section explains the problem and gives some practical solutions.

1.4.1 Checking for Errors

Most system calls return a value. In the `read` example (Section 1.3.1), the number of bytes read is returned. To indicate an error, a system call usually returns a value that can't be mistaken for valid data, typically –1. Therefore, my example should have been coded something like this:

```
if ((amt = read(fd, buf, numbyte)) == -1) {
    fprintf(stderr, "Read failed!\n");
    exit(EXIT_FAILURE);
}
```

Note that `exit` is a system call too, but it can't return an error because it doesn't return. The symbol `EXIT_FAILURE` is in Standard C.

Of the system calls covered in this book, about 60% return –1 on an error, about 20% return something else, such as `NULL`, zero, or a special symbol like `SIG_ERR`, and about 20% don't report errors at all. So, you just can't assume that they all behave the same way—you have to read the documentation for each call. I'll provide the information when I introduce each system call.

There are lots of reasons why a system call that returns an error indication might have failed. For 80% of them, the integer symbol `errno` contains a code that indicates the reason. To get at `errno` you include the header errno.h. You can use `errno` like an integer, although it's not necessarily an integer variable. If you're using threads, `errno` is likely to involve a function call, because the different threads can't reliably all use the same global variable. So don't declare `errno` yourself (which used to be exactly what you were supposed to do), but use the definition in the header, like this (other headers not shown):

```
#include <errno.h>

if ((amt = read(fd, buf, numbyte)) == -1) {
    fprintf(stderr, "Read failed! errno = %d\n", errno);
    exit(EXIT_FAILURE);
}
```

If, say, the file descriptor is bad, the output would be:

```
Read failed! errno = 9
```

Almost always, you can use the value of `errno` only if you've first checked for an error; you can't just check `errno` to see if an error occurred, because its value is set only when a function that is specified to set it returns an error. So, this code would be wrong:

```
amt = read(fd, buf, numbyte);
if (errno != 0) { /* wrong! */
    fprintf(stderr, "Read failed! errno = %d\n", errno);
    exit(EXIT_FAILURE);
}
```

Setting `errno` to zero first works:

```
errno = 0;
amt = read(fd, buf, numbyte);
if (errno != 0) { /* bad! */
    fprintf(stderr, "Read failed! errno = %d\n", errno);
    exit(EXIT_FAILURE);
}
```

But it's still a bad idea because:

- If you modify the code later to insert another system call, or any function that eventually executes a system call, before the call to `read`, the value of `errno` may be set by *that* call.
- Not all system calls set the value of `errno`, and you should get into the habit of checking for an error that conforms exactly to the function's specification.

Thus, for almost all system calls, check `errno` only after you've established that an error occurred.

Now, having warned you about using `errno` alone to check for an error, this being UNIX, I have to say that there are a few exceptions (e.g., `sysconf` and `readdir`) that do rely on a changed `errno` value to indicate an error, but even they return a specific value that tells you to check `errno`. Therefore, the rule

about not checking `errno` before first checking the return value is a good one, and it applies to most of the exceptions, too.

The `errno` value 9 that was displayed a few paragraphs up doesn't mean much, and isn't standardized, so it's better to use the Standard C function `perror`, like this:

```
if ((amt = read(fd, buf, numbyte)) == -1) {
    perror("Read failed!");
    exit(EXIT_FAILURE);
}
```

Now the output is:

```
Read failed!: Bad file number
```

Another useful Standard C function, `strerror`, just provides the message as a string, without also displaying it like `perror` does.

But the message "Bad file number," while clear enough, isn't standardized either, so there's still a problem: The official documentation for system calls and other functions that use `errno` refer to the various errors with symbols like `EBADF`, not by the text messages. For example, here's an excerpt from the SUS entry for `read`:

> [EAGAIN]
>
> The O_NONBLOCK flag is set for the file descriptor and the process would be delayed.
>
> [EBADF]
>
> The fildes argument is not a valid file descriptor open for reading.
>
> [EBADMSG]
>
> The file is a STREAM file that is set to control-normal mode and the message waiting to be read includes a control part.

