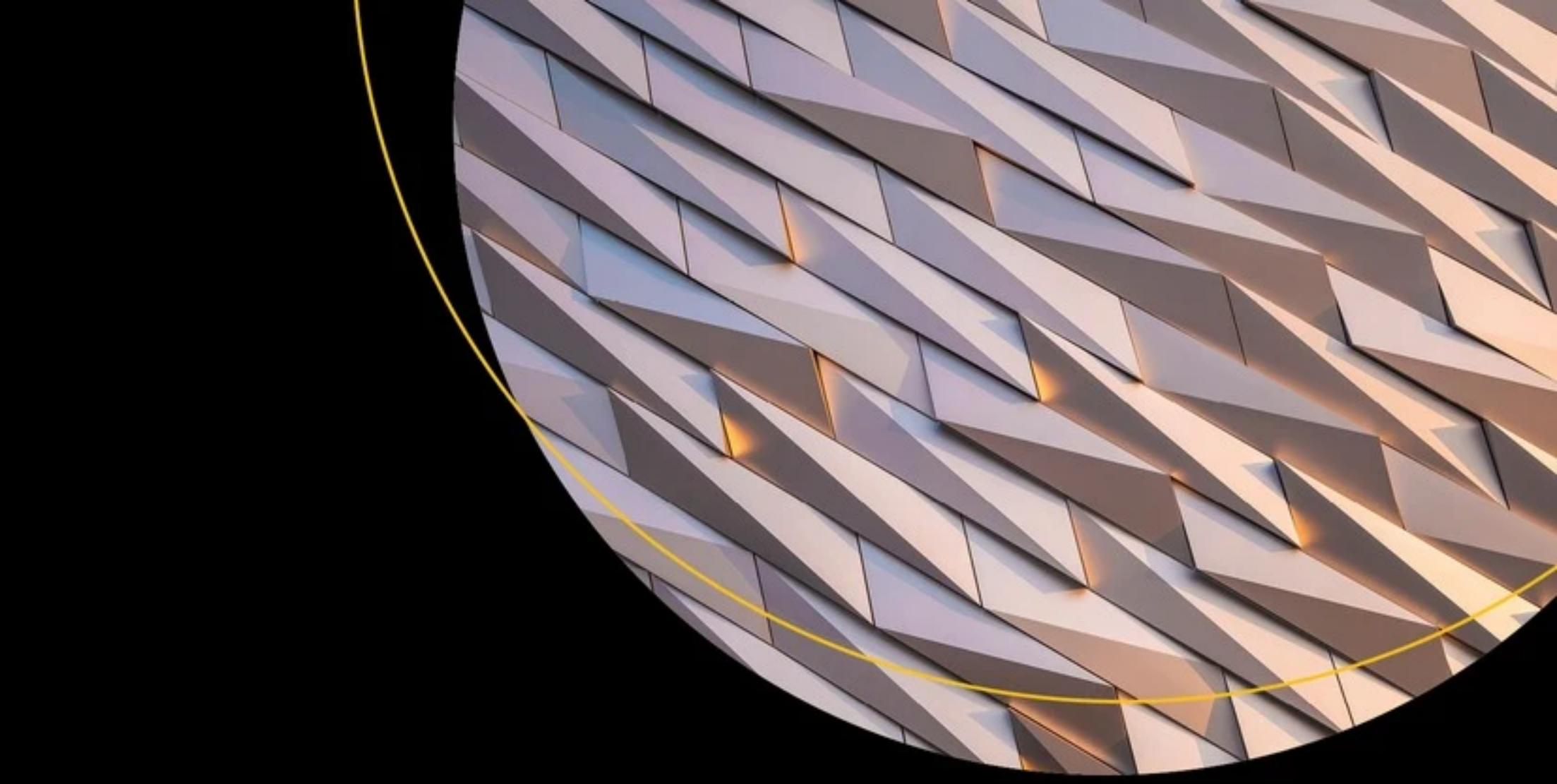


Modern C Up and Running

A Programmer's Guide to Finding Fluency and Bypassing the Quirks

Martin Kalin

Apress®

Modern C Up and Running

A Programmer's Guide to Finding Fluency and Bypassing the Quirks

Martin Kalin

Apress®

Modern C Up and Running: A Programmer's Guide to Finding Fluency and Bypassing the Quirks

Martin Kalin
Chicago, IL, USA

ISBN-13 (pbk): 978-1-4842-8675-3 ISBN-13 (electronic): 978-1-4842-8676-0
https://doi.org/10.1007/978-1-4842-8676-0

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: James Markham
Coordinating Editor: Jill Balzano

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media LLC, 1 New York Plaza, Suite 4600, New York, NY 10004. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub.

Printed on acid-free paper

To Janet, yet again.

Table of Contents

About the Author

Martin Kalin has a Ph.D. from Northwestern University and is a professor in the College of Computing and Digital Media at DePaul University. He has cowritten a series of books on C and C++ and written a book on Java web services. He enjoys commercial programming and has codeveloped, in C, large distributed systems in process scheduling and product configuration. He can be reached at http://condor.depaul.edu/mkalin.

About the Technical Reviewer

Germán González-Morris is a polyglot Software Architect/Engineer with 20+ years in the field, with knowledge in Java, Spring, C, Julia, Python, Haskell, and JavaScript, among others. He works for cloud, web distributed applications. Germán loves math puzzles (including reading Knuth), swimming, and table tennis. Also, he has reviewed several books including an application container book (Weblogic) and some books on languages (Haskell, TypeScript, WebAssembly, Math for coders, regexp, Julia, Algorithms). For more details, visit his blog (https://devwebcl.blogspot.com/) or Twitter account: @devwebcl.

Acknowledgments

My hearty thanks go to the Apress people who made this book possible. I would like to thank, in particular, the following: Steve Anglin, Associate Editorial Director of Acquisitions; Jill Balzano, Coordinating Editor; and James Markham, Development Editor. Thanks as well to the technical reviewers who made the book better than it otherwise would have been.

Preface

1. Why C?

C is a small but extensible language, with software libraries (standard and third party) extending the core language. Among high-level languages, C still sets the mark for performance; hence, C is well suited for applications, especially ones such as database systems and web servers that must perform at a high level. The syntax for C is straightforward, but with an oddity here and there. Anyone who programs in a contemporary high-level language already knows much of C syntax, as other languages have borrowed widely from C.

C is also the dominant *systems language:* modern operating systems are written mostly in C, with assembly language accounting for the rest. Other programming languages routinely and transparently use standard library routines written in C. For example, when an application written in any other high-level language prints the *Hello, world!* greeting, it is a C library function that ultimately writes the message to the screen. The standard system libraries for input/output, networking, string processing, mathematics, security, cryptography, data encoding, and so on are likewise written mainly in C. To write a program in C is to write in the system's native language.

WHO'S THE INTENDED AUDIENCE?

This book is for programmers and assumes experience in a general-purpose language—but none in C. You should be able to work from the command line. Linux and macOS come with a C compiler, typically GNU C (https://gcc.gnu.org) and Clang (https://clang.llvm.org), respectively. At the command-line prompt (% is used here), the command

```
% gcc -v
```

should provide details. For Windows, Cygwin (https://cygwin.com/install.html) is recommended.

C has been a modern language from the start. The familiar *function*, which can take arguments and return a value, is the primary code module in C. C exhibits a separation of concerns by distinguishing between *interfaces*, which describe how functions are called, and *implementations*, which provide the operational details. As noted, C is naturally and easily extended through software libraries, whether standard or third party. As these libraries become better and richer, so does C. C programmers can create arbitrarily rich data types and data structures and package their own code modules as reusable libraries. C supports higher-order functions (functions that can take functions as arguments) without any special, fussy syntax. This book covers C's modern features, but always with an eye on C's close-to-the-metal features.

To understand C is to understand the underlying architecture of a modern computing machine, from an embedded device through a handheld up to a node in a server cluster. C sits atop assembly language, which is symbolic (human-understandable) machine language. Every assembly language is specific to a computer architecture. The assembly language for an Intel device differs from that of an ARM device. Even within an architectural family such as Intel, changes in the architecture are

reflected in assembly language. As symbolic machine language, assembly language is approachable, although reading and writing assembly code can be daunting. Assembly language is of interest even to programmers in other languages because it reveals so much about the underlying system. C does not reveal quite as much, but far more than any other high-level language; C also reveals what is *common* across architectures. One sign of just how close C is to assembly language shows up in compilation: a C compiler can handle any mix of C and assembly source code, and C source is translated first into assembly code. From time to time, it will be useful to compare C source with the assembly source into which the C source translates.

C source code ports well: a C program that compiles on one platform should compile on another, unless platform-specific libraries and data structure sizes come into play. Perfect portability remains an ideal, even for C. C plays still another role—as the *lingua franca* among programming languages: any language that can talk to C can talk to any other language that does so. Most other languages support C calls in one form or another; a later code example shows how straightforwardly Python can consume library functions written in C.

2. From the Basics Through Advanced Features

This book is code centric, with full program examples and shorter code segments in the forefront throughout. The book begins, of course, with C basics: program structure, built-in data types and control structures, operators, pointers, aggregates such as arrays and structures, input and output, and so on. Here is an overview of some advanced topics:

- Memory safety and efficiency: Best practices for using the stack, the heap, and static area of memory; techniques and tools for avoiding memory leakage

- Higher-order functions: Simplifying code by passing functions as arguments to other functions
- Generic functions: How to use the *pointer-to-void* (`void*`) data type in creating and calling generic functions
- Functions with a variable number of arguments: How to write your own
- Defining new data types: A convenient way to name programmer-defined, arbitrarily rich data types
- Clarifying C code through assembly-language code: Getting closer to the metal
- Embedding assembly code: Checking for overflow with embedded assembly
- Floating-point issues: Code examples and the IEEE 754 specification in detail
- Low-level and high-level input/output: Flexibility and performance trade-offs in input/output operations
- Networking and wire-level security: Full code examples, including digital certificates and secure sockets
- Nonblocking input/output: Local machine and networking examples of this acceleration technique
- Concurrency and parallelism: Multiprocessing, interprocess communication, multithreading, deadlock, and instruction-level SIMD parallelism

- Interprocess communication: Pipes (named and unnamed), message queues, sockets, file sharing and locking, shared memory with a semaphore, and signals
- Data validation: Regular expressions in detail
- Internationalization: Standard libraries for locale management
- Assertions: Expressing and enforcing pre-, post-, and invariant conditions in programs
- WebAssembly: Compiling C code into WebAssembly for high-performance web modules
- Software libraries: How to build and deploy both static and dynamic software libraries for C and non-C clients

The code examples in the book are available at `https://github.com/mkalin/cbook.git`, and comments are welcome at mkalin@depaul.edu.

CHAPTER 1

Program Structure

1.1. Overview

This chapter focuses on how C programs are built out of functions, which are a construct in just about all modern program languages. The chapter uses short code segments and full programs to explain topics such as these:

- Functions as program modules
- Control flow within a program
- The special function named `main`
- Passing arguments to a function
- Returning a value from a function
- Writing functions that take a variable number of arguments

C distinguishes between function *declarations*, which show how a function is to be called, and function *definitions*, which provide the implementation detail. This chapter introduces the all-important distinction, and later chapters put the distinction to use in a variety of examples. The chapter also compares C functions with assembly-language blocks, which is helpful in clarifying how C source code compiles into machine-executable code.

M. Kalin, *Modern C Up and Running*, https://doi.org/10.1007/978-1-4842-8676-0_1

Every general-purpose programming language has control structures such as tests and loops. Once again, short code examples introduce the basics of C's principal control structures; later code examples expand and refine this first look at control structures.

1.2. The Function

A C program consists of one or more functions, with a *function* as a program module that takes zero or more arguments and can return a value. To *declare* a function is to describe how the function should be invoked, whereas to *define* a function is to implement it by providing the statements that make up the function's body. A function's body provides the operational details for whatever task the function performs. A declaration is a function's interface, whereas a definition is a function's implementation. The following is an example of the declaration and the definition for a very simple function that takes two integer values as arguments and returns their sum.

Listing 1-1. Declaring and defining a function

```
int add2(int, int); /* declaration ends with semicolon, no body */

int add2(int n1, int n2) { /* definition: the body is enclosed
                              in braces */
   int sum = n1 + n2;      /* could avoid this step, here for
                              clarity */
   return sum;             /* could just return n1 + n2 */
}                          /* end of block that implements the
                              function */
```

The *add2* example (see Listing 1-1) contrasts a function's declaration with its definition. The declaration has no body of statements enclosed in braces, but the definition must have such a body. In a contrived example, the body could be empty, but the braces still would be required in the definition and absent from the declaration.

If some other function `main` calls `add2`, then the declaration of `add2` must be visible to `main`. If the two functions are in the same file, this requirement can be met by declaring `add2` above `main`. There is, however, a shortcut. If `add2` is *defined* above `main` in the same file, then this definition doubles as a declaration (see Listing 1-2).

Listing 1-2. More on declaring and defining a function

```
int add2(int n1, int n2) { /* definition: the body is enclosed
                           in braces */
   int sum = n1 + n2;      /* could avoid this step, here for
                           clarity */
   return sum;             /* could just return n1 + n2 */
}                          /* end of block that implements the
                           function */

int main() {
   return add2(123, 987);  /* ok: add2 is visible to main */
}
```

Program structure may require that a function be declared and defined separately. For instance, if a program's functions are divided among various source files, then a function defined in a given file would have to be *declared* in another file to be visible there. Examples are forthcoming.

As noted, a function's body is enclosed in braces, and each statement within the body ends with a semicolon. Indentation makes source code easier to read but is otherwise insignificant—as is the placement of the braces. My habit is to put the opening brace after the argument list and the closing brace on its own line.

In a program, each function must be defined exactly once and with its own name, which rules out the name *overloading* (popular in languages such as Java) in which different functions share a name but differ in how they are invoked. A function can be declared as often as needed. As promised, an easy way of handling declared functions is forthcoming.

In the current example, the declaration shows that function `add2` takes two integer (`int`) arguments and returns an integer value (likewise an `int`). The definition of function `add2` provides the familiar details, and this definition could be shortened to a single statement:

```
return n1 + n2;
```

If a C function does *not* return a value, then `void` is used in place of a return data type. The term `void`, which is shorthand for *no value*, is technically not a data type in C; for instance, there are no variables of type `void`. By contrast, `int` is a data type. An `int` variable holds a *signed* integer value and so is able to represent negative and nonnegative values alike; the underlying implementation is almost certainly 32 bits in size and almost certainly uses the 2's complement representation, which is clarified later.

There are various C standards, which relax some rules of what might be called *orthodox* C. Furthermore, some C compilers are more forgiving than others. In orthodox C, for example, there are no *nested* function definitions: one function cannot be *defined* inside another. Also, later standardizations of C extend the comment syntax from the slash-star opening and star-slash closing illustrated in Listing 1-1, and an until-end-of-line comment may be introduced with a double slash. To make compilation as simple as possible, my examples stick with orthodox C, avoiding constructs such as nested functions and double slashes for comments.

1.3. The Function `main`

In style, C is a *procedural* or *imperative* language, not an object-oriented or functional one. The program modules in a C program are functions, which have *global* scope or visibility by default. There is a way to restrict a function's scope to the file in which the function is defined, as a later chapter explains. The functions in a C program can be distributed among arbitrarily many different source files, and a given source file can contain as many functions as desired.

A C program's *entry point* is the function `main` in that program execution begins with the first statement in `main`. In a given program, regardless of how many source files there are, the function `main` (like any function) should be defined exactly once. If a collection of C functions does not include the appropriate `main` function, then these functions compile into an *object module*, which can be part of an executable program, but do not, without `main`, constitute an executable program.

Listing 1-3. An executable program with `main` and another function

```
#include <stdio.h>

/* This definition of add2, occurring as it does _above_ main,
   doubles as the function's declaration: main calls add2
   and so the declaration of add2 must be visible above the
   call. If function add2 were _defined_ below main, then the
   function should be declared here above main to avoid
   compiler warnings. */
int add2(int n1, int n2) { /* definition: the body is enclosed
in the braces */
   int sum = n1 + n2;      /* could avoid this step, kept here
                           for verbosity */
   return sum;             /* we could just return n1 + n2 */
}
```

```
int main() {
    int k = -26, m = 44;
    int sum = add2(k, m); /* call the add2 function, save the
                          returned value */
    /* %i means: format as an integer */
    printf("%i + %i = %i\n", k, m, sum); /* output: -26 +
                                        44 = 18 */
    return 0; /* 0 signals normal termination, < 0 signals some
              error */
}
```

The revised *add2* example (see Listing 1-3) can be compiled and then run at the command line as shown in the following, assuming that the file with the two functions is named *add2.c*. These commands are issued in the very directory that holds the source file *add2.c*. My comments begin with two ## symbols:

```
% gcc -o add2 add2.c  ## alternative: % gcc add2.c -o add2
% ./add2              ## On Windows, drop the ./
```

The flag `-o` stands for *output*. Were this flag omitted, the executable would be named *a.out* (*A.exe* on Windows) by default. On some systems, the C compiler may be invoked as `cc` instead of `gcc`. If both commands are available, then `cc` likely invokes a native compiler—a compiler designed specifically for that system. On Unix-like systems, this command typically is a shortcut:

```
% make add2 ## expands to: gcc -o add2 add2.c
```

The *add2* program begins with an *include* directive. Here is the line:

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

This directive is used during the compilation process, with details to follow. The file *stdio.h*, with *h* for *header*, is an *interface* file that *declares* input/output functions such as `printf`, with the `f` for *formatted*. The angle brackets signal that *stdio.h* is located somewhere along the compiler's search path (on Unix-like systems, in a directory such as */usr/include* or */usr/local/include*). The *implementation* of a standard function such as `printf` resides in a binary library (on Unix-like systems, in a directory such as */usr/lib* or */usr/local/lib*), which is linked to the program during the full compilation process.

HEADER FILES FOR FUNCTION DECLARATIONS

Header files are the natural way to handle function declarations—but *not* function definitions. A header file such as *stdio.h* can be included wherever needed, and even multiple includes of the same header file, although inefficient, will work. However, if a header file contains function *definitions*, there is a danger. If such a file were included more than once in a program's source files, this would break the rule that every function must be defined exactly once in a program. The sound practice is to use header files for function declarations, but never for function definitions.

What is the point of having the `main` function return an `int` value? Which function gets the integer value that `main` returns? When the user enters

```
% ./add2
```

at the command-line prompt and then hits the *Return* key, a system function in the *exec* family (e.g., `execv`) executes. This *exec* function then calls the `main` function in the *add2* program, and `main` returns `0` to the *exec* function to signal normal termination (`EXIT_SUCCESS`). Were the *add2*

program to terminate abnormally, the `main` function might return the *negative* value -1 (`EXIT_FAILURE`). The symbolic constants `EXIT_SUCCESS` and `EXIT_FAILURE` are clarified later.

IS THERE EASY-TO-FIND DOCUMENTATION ON LIBRARY FUNCTIONS?

On Unix-like systems, or Windows with Cygwin installed (https://cygwin.com), there is a command-line utility *man* (short for *manual*) that contains documentation for the standard library functions and for utilities that often have the same name as a function: googling for *man pages* is a good start.

1.4. C Functions and Assembly Callable Blocks

The function construct is familiar to any programmer working in a modern language. In object-oriented languages, functions come in special forms such as the *constructor* and the *method*. Many languages, including object-oriented ones, now include *anonymous* or unnamed functions such as the lambdas added in object-oriented languages such as Java and C#, but available in Lisp since the 1950s. C functions are named.