It's straightforward to match "Bad file number" to `EBADF`, even though those exact words don't appear in the text, but not for the more obscure errors. What you really want along with the text message is the actual symbol, and there's no Standard C or SUS function that gives it to you. So, we can write our own function that translates the number to the symbol. We built the list of symbols in the code that follows from the errno.h files on Linux, Solaris, and BSD, since many symbols are system specific. You'll probably have to adjust this code for your

own system. For brevity, not all the symbols are shown, but the complete code is on the AUP Web site (Section 1.8 and [AUP2003]).

```
static struct {
    int code;
    char *str;
} errcodes[] =
{
    { EPERM, "EPERM" },
    { ENOENT, "ENOENT" },
      …
    { EINPROGRESS, "EINPROGRESS" },
    { ESTALE, "ESTALE" },
#ifndef BSD
    { ECHRNG, "ECHRNG" },
    { EL2NSYNC, "EL2NSYNC" },
      …
    { ESTRPIPE, "ESTRPIPE" },
    { EDQUOT, "EDQUOT" },
#ifndef SOLARIS
    { EDOTDOT, "EDOTDOT" },
    { EUCLEAN, "EUCLEAN" },
      …
    { ENOMEDIUM, "ENOMEDIUM" },
    { EMEDIUMTYPE, "EMEDIUMTYPE" },
#endif
#endif
    { 0, NULL}
};

const char *errsymbol(int errno_arg)
{
    int i;

    for (i = 0; errcodes[i].str != NULL; i++)
        if (errcodes[i].code == errno_arg)
            return errcodes[i].str;
    return "[UnknownSymbol]";
}
```

Here's the error checking for `read` with the new function:

```
if ((amt = read(fd, buf, numbyte)) == -1) {
    fprintf(stderr, "Read failed!: %s (errno = %d; %s)\n",
      strerror(errno), errno, errsymbol(errno));
    exit(EXIT_FAILURE);
}
```

Now the output is complete:

```
Read failed!: Bad file descriptor (errno = 9; EBADF)
```

It's convenient to write one more utility function to format the error information, so we can use it in some code we're going to write in the next section:

```
char *syserrmsg(char *buf, size_t buf_max, const char *msg, int errno_arg)
{
    char *errmsg;

    if (msg == NULL)
        msg = "???";
    if (errno_arg == 0)
        snprintf(buf, buf_max, "%s", msg);
    else {
        errmsg = strerror(errno_arg);
        snprintf(buf, buf_max, "%s\n\t\t*** %s (%d: \"%s\") ***", msg,
          errsymbol(errno_arg), errno_arg,
          errmsg != NULL ? errmsg : "no message string");
    }
    return buf;
}
```

We would use `syserrmsg` like this:

```
if ((amt = read(fd, buf, numbyte)) == -1) {
    fprintf(stderr, "%s\n", syserrmsg(buf, sizeof(buf),
      "Call to read function failed", errno));
    exit(EXIT_FAILURE);
}
```

with output like this:

```
Call to read function failed
   *** EBADF (9: "Bad file descriptor") ***
```

What about the other 20% of the calls that report an error but don't set `errno`? Well, around 20 of them report the error another way, typically by simply returning the error code (that is, a non-zero return indicates that an error occurred and also what the code is), and the rest don't provide a specific reason at all. I'll provide the details for every function (all 300 or so) in this book. Here's one that returns the error code directly (what this code is supposed to do isn't important right now):

```
struct addrinfo *infop;

if ((r = getaddrinfo("localhost", "80", NULL, &infop)) != 0) {
```

```
    fprintf(stderr, "Got error code %d from getaddrinfo\n", r);
    exit(EXIT_FAILURE);
}
```

The function `getaddrinfo` is one of those that doesn't set `errno`, and you can't pass the error code it returns into `strerror`, because that function works only with `errno` values. The various error codes returned by the non-`errno` functions are defined in [SUS2002] or in your system's manual, and you certainly could write a version of `errsymbol` (shown earlier) for those functions. But what makes this difficult is that the symbols for one function (e.g., `EAI_BADFLAGS` for `getaddrinfo`) aren't specified to have values distinct from the symbols for another function. This means that you can't write a function that takes an error code alone and looks it up, like `errsymbol` did. You have to pass the name of the function in as well. (If you do, you could take advantage of `gai_strerror`, which is a specially tailored version of `strerror` just for `getaddrinfo`.)

There are about two dozen functions in this book for which the standard [SUS2002] doesn't define any `errno` values or even say that `errno` is set, but for which your implementation may set `errno`. The phrase "errno not defined" appears in the function synopses for these.

Starting to get a headache? UNIX error handling is a real mess. This is unfortunate because it's hard to construct test cases to make system calls misbehave so the error handling you've coded can be checked, and the inconsistencies make it hard to get it right every single time. But the chances of it getting any better soon are zero (it's frozen by the standards), so you'll have to live with it. Just be careful!

1.4.2 Error-Checking Convenience Macros for C

It's tedious to put every system call in an `if` statement and follow it with code to display the error message and exit or return. When there's cleanup to do, things get even worse, as in this example:

```
if ((p = malloc(sizeof(buf))) == NULL) {
    fprintf(stderr, "%s\n", syserrmsg(buf, sizeof(buf),
      "malloc failed", errno));
    return false;
}
if ((fdin = open(filein, O_RDONLY)) == -1) {
    fprintf(stderr, "%s\n", syserrmsg(buf, sizeof(buf),
      "open (input) failed", errno));
```

```
        free(p);
        return false;
    }
    if ((fdout = open(fileout, O_WRONLY)) == -1) {
        fprintf(stderr, "%s\n", syserrmsg(buf, sizeof(buf),
          "open (output) failed", errno));
        (void)close(fdin);
        free(p);
        return false;
    }
```

The cleanup code gets longer and longer as we go. It's hard to write, hard to read, and hard to maintain. Many programmers will just use a `goto` so the cleanup code can be written just once. Ordinarily, `goto`s are to be avoided, but here they seem worthwhile. Note that we have to carefully initialize the variables that are involved in cleanup and then test the file-descriptor values so that the cleanup code can execute correctly no matter what the incomplete state.

```
    char *p = NULL;
    int fdin = -1, fdout = -1;

    if ((p = malloc(sizeof(buf))) == NULL) {
        fprintf(stderr, "%s\n", syserrmsg(buf, sizeof(buf),
          "malloc failed", errno));
        goto cleanup;
    }
    if ((fdin = open(filein, O_RDONLY)) == -1) {
        fprintf(stderr, "%s\n", syserrmsg(buf, sizeof(buf),
          "open (input) failed", errno));
        goto cleanup;
    }
    if ((fdout = open(fileout, O_WRONLY)) == -1) {
        fprintf(stderr, "%s\n", syserrmsg(buf, sizeof(buf),
          "open (output) failed", errno));
        goto cleanup;
    }
    return true;

cleanup:
    free(p);
    if (fdin != -1)
        (void)close(fdin);
    if (fdout != -1)
        (void)close(fdout);
    return false;
```

Still, coding all those `if`s, `fprintf`s, and `goto`s is a pain. The system calls themselves are almost buried!

We can streamline the jobs of checking for the error, displaying the error information, and getting out of the function with some macros. I'll first show how they're used and then how they're implemented. (This stuff is just Standard C coding, not especially connected to system calls or even to UNIX, but I've included it because it'll be used in all the subsequent examples in this book.)

Here's the previous example rewritten to use these error-checking ("ec") macros. The context isn't shown, but this code is inside of a function named `fcn`:

```
    char *p = NULL;
    int fdin = -1, fdout = -1;

    ec_null( p = malloc(sizeof(buf)) )
    ec_neg1( fdin = open(filein, O_RDONLY) )
    ec_neg1( fdout = open(fileout, O_WRONLY) )

    return true;

EC_CLEANUP_BGN
    free(p);
    if (fdin != -1)
        (void)close(fdin);
    if (fdout != -1)
        (void)close(fdout);
    return false;
EC_CLEANUP_END
```