Most languages follow the basic C syntax for functions, with some innovations along the way. The Go language, for example, allows a function to return multiple values explicitly. Functions are straightforward with respect to flow of control: one function calls another, and the called function normally returns to its caller. Information can be sent from the caller to the callee through arguments passed to the callee; information can be sent from the callee back to the caller through a return value. Even in C, which allows only a single return value at the syntax level, multiple values can be returned by returning an *array* or other aggregate structure. Additional tactics for returning multiple values are available, as shown later.

Assembly languages do not have functions in the C sense, although it is now common to talk about *assembly language functions*. The assembly counterpart to the function is the *callable block*, a routine with a label as its identifier; this label is the counterpart of a function's name. Information is passed to a called routine in various ways, but with CPU registers and the stack as the usual way. This section uses the traditional *Hello, world!* program in a first look at (Intel) assembly code.

Listing 1-4. The traditional greeting program in C

```
#include <stdio.h>

int main() {
    /* msg is a pointer to a char, the H in Hello, world! */
    char* msg = "Hello, world!"; /* the string is implemented
                                 as an array of characters */
    printf("%s\n", msg);         /* %s formats the argument as
                                 a string */
    return 0;                    /* main must return an int
                                 value */
}
```

The *hi* program (see Listing 1-4) has three points of interest for comparing C and assembly code. First, the program initializes a variable `msg` whose data type is `char*`, which is a *pointer to a character*. The star could be flush against the data type, in between `char` and `msg`, or flush against `msg`:

```
char* msg  = ...;   /* my preferred style, some limitations */
char * msg = ...;   /* ok, but unusual */
char *msg  = ...;   /* perhaps the most common style */
```

A string in C is implemented as array of `char` values, with the 1-byte, nonprinting character `0` terminating the array:

```
       +---+---+---+---+---+---+   +---+---+---+
msg--->| H | e | l | l | o |...| l | d | \0|  ## \0 is a char
       +---+---+---+---+---+---+   +---+---+---+
```

The slash before the `0` in `\0` identifies an 8-bit (1-byte) representation of zero. A zero without the backslash (`0`) would be an `int` constant, which is typically 32 bits in size. In C, character literals such as `\0` are enclosed in single quotes:

```
char big_A = 'A'; /* 65 in ASCII (and Unicode) */
char nt = '\0';   /* non-printing 0, null terminator for
strings */
```

In the array to which `msg` points, the last character `\0` is called the *null terminator* because its role is to mark where the string ends. As a *nonprinting* character, the null terminator is perfect for the job. Of interest now is how the assembly code represents a string literal.

The second point of interest is the call to the `printf` function. In this version of `printf`, two arguments are passed to the function: the first argument is a format string, which specifies *string* (`%s`) as the formatting type; the second argument is the pointer variable `msg`, which points to the greeting by pointing to the first character *H*. The third and final point of interest is the value 0 (`EXIT_SUCCESS`) that `main` returns to its caller, some function in the *exec* family.

The C code for the *hi* program can be translated into assembly. In this example, the following command was used:

```
% gcc -O1 -S hi.c   ## -O1 = Optimize level 1, -S = save
assembly code
```

The flag `-O1` consists of capital letter O for *optimize* followed by `1`, which is the lowest optimization level. This command produces the output

file *hi.s*, which contains the corresponding assembly code. The file *hi.s* could be compiled in the usual way:

```
% gcc -o hi hi.s    ## produces same output as compiling hi.c
```

Listing 1-5. The hi program in assembly code

```
      .file   "hi.c"              ## C source file
.LC0:                             ## .LC0 is the string's label
                                  (address)
      .string "Hello, world!"     ## string literal
      .text                       ## text (program) area: code,
                                  not data
      .globl  main                ## main is globally visible
      .type   main, @function     ## main is a function, not a
                                  variable (data)
main:                             ## label for main, the
                                  entry point
      .cfi_startproc              ## Call Frame Information:
                                  metadata
      Subq    $8, %rsp            ## grow the stack by 8 bytes
      .cfi_def_cfa_offset 16      ## more metadata
      Movl    $.LC0, %edi         ## copy (pointer to) the
                                  string into register %edi
      Call    puts                ## call puts, which expects
                                  its argument in %edi
      Movl    $0, %eax            ## copy 0 into register %eax,
                                  which holds return value
      Addq    $8, %rsp            ## shrink the stack by 8 bytes
      .cfi_def_cfa_offset 8       ## more metadata
      ret                         ## return to caller
```

```
        .cfi_endproc                    ## all done (metadata)
```

The *hi* program in assembly code (see Listing 1-5) uses AT&T syntax. There are alternatives, including so-called Intel assembly. The AT&T version has advantages, which are explained in the forthcoming discussions. In the example, the ## symbols introduce my comments.

To begin, some points about syntax should be helpful:

- Identifiers that begin with a period (e.g., `.file`) are *directives* that guide the assembler in translating the assembly code into machine-executable code.
- Identifiers that end with a colon (with or without a starting period) are *labels*, which serve as pointers (addresses) to relevant parts of the code. For example, the label `main:` points to the start of the callable code block that, in assembly, corresponds to the `main` function in C.
- CPU registers begin with a percentage sign `%`. In a register name such as `%eax`, the *e* is for *extended*, which means 32 bits in Intel. On a 64-bit machine, the register `%eax` comprises the lower 32 bits of the 64-bit register `%rax`. In general, register names that start with the *e* are the lower 32 bits of the corresponding registers whose names start with *r*: `%eax` and `%rax` are one example, and `%edi` and `%rdi` are another example. A 32-bit machine would have only *e* registers.
- In instructions such as `movl`, the *l* is for *longword*, which is 32 bits in Intel. In instruction `addq`, the *q* is for *quadword*, which is 64 bits. By the way, the various `mov` instructions are actually *copy* instructions: the contents of the source are copied to the destination, but the source remains unchanged.

The essentials of this assembly code example begin with two labels. The first, `.LC0:`, locates the string greeting “Hello, world!”. This label thus

serves the same purpose as the pointer variable `msg` in the C program. The label `main:` locates the program's entry point and, in this way, the callable code block that makes up the body of the `main:` routine.

Two other parts of the `main:` routine deserve a look. The first is the call to the library routine `puts`, where the *s* indicates a *string*. In C code, the call would look like this:

```
puts("This is a string."); /* C code (puts adds a newline) */
```

In C, `puts` would be called with a single argument. In assembly code, however, the `puts` is called *without* an explicit argument. Instead, the expected argument—the address of the string to print—is copied to the register `%edi`, which comprises the lower 32 bits of the 64-bit register `%rdi`. For review, here is the code segment:

```
Movl   $.LC0, %edi   ## copy (pointer to) the string into %edi
Call   puts          ## call puts, which expects argument in %edi
```

A second interesting point about the `main:` routine is the integer value returned to its invoker, again some routine in the *exec* family. The 32-bit register `%eax` (the lower 32 bits of the 64-bit `%rax`) is sometimes used for general-purpose scratchpad, but in this case is used for a special purpose—to hold the value returned from the `main:` routine. The assembly code thus puts 0 in the register immediately before cleaning up the stack and returning:

```
movl    $0, %eax  ## copy 0 into %eax, which holds return value
```

Although assembly-language programs are made up of callable routines rather than functions in the C sense, it is common and, indeed, convenient to talk about assembly *functions*. For the most part, the machine-language library routines originate as C functions that have

been translated first into assembly language and then into machine code (see the sidebar).

HOW ARE C PROGRAMS COMPILED?

The compilation of a C program is a staged process, with four stages:

```
               +----------+    +-------+    +--------+    +----+
source code--->|preprocess|--->|compile|--->|assemble|--->|link|--->machine code
               +----------+    +-------+    +--------+    +----+
```

There are flags for the `gcc` utility, as well as separately named utilities (e.g., *cpp* for *preprocess only*), for carrying out the process only to a particular stage. The *preprocess* stage handles directives such as `#include`, which start with a sharp sign. The *compile* stage generates assembly language code, which the *assemble* stage then translates into machine code. The *link* stage connects the machine code to the appropriate libraries. The command

```
% gcc --save-temps net.c
```

would compile the code but also save the temporary files: *net.i* (text, from preprocess stage), *net.s* (text, from compile stage), and *net.o* (binary, from assemble stage).

1.4.1. A Simpler Program in Assembly Code

A simpler program in assembly language shows that many assembler directives can be omitted; the remaining directives make the code easier to read. Also, no explicit stack manipulation is needed in the forthcoming example, which is written from scratch rather than generated from C source code.

Listing 1-6. A bare-bones program in assembly language

```
## hello program
.data                       # data versus code section
.globl hello                # global scope for label hello
hello:                      # label == symbolic address
  .string "Hello, world!"   # a character string

.text                       # text == code section
.global main                # global scope for main subroutine
main:                       # start of main
  movq   $hello, %rdi      # copy address of the greeting to %rdi
  call   puts               # call library routine puts
  movq   $0, %rax           # copy 0 to %rax (return value)
  ret                       # return control to routine's caller
```

The *hiAssem* program (see Listing 1-6) prints the traditional greeting, but using assembly code rather than C. The program can be compiled and executed in the usual way except for the added flag `-static`:

```
% gcc -o hiAssem -static hiAssem.s
% ./hiAssem    ## on Windows, drop the ./
```

The program structure is straightforward:

1. Identify a string greeting with a label, in this case `hello:`.
2. Identify the entry point with a label, in this case `main:`.
3. Copy the greeting's address `hello:` into register `%rdi`, where the library routine `puts` expects this address.

4. Call `puts`.
5. Copy zero into register `%rax`, which holds a called routine's return value.
6. Return to the caller.

Even the short examples in this section illustrate the basics of C programs: functions in C correspond to *callable blocks* (routines) in assembly language, and in the normal flow of control, a called function returns to its caller. With respect to called functions, the system provides scratchpad storage, for local variables and parameters, with CPU registers and the stack as backup.