Here's the call to the function. Because it's in the `main` function, it makes sense to exit on an error.

```
    ec_false( fcn() )

    /* other stuff here */

    exit(EXIT_SUCCESS);

EC_CLEANUP_BGN
    exit(EXIT_FAILURE);
EC_CLEANUP_END
```

Here's what's going on: The macros `ec_null`, `ec_neg1`, and `ec_false` check their argument expression against `NULL`, –1, and `false`, respectively, store away the error information, and go to a label that was placed by the `EC_CLEANUP_BGN` macro. Then the same cleanup code as before is executed. In `main`, the test of the return value of `fcn` also causes a jump to the same label in `main` and the program exits. A function installed with `atexit` (introduced in Section 1.3.4) displays all the accumulated error information:

```
ERROR:  0: main [/aup/c1/errorhandling.c:41] fcn()
        1: fcn [/aup/c1/errorhandling.c:15] fdin = open(filein, 0x0000)
               *** ENOENT (2: "No such file or directory") ***
```

What we see is the `errno` symbol, value, and descriptive text on the last line. It's preceded by a reverse trace of the error returns. Each trace line shows the level, the name of the function, the file name, the line number, and the code that returned the error indication. This isn't the sort of information you want your end users to see, but during development it's terrific. Later, you can change the macros (we'll see how shortly) to put these details in a log file, and your users can see something more meaningful to them.

We accumulated the error information rather than printing it as we went along because that gives an application developer the most freedom to handle errors as he or she sees fit. It really doesn't do for a function in a library to just write error messages to `stderr`. That may not be the right place, and the wording may not be appropriate for the application's end users. In the end we did print it, true, but that decision can be easily changed if you use these macros in a real application.

So, what these macros give us is:

- Easy and readable error checking
- An automatic jump to cleanup code
- Complete error information along with a back trace

The downside is that the macros have a somewhat strange syntax (no semicolon at the end) and a buried jump in control flow, which some programmers think is a very bad idea. If you think the benefits outweigh the costs, use the macros (as I will in this book). If not, work out your own set (maybe with an explicit `goto` instead of a buried one), or skip them entirely.

Here's most of the header file (ec.h) that implements the error-checking macros (function declarations and a few other minor details are omitted):

```
extern const bool ec_in_cleanup;

typedef enum {EC_ERRNO, EC_EAI} EC_ERRTYPE;

#define EC_CLEANUP_BGN\
    ec_warn();\
    ec_cleanup_bgn:\
    {\
        bool ec_in_cleanup;\
        ec_in_cleanup = true;
```

```
#define EC_CLEANUP_END\
    }

#define ec_cmp(var, errrtn)\
    {\
        assert(!ec_in_cleanup);\
        if ((intptr_t)(var) == (intptr_t)(errrtn)) {\
            ec_push(__func__, __FILE__, __LINE__, #var, errno, EC_ERRNO);\
            goto ec_cleanup_bgn;\
        }\
    }

#define ec_rv(var)\
    {\
        int errrtn;\
        assert(!ec_in_cleanup);\
        if ((errrtn = (var)) != 0) {\
            ec_push(__func__, __FILE__, __LINE__, #var, errrtn, EC_ERRNO);\
            goto ec_cleanup_bgn;\
        }\
    }

#define ec_ai(var)\
    {\
        int errrtn;\
        assert(!ec_in_cleanup);\
        if ((errrtn = (var)) != 0) {\
            ec_push(__func__, __FILE__, __LINE__, #var, errrtn, EC_EAI);\
            goto ec_cleanup_bgn;\
        }\
    }

#define ec_neg1(x) ec_cmp(x, -1)
#define ec_null(x) ec_cmp(x, NULL)
#define ec_false(x) ec_cmp(x, false)
#define ec_eof(x) ec_cmp(x, EOF)
#define ec_nzero(x)\
    {\
        if ((x) != 0)\
            EC_FAIL\
    }

#define EC_FAIL ec_cmp(0, 0)

#define EC_CLEANUP goto ec_cleanup_bgn;
```

```
#define EC_FLUSH(str)\
    {\
        ec_print();\
        ec_reinit();\
    }
```