1.5. Passing Command-Line Arguments to main

The `main` function seen so far returns an `int` value and takes no arguments. The declaration is

```
int main(); /* one version */
```

The `main` function need not return a value, however:

```
void main(); /* another version, returns nothing */
```

The function `main` also can take arguments from the command line:

```
int main(int argc, char* argv[ ]); /* with two arguments, also
could return void */
```

The two arguments in the last declaration of `main` are named, by tradition, `argc` (*c* for *count*) and `argv` (*v* for *values*). Here is a summary of the information in each argument:

- The first argument to the `main` function is `argc`, a count of the command-line arguments. This count is one or more because the name of the executable program is, again by tradition, the *first* command-line argument. If the program *hi* is invoked from the command line as follows:

  ```
  % ./hi
  ```

  then `argc` would have a value of one. If the same program were invoked as follows:

  ```
  % ./hi one two three
  ```

  then `argc` would have a value of four. A program is not obligated to use the command-line arguments passed to it.

- The second argument (`argv`) passed to `main` is trickier to explain. All of the command-line arguments, including the program's name, are *strings*. Recall that a string in C is an array of characters with a null terminator. Because there may be multiple command-line arguments, these are stored in a list (a C array), each of whose elements holds the address of the first character in a command-line string. For example, in the invocation of program *hi*, the first element in the `argv` array points to the *h* in *hi*; the second element in this array points to the *o* in *one*; and so forth.

  The empty square brackets in `argv[ ]` indicate an array of unspecified length, as the array's length is given in `argc`; the `char*` (*pointer to character*) data type indicates that each array element is a pointer to the first character in each command-line string. The `argv` argument is thus a pointer to an array of

pointers to char; hence, the argv argument is sometimes written as char** argv, which means literally that argv is a *pointer to pointer(s) to characters*.

The details about arrays are covered thoroughly in Chapter 3, but the preceding sketch should be enough to clarify how command-line arguments work in C.

Listing 1-7. Command-line arguments for main

```
#include <stdio.h>

int main(int argc, char* argv[ ]) {
  if (argc < 2) {
    puts("Usage: cline <one or more cmd-line args>");
    return -1; /** -1 is EXIT_FAILURE **/
  }

  puts(argv[0]);   /* executable program's name */

  int i;
  for (i = 1; i < argc; i++)
    puts(argv[i]); /* additional command-line arguments */
  return 0;    /** 0 is EXIT_SUCCESS **/
}
```

The *cline* program (see Listing 1-7) first checks whether there are at least two command-line arguments—at least one in addition to the program's name. If not, the *usage* section introduced by the if clause explains how the program should be run. Otherwise, the program uses the library function puts (*put string*) to print the program's name (argv[0]) and the other command-line argument(s). (The for loop used in the program is clarified in the next section.) Here is a sample run:

```
% ./cline A 1 B2
```

```
./cline
A
1
B2
```

Later examples put the command-line arguments to use. The point for now is that even `main` can have arguments passed to it. Both of the *control structures* used in this program, the `if` test and the `for` loop, now need clarification.

1.6. Control Structures

A *block* is a group of expressions (e.g., integer values to initialize an array) or statements (e.g., the body of a loop). In either case, a block starts with the left curly brace { and ends with a matching right curly brace }. Blocks can be nested to any level, and the *body* of a function—its definition—is a block. Within a block of statements, the default flow of control is *straight-line execution*.

Listing 1-8. Default flow of control

```
#include <stdio.h>

int main() {
  int n = 27;                              /** 1 **/
  int k = 43;                              /** 2 **/
  printf("%i * %i = %i\n", n, k, n * k);   /** 3 **/
  return 0;                                /** 4 **/
}
```

The *straight-line* program (see Listing 1-8) consists of the single function `main`, whose body has four statements, labeled in the comments for reference. There are no tests, loops, or function calls that interfere

with the straight-line execution: first statement 1, then statement 2, then statement 3, and then statement 4. The last statement exits `main` and thereby effectively ends the program's execution. Straight-line execution is fast, but program logic typically requires a more nuanced flow of control.

C has various flavors of the expected control structures, which can be grouped for convenience into three categories: *tests*, *loops*, and (function) *calls*. This section covers the first two, tests and loops; the following section expands on flow of control in function calls.

Listing 1-9. Various ways to test in C

```
#include <stdio.h>

int main() {
  int n = 111, k = 98;

  int r = (n > k) ? k + 1 : n - 1;  /* conditional operator */
  printf("r's value is %i\n", r);   /* 99 */

  if (n < k) puts("if");
  else if (r > k) puts("else if");   /** prints **/
  else puts("else");

  r = 0;               /* reset r to zero */
  switch (r) {
  case 0:
    puts("case 0");    /** prints **/
  case 1:
    puts("case 1");    /** prints **/
    break;             /** break out of switch construct **/
  case 2:
    puts("case 2");
    break;
```

```
  case 3:
    puts("case 3");
    break;
  default:
    puts("none of the above");
  } /* end of switch */
}
```

The *tests* program (see Listing 1-9) shows three ways in which to test in a C program. The first way uses the *conditional operator* in an assignment statement. The conditional expression has three parts:

```
(test) ? if-test-is-true : if-test-is-false  ## true is non-
zero, false is zero
```

In this example, the conditional expression is used as source in an assignment:

```
int r = (n > k) ? k + 1 : n - 1;  /* n is 111, k is 98 */
```

A conditional expression consists of a *test*, which yields one of two values: one value if the test is *true* and another if the test is *false*. The *test* evaluates to *true* (nonzero in C, with a default of 1) because `n` is 111 and `k` is 98, making the expression `(n > k)` true; hence, variable `r` is assigned the value of the expression immediately to the right of the question mark, `k + 1` or 99. Otherwise, variable `r` would be assigned the value of the expression immediately to the right of colon, in this case 110. The expressions after the question mark and the colon could themselves be conditional expressions, but readability quickly suffers.

The conditional operator is convenient and is used commonly to assign a value to a variable or to return a value from a function. This operator also highlights a general rule in C syntax: tests are enclosed in

parentheses, in this example, `(n > k)`. The same syntax applies to if-tests and to loop-tests. Parentheses always can be used to enhance readability, as later examples emphasize, but parentheses are required for test expressions.

The middle part of the *tests* program introduces the syntax for *if-else* constructs, which can be nested to any level. For instance, the body of an `else` clause could itself contain an `if else` construct. In an `if` and an `else if` clause, the test is enclosed in parentheses. There can be an `if` without either an `else if` or an `else`, but any `else` clause must be tied to a prior `if` or `else if`, and every `else if` must be tied to an `if`. In this example, the conditions and results (in this case, `puts` calls) are on the same line. Here is a more readable version:

```
if (n < k)
  puts("if");
else if (r > k)
  puts("else if");   /** prints **/
else
  puts("else");
```

In this example, the body of the `if`, the `else if`, and the `else` is a *single* statement; hence, braces are not needed. The bodies are indented for readability, but indentation has no impact on flow of control. If a body has more than one statement, the body must be enclosed in braces:

```
if (n < k) {              /* braces needed here */
  puts("if");
  puts("just demoing");
}
```

Using braces to enclose even a single body statement is admirable but rare.

The last section of the *tests* program introduces the `switch` construct, which should be used with caution. The `switch` expression, in this case

the value of variable `r`, is enclosed as usual in parentheses. The value of `r` now determines the flow of control. Four `case` clauses are listed, together with an optional `default` at the end. The value of `r` is zero, which means control moves to `case 0` and the `puts` statement is executed. However, there is no `break` statement after this `puts` statement—and so control continues *through* the next case, in this example `case 1`; hence, the second `puts` statement executes. If the value of `r` happened to be 2, only one `puts` statement would execute because the `case 2` body consists of the `puts` statement followed by a `break` statement.

The body of a `case` statement can consist of arbitrarily many statements. The critical point is this: once control enters a `case` construct, the flow is sequential until either a `break` is encountered or the `switch` construct itself is exited. In effect, the `case` expressions are targets for a high-level *goto*, and control continues straight line until there is a `break` or the end of the `switch`.

The `break` statement can be used to break out of a `switch` construct, or out of a loop. The discussion now turns to loops.

C has three looping constructs: `while`, `do while`, and `for`. Any one of the three looping constructs is sufficient to implement program logic, but each type of loop has its natural uses. For instance, a *counted loop* that needs to iterate a specified number of times could be implemented as `while` loop, but a `for` loop readily fits this bill. A *conditional loop* that iterates until a specified condition fails to hold is implemented naturally as a `while` or a `do while` loop.

The general form of a `while` loop is

```
while (<condition>) {
  /* body */
}
```

If the *condition* is *true* (nonzero), the body executes, after which the *condition* is tested again. If the *condition* is *false* (zero), control jumps to the first statement beyond the loop's body. (If the loop's body consists of

a *single* statement, the body need not be enclosed in parentheses.) The `do while` construct is similar, except that the loop *condition* occurs at the *end* rather than at the beginning of a loop; hence, the body of a `do while` loop executes at least once. The general form is

```
do {
  /* body */
} while (<condition>);
```

The `break` statement in C breaks out of a *single* loop. Consider this code segment:

```
while (someCondition) {          /* loop 1 */
   while (anotherCondition) {    /* loop 2 */
     /* ... */
     if (thisHappens) break;     /* breaks out of loop2, but
                                 not loop1 */
   }
   /* ... */
}
```

The `break` statement in *loop2* breaks out of this loop only, and control resumes within *loop1*. C does have `goto` statement whose target is a label, but this control construct should be mentioned just once and avoided thereafter.