Before I explain the macros, I have to bring up a problem and talk about its solution. The problem is that if you call one of the error-checking macros (e.g., `ec_neg1`) inside the cleanup code and an error occurs, there will most likely be an infinite loop, since the macro will jump to the cleanup code! Here's what I'm worried about:

```
EC_CLEANUP_BGN
    free(p);
    if (fdin != -1)
        ec_neg1( close(fdin) )
    if (fdout != -1)
        ec_neg1( close(fdout) )
    return false;
EC_CLEANUP_END
```

It looks like the programmer is being very careful to check the error return from `close`, but it's a disaster in the making. What's really bad about this is that the loop would occur only when there was an error cleaning up after an error, a rare situation that's unlikely to be caught during testing. We want to guard against this—the error-checking macros should increase reliability, not reduce it!

Our solution is to set a local variable `ec_in_cleanup` to `true` in the cleanup code, which you can see in the definition for the macro `EC_CLEANUP_BGN`. The test against it is in the macro `ec_cmp`—if it's set, the `assert` will fire and we'll know right away that we goofed.

(The type `bool` and its values, `true` and `false`, are new in C99. If you don't have them, you can just stick the code

```
typedef int bool;
#define true 1
#define false 0
```

in one of your header files.)

To prevent the assertion from firing when `ec_cmp` is called outside of the cleanup code (i.e., a normal call), we have a global variable, also named `ec_in_cleanup`, that's permanently set to `false`. This is a rare case when it's OK (essential, really) to hide a global variable with a local one.

Why have the local variable at all? Why not just set the global to `true` at the start of the cleanup code, and back to `false` at the end? That won't work if you call a function from within the cleanup code that happens to use the `ec_cmp` macro legitimately. It will find the global set to true and think it's in its own cleanup code, which it isn't. So, each function (that is, each unique cleanup-code section) needs a private guard variable.

Now I'll explain the macros one-by-one:

- `EC_CLEANUP_BGN` includes the label for the cleanup code (`ec_cleanup_bgn`), preceded by a function call that just outputs a warning that control flowed into the label. This guards against the common mistake of forgetting to put a `return` statement before the label and flowing into the cleanup code even when there was no error. (I put this in after I wasted an hour looking for an error that wasn't there.) Then there's the local `ec_in_cleanup`, which I already explained.
- `EC_CLEANUP_END` just supplies the closing brace. We needed the braces to create the local context.
- `ec_cmp` does most of the work: Ensuring we're not in cleanup code, checking the error, calling `ec_push` (which I'll get to shortly) to push the location information (`__FILE__`, etc.) onto a stack, and jumping to the cleanup code. The type `intptr_t` is new in C99: It's an integer type guaranteed to be large enough to hold a pointer. If you don't have it yet, `typedef` it to be a `long` and you'll probably be OK. Just to be extra safe, stick some code in your program somewhere to test that `sizeof(void *)` is equal to `sizeof(long)`. (If you're not familiar with the notation `#var`, read up on your C—it makes whatever `var` expands to into a string.)
- `ec_rv` is similar to `ec_cmp`, but it's for functions that return a non-zero error code to indicate an error and which don't use `errno` itself. However, the codes it returns are `errno` values, so they can be passed directly to `ec_push`.
- `ec_ai` is similar to `ec_rv`, but the error codes it deals with aren't `errno` values. The last argument to `ec_push` becomes `EC_EAI` to indicate this. (Only a couple of functions, both in Chapter 8, use this approach.)
- The macros `ec_neg1`, `ec_null`, `ec_false`, and `ec_eof` call `ec_cmp` with the appropriate arguments, and `ec_nzero` does its own checking. They cover the most common cases, and we can just use `ec_cmp` directly for the others.

- `EC_FAIL` is used when an error condition arises from a test that doesn't use the macros in the previous paragraph.
- `EC_CLEANUP` is used when we just want to jump to the cleanup code.
- `EC_FLUSH` is used when we just want to display the error information, without waiting to exit. It's handy in interactive programs that need to keep going. (The argument isn't used.)

The various service functions called from the macros won't be shown here, since they don't illustrate much about UNIX system calls (they just use Standard C), but you can go to the AUP Web site [AUP2003] to see them along with an explanation of how they work. Here's a summary:

- `ec_push` pushes the error and context information passed to it (by the `ec_cmp` macro, say) onto a stack.
- There's a function registered with `atexit` that prints the information on the stack when the program exits:

```
static void ec_atexit_fcn(void)
{
    ec_print();
}
```

- `ec_print` walks down the stack to print the trace and error information.
- `ec_reinit` erases what's on the stack, so error-checking can start with a fresh trace.
- `ec_warn` is called from the `EC_CLEANUP_BGN` code if we accidentally fall into it.

All the functions are thread-safe so they can be used from within multithreaded programs. More on what this means in Section 5.17.

1.4.3 Using C++ Exceptions

Before you spend too much time and energy deciding whether you like the "ec" macros in the previous section and coming up with some improvements, you might ask yourself whether you'll even be programming in C. These days it's much more likely that you'll use C++. Just about everything in this book works fine in a C++ program, after all. C is still often preferred for embedded systems, operating systems (e.g., Linux), compilers, and other relatively low-level software, but those kinds of systems are likely to have their own, highly specialized, error-handling mechanisms.

C++ provides an opportunity to handle errors with exceptions, built into the C++ language, rather than with the combination of `goto`s and `return` statements that

we used in C. Exceptions have their own pitfalls, but if used carefully they're easier to use and more reliable than the "ec" macros, which don't protect you from, say, using `ec_null` when you meant to use `ec_neg1`.

As the library that contains the system-call wrapper code is usually set up just for C, it won't throw exceptions unless someone's made a special version for C++. So, to use exceptions you need another layer of wrapping, something like this for the `close` system call:

```
class syscall_ex {
public:
    int se_errno;

    syscall_ex(int n)
        : se_errno(n)
        { }
    void print(void)
        {
            fprintf(stderr, "ERROR: %s\n", strerror(se_errno));
        }

};

class syscall {
public:
    static int close(int fd)
        {
            int r;
            if ((r = ::close(fd)) == -1)
                throw(syscall_ex(errno));
            return r;
        }
};
```

Then you just call `syscall::close` instead of plain `close`, and it throws an exception on an error. You probably don't want to type in the code for the other 1100 or so UNIX functions, but perhaps just the ones your application uses.

If you want the exception information to include location information such as file and line, you need to define *another* wrapper, this time a macro, to capture the preprocessor data (e.g., via `__LINE__`),[13] so here's an even fancier couple of classes:

13. We need the macro because if you just put `__LINE__` and the others as direct arguments to the `syscall_ex` constructor, you get the location in the definition of `class syscall`, which is the wrong place.

```
class syscall_ex {
public:
    int se_errno;
    const char *se_file;
    int se_line;
    const char *se_func;

    syscall_ex(int n, const char *file, int line, const char *func)
        : se_errno(n), se_file(file), se_line(line), se_func(func)
        { }
    void print(void)
        {
            fprintf(stderr, "ERROR: %s [%s:%d %s()]\n",
              strerror(se_errno), se_file, se_line, se_func);
        }

};

class syscall {
public:
    static int close(int fd, const char *file, int line, const char *func)
        {
            int r;
            if ((r = ::close(fd)) == -1)
                throw(syscall_ex(errno, file, line, func));
            return r;
        }
};

#define Close(fd) (syscall::close(fd, __FILE__, __LINE__, __func__))
```

This time you call `Close` instead of `close`.

You can goose this up with a call-by-call trace, as we did for the "ec" macros if you like, and probably go even further than that.

There's an example C++ wrapper, Ux, for all the system calls in this book that's described in Appendix B.

1.5 UNIX Standards

Practically speaking, what you'll mostly find in the real world is that the commercial UNIX vendors follow the Open Group branding (Section 1.2), and the open-source distributors claim only conformance to POSIX1990, although, with few exceptions, they don't bother to actually run the certification tests. (The tests have now been made freely available, so future certification is now more likely.)