Listing 1-10. The `while` and `do while` loops

```
#include <stdio.h>

int main() {
  int n = -1;
  while (1) {  /* 1 == true */
    printf("A non-negative integer, please: ");
    scanf("%i", &n);
```

```
    if (n > 0) break;  /* break out of the loop */
  }

  printf("n is %i\n", n);
  n = -1;
  do {
     printf("A non-negative integer, please: ");
     scanf("%i", &n);
  } while (n < 0);
  printf("n is %i\n", n);
  return 0;
}
```

The *whiling* program (see Listing 1-10) prompts the user for a nonnegative integer and then prints its value. The program does not otherwise validate the input but rather assumes that only decimal numerals and, perhaps, the minus sign are entered. The focus is on contrasting a `while` and a `do while` for the same task.

The condition for the `while` loop is 1, the default value for *true*:

```
while (1) {  /* 1 == true */
```

This loop might be an *infinite* one except that there is a `break` statement, which exits the loop: if the user enters a nonnegative integer, the `break` executes.

The `do while` loop is better suited for the task at hand: first, the user enters a value, and only then does the loop condition test whether the value is greater than zero; if so, the loop exits. In both loops, the `scanf` function is used to read user input. The details about `scanf` and its close relatives can wait until later.

Among the looping constructs, the `for` loop has the most complicated syntax. Its general form is

```
for (<init>;<condition>;<post-body>) {
   /* body */
}
```

A common example is

```
for (i = 0; i < limit; i = i + 1) { /* int i, limit = 100; from
above */
  /* body */
}
```

The *init* section executes exactly once, before anything else. Then the *condition* is evaluated: if *true*, the loop's body is executed; otherwise, control goes to the first statement beyond the loop's body. The *post-body* expression is evaluated per iteration *after* the body executes; then the *condition* is evaluated again; and so on. Any part of the `for` loop can be empty. The construct

```
for (;;) {  /* huh? */ }
```

is an obfuscated version of a potentially *infinite* loop. As shown earlier, a more readable way to write such a loop is

```
while (1) { /** clearer **/ }
```

1.7. Normal Flow of Control in Function Calls

A called function usually returns to its caller. If a called function returns a value, the function has a `return` statement that both returns the value and marks the end of the function's execution: control returns to the caller at the point immediately beyond the call. A function with `void` instead of a return type might contain a `return` statement, but without a value; if not,

the function returns after executing the last statement in the block that makes up the function's body.

The normal *return-to-caller* behavior takes advantage of how modern systems provide scratchpad for called functions. This scratchpad is a mix of general-purpose CPU registers and stack storage. As functions are called, the *call frames* on the stack are allocated automatically; as functions return, these *call frames* can be freed up for future use. The underlying system bookkeeping is simple, and the mechanism itself is efficient in that registers and stack call frames are reused across consecutive function calls.

Example 1-1. Normal calls and returns for functions

```
#include <stdio.h>
#include <stdlib.h>  /* rand() */

int g() {
  return rand() %  100; /* % is modulus; hence, a number 0
                           through 99 */
}

int f(int multiplier) {
  int t = g();
  return t * multiplier;
}

int main() {
  int n = 72;
  int r = f(n);
  printf("Calling f with %i resulted in %i.\n", n, r);
  /* 5976 on sample run */
  return r; /* not usual, but all that's required is a
  returned int */
}
```

The *calling* program (see Example 1-1) illustrates the basics of normal return-to-caller behavior. When the *calling* program is launched from the command line, recall that a system function in the *exec* family invokes the *calling* program's `main` function. In this example, `main` then calls function `f` with an `int` argument, which function `f` uses a multiplier. The number to be multiplied comes from function g, which `f` calls. Function g, in turn, invokes the library function `rand`, which returns a pseudorandomly generated integer value. Here is a summary of the calls and returns, which seem so natural in modern programming languages:

```
              calls         calls         calls      calls
exec-function------->main()------->f(int)------->g()------->rand()

exec-function<-------main()<-------f(int)<-------g()<-------rand()
              returns       returns       returns    returns
```

Further examples flesh out the details in the return-to-caller pattern. One such example analyzes the assembly code in the pattern. A later example looks at abnormal flow of control through *signals*, which can interrupt an executing program and thereby disrupt the normal pattern.

1.8. Functions with a Variable Number of Arguments

The by-now-familiar `printf` function takes a variable number of arguments. Here is its declaration:

```
int printf(const char* format, ...); /* returns number of
                                        characters printed */
```

The first argument is the *format string*, and the optional remaining arguments—represented by the ellipsis—are the values to be formatted. The `printf` function requires the first argument, but the number of additional arguments depends on the number of values to be formatted. There are many other library functions that take a variable number of arguments, and programmer-defined functions can do the same. Two examples illustrate.

Example 1-2. The library function `syscall`

```
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>

int main() {
  /* 0755: owner has read/write/execute permissions, others
  read/execute permissions */
  int perms = 0755; /* 0 indicates base-8, octal */
  int status = syscall(SYS_chmod, "/usr/local/website", perms);
  if (-1 == status) perror(NULL);
  return 0;
}
```

The *sysCall* program (see Example 1-2) invokes the library function `syscall`, which takes a variable number of arguments; the first argument, in this case the symbolic constant `SYS_chmod`, is required. `SYS_chmod` is clarified shortly.

The `syscall` function is an indirect way to make *system calls*, that is, to invoke functions that execute in *kernel space*, the address space reserved for those privileged operating system routines that manage shared system resources: processors, memory, and input/output devices. This indirect approach allows for fine-tuning that the direct approach might not provide. This example is contrived in that the function `chmod` (*change*

mode) could be called directly with the same effect. The *mode* refers to various permissions (e.g., *read* and *write* permissions) on the target, in this case a directory on the local file system.

As noted, the first argument to `syscall` is required. The argument is an integer value that identifies the system function to call. In this case, the argument is `SYS_chmod`, which is defined as 90 in the header file *syscall.h* and identifies the system function `chmod`. The variable arguments to function `syscall` are as follows:

- The path to the file whose mode is to be changed, in this case */usr/local/website*. The path is given as a string. (The directory */usr/local/website* must exist for the program to work, and this directory must be accessible to whoever runs the program.)
- The file permissions, in this case 0777 (base-8): everyone can read/write/execute.

The header file *stdarg.h* has a data type `va_list` (*list of variable arguments*) together with utilities to help programmers write functions with a variable number of arguments. These utilities allocate and deallocate storage for the variable arguments, support iteration over these arguments, and convert each argument to whatever data type is appropriate. The utilities are well designed and worth using. As a popular illustration of a function with a variable number of arguments, the next code example sums up and then averages the arguments. In the example, the required argument and the others happen to be of the same data type, in the current case `int`, but this is not a requirement. Recall again the `printf` function, whose first argument is a string but whose optional, variable arguments all could be of different types.

Example 1-3. A function with a variable number of arguments

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

```
#include <stdarg.h> /* va_list type, va_start va_arg va_end
                    utilities */

double avg(int count, ...) { /* count is how many, ellipses are
                             the other args */
  double sum = 0.0;
  va_list args;
  va_start(args, count); /* allocate storage for the
                         additional args */

  int i;
  for (i = 0; i < count; i++) sum += va_arg(args, int);
/* compute the running sum */
  va_end(args);          /* deallocate the storage for
                         the list */

  if (count > 0) return sum / count;    /* compiler promotes
                                        count to double */
  else return 0;
}

void main() {
  printf("%f\n", avg(4, 1, 2, 3, 4));
  printf("%f\n", avg(9, 9, 8, 7, 6, 5, 4, 3, 2, 1));
  printf("%f\n", avg(0));
}
```

The *varArgs* program (see Example 1-3) defines a function avg with one named argument count and then an ellipsis that represents the variable number of other arguments. In this example, the int parameter count is a placeholder for the *required* argument, which specifies how many other arguments there are. In the first call from main to the function avg, the first 4 in the list become count, and the remaining four values make up the variable arguments.

In the function avg, local variable nums is declared to be of type va_list. The utility va_start is called with args as its first argument and count as its second. The effect is to provide storage for the variable arguments. The later call to va_end signals that this storage no longer is needed. Between the two calls, the va_arg utility is used to extract from the list one int value at a time. The programmer needs to specify, in the second argument to va_arg, the data type of the variable arguments. In this example, the type is the same throughout: int. In a richer example, however, the type could vary from one argument to the next. Finally, function main makes three calls to function avg, including a call that has no arguments other than the required one, which is 0.

1.9. What's Next?

C has basic or primitive data types such as char (8 bits), int (typically 32 bits), float (typically 32 bits), and double (typically 64 bits) together with mechanisms to create arbitrarily rich, programmer-defined types such as Employee and digital_certificate. Names for the primitive types are in lowercase. Data type names, like identifiers in general, start with a letter or an underscore, and the names can contain any mix of uppercase and lowercase characters together with decimal numerals. Most modern languages have naming conventions similar to those in C. The basic types in C deliberately match the ones on the underlying system, which is one way that C serves as a portable assembly language. The next chapter focuses on data types, built-in and programmer-defined.

CHAPTER 2

Basic Data Types

2.1. Overview

C requires explicit data typing for variables, arguments passed to a function, and a value returned from a function. The names for C data types occur in many other languages as well: `int` for signed integers, `float` for floating-point numbers, `char` for numeric values that serve as character codes, and so on. C programmers can define arbitrarily rich data types of their own such as `Employee` and `Movie`, which reduce ultimately to primitive types such as `int` and `float`. C's built-in data types deliberately mirror machine-level types such as integers and floating-point numbers of various sizes.

At a technical level, a data type such as `int`, `float`, `char`, or `Employee` determines

- The amount of memory required to store values of the type (e.g., the `int` value -3232, a pointer to the string "ABC")
- The operations allowed on values of type (e.g., an `int` value can be shifted left or right, whereas a `float` value should not be shifted at all)

M. Kalin, *Modern C Up and Running*, https://doi.org/10.1007/978-1-4842-8676-0_2

The sizeof operator gives the size in bytes for any data type or value of that type. Here is a code segment to illustrate:

```
printf("%lu\n", sizeof(char));  /* 1 (%lu... for long
                                   unsigned) */
printf("%lu %lu\n", sizeof(float), sizeof(99));  /* 4, 4 */
```

The sizeof(char) is required to be 1, which accommodates 7-bit and 8-bit character encodings such as ASCII and Latin-1, respectively. C also has a wchar_t type (w for *wide*), which is 4 bytes in size and designed for multibyte character codes such as Unicode. Types other than char, such as int and float, must be at least sizeof(char) but typically are greater. On a modern handheld device or desktop computer, for example, sizeof(int) and sizeof(float) are 4 bytes apiece.

Listing 2-1. The sizeof various basic data types

```
#include <stdio.h>
#include <wchar.h> /* wchar_t type */

void main() {
  printf("char size:        %lu\n", sizeof(char));
                            /* 1 (long unsigned) */
  printf("wchar_t size:     %lu\n", sizeof(wchar_t)); /* 4 */

  /* Signed and unsigned variants of each type are of same
  size. */
  printf("short size:       %lu\n", sizeof(short));     /* 2 */
  printf("int size:         %lu\n", sizeof(int));       /* 4 */
  printf("long size:        %lu\n", sizeof(long));      /* 8 */
  printf("long long size:   %lu\n", sizeof(long long)); /* 8,
                                                        maybe
                                                        more */
```

```
  /* floating point types are all signed */
  printf("float size:     %lu\n", sizeof(float));      /*  4 */
  printf("double size:     %lu\n", sizeof(double));    /*  8 */
  printf("long double size: %lu\n", sizeof(long double)); /* 16 */
}
```

The *dataTypes* (see Listing 2-1) program prints the byte sizes for the basic C data types. These sizes are the usual ones on modern devices. The following sections focus on C's built-in data types and built-in operations on these types. Technical matters such as the 2's complement representation of integers and the IEEE 754 standard for floating-point formats is covered in detail.

2.2. Integer Types

All of C's integer types come in *signed* and *unsigned* flavors. The unsigned types have a one-field implementation: all of the bits are magnitude bits. By contrast, signed types have a two-field implementation:

- The most significant (by convention, the leftmost) bit is the sign bit, with 0 for nonnegative and 1 for negative.
- The remaining bits are magnitude bits.

The signed and unsigned integer types come in various sizes.

Table 2-1. *Basic integer data types*

Type	Byte size	Range
unsigned char	1	0 to 255
signed char	1	-128 to 127
unsigned short	2	0 to 65,535
signed short	2	-32,768 to 32,767
unsigned int	4	0 to 4,294,967,295
signed int	4	-2,147,483,648 to 2,147,483,647
unsigned long	8	0 to 18,446,744,073,709,551,615
signed long	8	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Table 2-1 lists the basic integer types in C, which have the very bit sizes as their machine-level counterparts. C also has a `long long` type, which must be at least 8 bytes in size and typically is the same size as `long`: 8.

C does *not* have a distinct *boolean* type but instead uses integer values to represent *true* and *false*: 0 represents *false*, and any nonzero value (e.g., -999 and 403) represents *true*. The default value for *true* is 1. For example, a potentially infinite loop might start out like this:

```
while (1) { /** 1 is true in boolean context **/
```

In C source code, an integer constant such as `22` defaults to type `int`, where `int` is shorthand for `signed int`. The constant `22L` or `22l` is of type `long`. Here are some quick examples of data type shorthands:

```
int n;          /* short for: signed int n; */
signed m;       /* short for: signed int m; */
unsigned k;     /* short for: unsigned int k; */
short s;        /* short for: signed short s; */
signed short t; /* the full type written out */
```

As the examples indicate, unsigned must be used explicitly if *unsigned* is the desired variant.

The type of a variable does not restrict the bits that can be stored in it, which means that even everyday C can be obfuscating. An example may be useful here.

Listing 2-2. Data types and bits

```
#include <stdio.h>
#include <limits.h> /* includes convenient min/max values for
                    integer types */

void main() { /* void instead of int for some variety */
  unsigned int n = -1, m = UINT_MAX;  /* In 2's complement, -1
                                      is all 1s */
  signed int k = 0xffffffff;          /* 0x or 0X for hex: f =
                                      4 1s in hex */

  if (n == m) printf("m and n have the same value\n");
  /* prints */
  if (k == m) printf("m and k have the same value\n");
  /* prints */

  printf("small as signed == %i,  small as unsigned == %u\n",
     n, n); /* -1, 4294967295 */

  signed int small = -1;    /* signed converts to unsigned in
                            mixed comparisons */
  unsigned int big = 98765; /* comparing big and small is a
                            mixed comparison */
  if (small > big) printf("yep, something's up...\n");
  /** small value is UINT_MAX **/
}
```

The *obfusc* program (see Listing 2-2) is a cautionary tale on the distinction between *internal* (machine-level) and *external* (human-level) representation. The example's important points can be summed up as follows:

- The data type of a variable does *not* restrict the bits that can be assigned to it. For example, the compiler does not warn against assigning the negative value -1 to the unsigned variable `n`. For the compiler, the decimal value -1 is, in the 2's complement representation now common across computing devices, all 1s in binary. Accordingly, the variable `n` holds 32 1s when -1 is assigned to this variable. (Further details of the 2's complement representation are covered shortly.)
- The equality operator `==`, when applied to integer values, checks for *identical* bit patterns. If the left and the right side expressions (in this example, the values of two variables) have identical bit patterns, the comparison is *true*; otherwise, *false*. The variables `n`, `m`, and `k` all store 32 1s in binary; hence, they are all equal in value by the equality operator `==`.
- In print statements, the *internal representation* of a value (the bit string) can be formatted to yield different *external representations*. For example, the 32 1s stored in variable `n` can be printed as a negative decimal value using the formatter `%i` (*integer*) or `%d` (*decimal*). Recall that in 2's complement, a value is negative if its high-order (leftmost) bit is a 1; hence, the `%i` formatter for *signed* values treats the 32 1s as the negative value -1: the high-order bit is the sign bit 1 (*negative*), and the

remaining bits are the magnitude bits. By contrast, the %u formatter for *unsigned* treats all of the bits as *magnitude* bits, which yields the value of the symbolic constant UINT_MAX (4,294,967,295) in decimal.

- Comparing expressions of *mixed* data types is risky because the compiler coerces one of the types to the other, following rules that may not be obvious. In this example, the value -1 stored in the signed variable small is converted to unsigned so that the comparison is apple to apple rather than apple to orange. As noted earlier, -1 is all 1s in binary; hence, as unsigned, this value is UNIT_MAX, far greater than the 98,765 stored in big.

In mixed integer comparisons, the compiler follows two general rules:

- Signed values are converted to unsigned ones.
- Smaller value types are converted to larger ones. For example, if a 2-byte short is compared to a 4-byte int, then the short value is converted to an int value for the comparison.

When floating-point values occur in expressions with integer values, the compiler converts the integer values into floating-point ones.

In assembly code, an instruction such as cmpl would be used to compare two integer values. The l in cmpl determines the number of bits compared: in this case, 32 because l is for *longword*, a 32-bit word in the Intel architecture. Were two 64-bit values being compared, then the instruction would be cmpq instead, as the q stands for *quadword*, a 64-bit word in this same architecture. At the assembly level, as at the machine level, the size of a data type is built into the instruction's opcode, in this example cmpl.

An earlier example showed that C's `signed char` and `unsigned char` are likewise integer types. As the name `char` indicates, the `char` type is designed to store single-byte character codes (e.g., ASCII and Latin-1); the more recent `wchar_t` type also is an integer type, but one designed for multibyte character codes (e.g., Unicode). For historical reasons, the `char` type is shorthand for either `signed char` or `unsigned char`, but which is platform dependent. For the remaining types, this is *not* the case. For example, `short` is definitely an abbreviation for `signed short`.

2.2.1. A Caution on the 2's Complement Representation

The 2's complement representation of signed integers has a surprising but well-publicized peculiarity. The header file *limits.h* provides the constant `INT_MIN`, the minimum value for a 4-byte `signed int` value. The binary representation, with the most significant bits on the left, is

```
10000000 00000000 00000000 00000000  /* INT_MIN in binary */
```

For readability, the binary representation has been broken into four 8-bit chunks. The rightmost (least significant) bit is a 0, which makes the value (-2,147,483,648) even rather than odd. The leftmost (most significant) bit is the sign bit: `1` for *negative* as in this case and `0` for nonnegative. There are similar constants for other integer types (for instance, `SHRT_MIN` and `LONG_MIN`).

There is a straightforward algorithm for computing the absolute value of a negative 2's complement value. For example, recall that the -1 in binary, under the 2's complement representation, is all 1s: 1111…1. Here is the recipe for computing the absolute value in binary:

1. Invert the 1s in -1, which yields all 0s: 00000…000.
2. Add 1, which yields 00000…001 or 1 in binary and decimal, the absolute value of -1 in decimal.

The same recipe yields -1 from 1: invert the bits in 1 (yielding 11111…0) and then add 1 (yielding 11111…1), which again is all 1s in binary and -1 in decimal.

In the case of `INT_MIN`, the peculiarity becomes obvious:

1. Invert the bits, which transforms `INT_MIN` to `01111111 11111111 11111111 11111111`.
2. Add 1 to yield `10000000 00000000 00000000 00000000`, which is `INT_MIN` again.

In C, the unary minus operator is shorthand for (a) inverting the bits and (b) adding 1. This code segment illustrates

```
int n = 7;
int k = -n;      /* unary-minus operator */
int m = ~n + 1;  /* complement operator and addition by 1 */
```

The value of `k` and of `m` is the same: `-7`. In the case of `INT_MIN`, however, the peculiarity is that

```
INT_MIN == -INT_MIN
```

A modern C compiler does issue a warning when encountering the expression `-INT_MIN`, cautioning that the expression causes an *overflow* because of the addition operation. By the way, no other `int` value is equal to its negation under the 2's complement representation.

2.2.2. Integer Overflow

A programmer who uses any of the primitive C types needs to stay alert when it comes to `sizeof` and the potential for overflow. The next code example illustrates with the `int` type.

Listing 2-3. Integer overflow

```
#include <stdio.h>
#include <limits.h> /* INT_MAX */

int main() {
  printf("Max int in %lu bytes %i.\n", sizeof(int), INT_MAX);
  /* 4 bytes 2,147,483,647 */
  int n = 81;

  while (n > 0) {
    printf("%12i %12x\n", n, n);
    n *= n; /* n = n * n */
  }
  printf("%12i  %12x\n", n, n); /*  -501334399     e21e3e81 */
  return 0;
}
/*        81           51
        6561         19a1
    43046721      290d741
  -501334399     e21e3e81   ## e is 1101 in binary */
```

The *overflow* program (see Listing 2-3) initializes `int` variable `n` to 81 and then loops. In each loop iteration, `n` is multiplied by itself as long as the resulting value is greater than zero. The trace shows that loop iterates three times, and on the third iteration, the new value of `n` becomes negative. As the hex output shows, the leftmost (most significant) four bits are hex digit *e*, which is 1110 in binary: the leftmost 1 is now the sign bit for *negative*. In this example, the overflow could be delayed, but not prevented, by using a `long` instead of an `int`.

There is no compiler warning in the *overflow* program that overflow may result. It is up to the programmer to safeguard against this possibility. There are libraries that support arbitrary-precision arithmetic in C, including the GMP library (GNU Multiple Precision Arithmetic Library at https://gmplib.org). A later code example uses embedded assembly code to check for overflow.

2.3. Floating-Point Types

C has the floating-point types appropriate in a modern, general-purpose language. Computers as a rule implement the IEEE 754 specification (https://standards.ieee.org/ieee/754/6210/) in their floating-point hardware, so C implementations follow this specification as well.

Table 2-2 lists C's basic floating-point types. Floating-point types are signed only, and their values have a three-field representation under IEEE 754: sign bits, exponent bits, and significand (magnitude) bits. A floating-point constant such as `3.1` is of type `double` in C, whereas `3.1F` and `3.1f` are of type `float`. Recall that a `double` is 8 bytes in size, but a `float` is only 4 bytes in size.

Table 2-2. *Basic floating-point data types*

Type	Byte size	Range	Precision
`float`	4	1.2E-38 to 3.4E+38	6 places
`double`	8	2.3E-308 to 1.7E+308	15 places
`long double`	16	3.4E-4932 to 1.1E+4932	19 places

2.3.1. Floating-Point Challenges

Floating-point types pose challenges that make these types unsuitable for certain applications. For instance, there are decimal values such as 0.1 that have no exact binary representation, as this short code segment shows:

```
float n = 0.1f;
printf("%.24f\n", n); /* 0.100000001490116119384766 */
```

In the `printf` statement, the formatter `%.24f` specifies a precision of 24 decimal places. As a later example illustrates, unexpected rounding up can occur when a particular decimal value does not have an exact binary representation. Even this short code segment underscores that floating-point types should not be used in financial, engineering, and other applications that require exactness and precision. In such applications, there are libraries such as GMP (http://gmplib.org), mentioned earlier, to support arbitrary-precision arithmetic.

WHAT'S A MACRO?

A *macro* is a code fragment with a name and is created with a `#define` directive. The macro *expands* into its definition during the *preprocessing* stage of compilation. Here is a macro for *pi* from the *math.h* header file:

```
#define M_PI   3.14159265358979323846 /* the # need not be flush
against the define */
```

Although macros are often named in uppercase, this is convention only. Here are two *parameterized macros* for computing the *max* and *min* of two integer arguments:

```
#define min(x, y)  (y) ^ ((x ^ y) & -(x < y))  /* details of
                                                bitwise operators
                                                later */
```

```
#define max(x, y)  (x) ^ ((x ^ y) & -(x < y))  /* ^ bitwise xor,
                                                  & bitwise and */
```

These macros look like functions, but the compiler does no type-checking on the arguments. Here are two sample uses:

```
int n = min(-127, 44);  /* -127 */
n = max(373, 1404);     /* 1404 */
```

Another example underscores the problem of comparing floating-point values, in particular for equality. Imagine a company in which sales people earn a bonus if they sell 83% of their quota by the end of the third quarter. The company assumes that the remaining 17% of the quota, and probably more, will be met in the last quarter. In this company, 83% is defined in the official spreadsheet as the value 5.0 / 6.0. (On my handheld calculator, 5.0 / 6.0 evaluates to 0.833333333.) However, a legacy program computes 83% as (1.0 / 3.0) × 2.5. At issue, then, is whether (1.0 / 3.0) × 2.5 = 5.0 / 6.0. Here is a segment of C code that makes the comparison, using `double` values:

```
if (((1.0 / 3.0) * 2.5) == (5.0 / 6.0)) /* equal? */
  printf("Equal\n");
else
  printf("Not equal\n");  /** prints **/
```

A look at the hex values for the two expressions confirms that they are not equal:

```
3f ea aa aa aa aa aa aa /* (1.0 / 3.0) x 2.5 */
3f ea aa aa aa aa aa ab /* 5.0 / 6.0 */
```

The two differ in the least significant digit: hex `a` is `1010` in binary, whereas hex `b` is `1011` in binary. The two values differ ever so slightly, in the least significant (rightmost) bit of their binary representations. In close-to-the-metal C, the equality operator compares bits; at the bit level, the two expressions differ.

High-level languages provide a way to make approximate comparisons where appropriate. In particular, the header file *math.h* defines the macro `FLT_EPSILON`, which represents the difference between `1.0f` and the smallest, 32-bit floating-point value greater than `1.0f`. The value of `FLT_EPSILON` should be no greater than 1.0e-5f. On my desktop computer:

```
FLT_EPSILON == 1.192092895508e-07  /** e or E for scientific
notation **/
```

C has similar constants for other floating-point types (e.g., `DBL_EPSILON`).

Listing 2-4. Approximate equality

```
float f1 = 5.0f / 6.0f;
float f2 = (1.0f / 3.0f) * 2.5f;
if (fabs(f1 - f2) < FLT_EPSILON)  /* fabs for floating-point
                                  absolute value */
  printf("fabs(f1 - f2) < FLT_EPSILON\n"); /* prints */
```

The *comp* code segment (see Listing 2-4) shows how a comparison can be made using `FLT_EPSILON`. The library function `fabs` returns the absolute value of the difference between `f1` and `f2`. This value is less than `FLT_EPSILON`; hence, the two values might be considered equal because their difference is less than `FLT_EPSILON`.

The next two examples reinforce the risks that come with floating-point types. The goal is to show various familiar programming contexts in which floating-point issues arise. Following each example is a short discussion.

Listing 2-5. Issues with floating-point data types

```
/* 1.010000
   2.020000
   ...
   7.070001   ;; rounding up now evident
   ...
  10.100001
*/
float incr = 1.01f;
float num = incr;
int i = 0;
while (i++ < 10) {   /* i++ is the post-increment operator */
  printf("%12f\n", num); /* %12f is field width, not
                           precision */
  num += incr;
}
```

The *rounding* program (see Listing 2-5) initializes a variable to 1.01 and then increments this variable by that amount in a loop that iterates ten times. The rounding up becomes evident in the seventh loop iteration: the expected value is 7.070000, but the printed value is 7.07001. Note that the formatter is `%12f` rather than `%.12f`. In the latter case, the printouts would show 12 decimal places but here show the default places, which happens to be six. Instead, the 12 in `%12f` sets the field *width*, which right-justifies the output to make it more readable.

WHAT'S THE DIFFERENCE BETWEEN THE PRE-INCREMENT AND POST-INCREMENT OPERATORS?

The *rounding* program uses the post-increment operator on loop counter `i` to check, in the `while` condition, whether the loop counter is less than ten. C also has a pre-increment operator and both pre- and post-decrement operators. Each operator involves an *evaluation* and an *update*. Here is a code segment to illustrate the difference:

```
int i = 1;
printf("%i\n", i++); /* 1  (evaluate, then increment) */
printf("%i\n", i);   /* 2  (i has been incremented above) */
printf("%i\n", ++i); /* 3  (increment, then evaluate) */
```

Listing 2-6. More examples of decimal-to-binary conversion

```
#include <stdio.h>
#include <math.h> /* pi and e as macros, M_PI and M_E,
                  respectively */

void main() {
  printf("%0.50f\n", 10.12);
  /* 10.11999999999999921840299066388979554176330566406250 */

  /* On my handheld calculator: 2.2 * 1234.5678 = 2716.04916 */
  double d1 = 2.2, d2 = 1234.5678;
  double d3 = d1 * d2;
  if (2716.04916 == d3) printf("As expected.\n");
  /* does not print */
  else printf("Not as expected: %.16f\n", d3);
  /* 2716.0491600000004837 */
  printf("\n");
```

```
  /* Expected price: $84.83 */
  float price = 4.99f;
  int quantity = 17;
  float total = price * quantity; /* compiler converts quantity
                                  to a float value */
  printf("The total price is $%f.\n", total); /* The total
                                                price is
                                                $84.829994. */

  /* e and pi */
  double ans = pow(M_E, M_PI) - M_PI; /* e and pi, respectively */
  printf("%lf\n", ans); /* 19.999100 prints: expected is
  19.99909997 */
}
```

The *d2bconvert* program (see Listing 2-6) shows yet again how information may be lost in converting from decimal to binary. In these isolated examples, of course, no harm is done; but these cases underscore that floating-point types such as `float` and `double` are not suited for applications involving, for instance, currency.

2.3.2. IEEE 754 Floating-Point Types

This section digs into the details of the IEEE 754 binary floating-point specification (https://standards.ieee.org/standard/754-2019.html), using 32-bit floating-point values as the working example. The specification also covers 16-bit and 64-bit binary representations and decimal representations as well. Here is the layout of a 32-bit (*single-precision*) binary floating-point value under IEEE 754:

```
+-+--------+----------------------+
|s|exponent|       magnitude      | 32 bits
+-+--------+----------------------+
 1    8              23
```

For reference, the *written exponent* comprises the 8 bits depicted previously. In the discussion that follows, the *written exponent* is contrasted with the *actual exponent*. Also, the *written magnitude* comprises the 23 bits shown previously and is contrasted with the *actual magnitude*.

The IEEE 754 specification categorizes floating-point values as either *normalized* or *denormalized* or *special*. The category depends on the value of the 8-bit exponent:

- If the written exponent field contains a *mix* of 0s and 1s, the value is *normalized*.
- If the written exponent field contains *only* 0s, the value is *denormalized*.
- If the written exponent field contains *only* 1s, the value is *special*.

As the name suggests, *normalized values* are typical or expected ones such as -118.625, which is -1110110.101 in binary. A normalized value has an *implicit leading 1*, which means the written magnitude is the fractional part of the *actual magnitude*:

```
1.??????...???  ## the question marks ? are the written magnitude
```

For the sample value -1110110.101 (-118.625 in decimal), the *implicit leading 1* is obtained by moving the binary point six places to the left, which yields -1.110110101×2^6. The written magnitude is then the fractional part 110110101.

In the example, the *actual exponent* is 6, as shown in the expression -1.110110101×2^6. However, the written exponent of 133 (10000101 in binary) is *biased*, with a bias of 127 for the 32-bit case. The bias is subtracted from the written exponent to get the actual exponent:

```
actual exponent = written exponent - 127 ## 133 - 127 = 6
```

In summary, the decimal value -118.625 has a written exponent of 133 in IEEE 754, but an actual exponent of 6.

Finally, the sample value is negative, which means the most significant (leftmost) bit is a 1. The 32-bit representation for the decimal value -188.625 is

```
1 10000101 11011010100000000000000 ## 14 zeros pad to
make 23 bits
```

The middle field alone, the 8-bit exponent, indicates that this value is indeed *normalized*: the written exponent contains a mix of 0s and 1s.

Denormalized values cover *two* representations of zero and evenly spaced values in the vicinity of zero. Zero can represented as either a negative or a nonnegative value under the IEEE specification, which the C compiler honors:

```
if (-0.0F == 0.0F) puts("yes!");  /* prints */
```

The IEEE representation of zero is intuitive in that every bit—except, perhaps, the sign bit—is a 0. A denormalized value does not have an implicit leading 1, and the actual exponent has a fixed value of -126 in the 32-bit case. The written exponent is always all 0s.

What motivates the *denormalized* category beyond the two representations of zero? Consider the three values in Table 2-3, in particular the binary column. In the first row, the value has a single 1—the least significant bit of the written exponent. Yet this exponent still contains a *mix* of 0s and 1s and so is *normalized*: it is the *smallest* positive normalized value in 32 bits.

Table 2-3. *Positive denormalized and normalized values*

Binary	Decimal
0 00000001 00000000000000000000000	1.175494350822e-38
0 00000000 11111111111111111111111	1.175494210692e-38
0 00000000 00000000000000000000001	1.401298464325e-45

The value in the middle row has all 0s in the exponent, which makes the value *denormalized*. This value is the *largest* denormalized value in 32 bits, but this value is still *smaller* than the very small normalized value above it. The smallest denormalized value, the bottom row in the table, has a single 1 as the least significant bit: all the rest are 0s. Between the smallest and the largest denormalized values are many more, all differing in the bit pattern of the written magnitude. Although the denormalized values shown so far are positive, there are negative ones as well: the sign bit is 1 for such values.

In summary, *denormalized* values cover the two representations of zero, as well as evenly spaced values that are close to zero. The preceding examples show that the gap between the smallest positive normalized value and positive zero is considerable and filled with denormalized values.

The third IEEE category covers *special* values, three in particular: NaN (Not a Number), positive infinity, and negative infinity. A written exponent of *all* 1s signals a special value. If the written magnitude contains all 0s, then the value is either negative or positive infinity, with the sign bit determining the difference. If the written magnitude contains at least one 1, the value is NaN. A short code segment clarifies.

Listing 2-7. Special values under the IEEE 754 specification

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

/** gcc -o specVal specVal.c -lm **/
void main() {
  printf("Sqrt of -1:    %f\n", sqrt(-1.0F));
                        /* 1 11111111 10000000000000000000000 */
  printf("Neg. infinity: %f\n", 1.0F / -0.0F);
                        /* 1 11111111 00000000000000000000000 */
  printf("Pos. infinity: %f\n", 1.0F / 0.0F);
                        /* 0 11111111 00000000000000000000000 */
}
```

The *specVal* program (see Listing 2-7) has the following output, with comments introduced by ##:

```
Sqrt of -1:     -nan   ## minus sign because -1.0F is negative
Neg. infinity:  -inf   ## negative zero as divisor
Pos. infinity:   inf   ## non-negative zero as divisor
```

The floating-point units (FPUs) of modern computers commonly follow the IEEE specification; modern languages, including C, do so in any case. There are heated discussions within the computing community on the merits of the IEEE specification, but there is little doubt that this specification is now a de facto standard across programming languages and systems.

HOW DOES LINKING WORK IN THE COMPILATION PROCESS?

Compiling the *specVal* program into an executable requires an explicit link flag:

```
% gcc -o specVal specVal.c -lm
```

In the flag `-lm` (lowercase L followed by m), the `-l` stands for *link*, and the `m` identifies the standard mathematics library *libm*, which resides in a file such as *libm.so* on the compiler/linker search path (e.g., in a directory such as */usr/lib* or */usr/local/lib*). Note that the prefix `lib` and the file extension `so` fall away in a link specification, leaving only the `m` for the mathematics library. The linking is needed because the *specVal* program calls the `sqrt` function from the mathematics library. A compilation command may contain several explicit link flags in same style shown previously: `-l` followed by the name of the library without the prefix `lib` and without the library extension such as `so`.

During compilation, libraries such as the standard C library and the input/output library are linked in automatically. Other libraries, such as the mathematics and cryptography libraries, must be linked in explicitly. In Chapter 8, the section on building libraries goes into more detail on linking.

2.4. Arithmetic, Bitwise, and Boolean Operators

C has the usual arithmetic, bitwise, and boolean (*relational*) operators. Recall that even the *character* types `char` and `wchar_t`, and the makeshift-boolean type (zero for *false*, nonzero for *true*), are fundamentally arithmetic types. However, some operators are ill-suited for some types. For example, floating-point values should *not* be bit-shifted, left or right.

Recall the layout for a 32-bit floating-point value under IEEE 754:

```
+-+--------+----------------------+
|s|exponent|       magnitude      |
+-+--------+----------------------+
 1    8               23
```

Bit-shifting a floating-point type, either left or right, would cause one or more bits to change fields. On a 2-bit left shift, for instance, magnitude bits would become exponent bits, and an exponent bit would become the sign bit. The following code segment illustrates the peril of shifting floating-point values:

```
float f = 123.456f;
f = (int) f << 2;   /* ERROR without the cast operation (int) */
printf("%f\n", f);  /* 492.000000 */
```

The second line uses a *cast* operation, which is an explicit type-conversion operation; in this case, the floating-point value of variable `f` is converted to an `int` value so that the compiler does not complain. (The syntax of casts is covered in the following sidebar.) In the shift operation, `<<` represents a *left* shift, and `>>` represents a *right* shift. To the left of the shift operator is the value (in this case, variable `f`) to be shifted, and to the right is the number of bit places to shift. On left shifts, the vacated positions are filled with 0s.

If the preceding example were to omit the cast operation, the compiler would complain, with an error rather than just a warning, that the left operand to `<<` should be an `int`, not a `float`. To get by the compiler, the code segment thus includes the cast operation.

It should be emphasized that a cast operation is not an assignment operation. In this example, the casted value `123.456` is still stored in variable `f`. The salient point is that floating-point values, in general, should not be shifted at all. The shift operation is intended for integer values only, and even then caution is in order—as later examples illustrate.

HOW DO CAST OPERATIONS WORK?

A *cast operation* consists of a data type enclosed in parentheses immediately to the left of a value:

```
int n = (int) 1234.5678f;   /* cast float value to int value,
                               which is assigned to n */
float f = (float) n;        /* compiler would do the conversion
                               in any case */
n = (int) 1234.5678F << 2;  /* cast required: float values
                               should not be shifted */
```

A cast is *not* an assignment: in the second example shown previously, the cast `(float)` does not change what is stored in `n` but rather creates a new value then assigned to variable `f`. A cast is thus an *explicit* conversion of one type to another. The compiler regularly does such conversions automatically:

```
int n = 1234.567f; /* compiler assigns 1234 to n: automatic
                      conversion */
```

For convenience, the following subsections divide the operators into the traditional categories of arithmetic, bitwise, and boolean (relational). Miscellaneous operators such as `sizeof` and the cast will continue to be clarified as needed.

2.4.1. Arithmetic Operators

C has the usual unary and binary arithmetic operators, and C uses the standard symbols to represent these operators. For operations such as exponentiation and square roots, C relies upon library routines, in this case the `pow` and `sqrt` functions, respectively. Table 2-4 clarifies the binary arithmetic operators with sample expressions.

Table 2-4. *Binary arithmetic operators*

Operation	C	Example
Addition	+	12 + 3
Subtraction	-	12 - 3
Multiplication	*	12 * 3
Division	/	12 / 3
Modulus	%	12 % 3

The plus and minus signs also designate the *unary plus* and *unary minus* operators, respectively:

```
int k =  5;
printf("%i  %i\n", +k, -k); /* 5  -5 */
```

The binary arithmetic operators associate left to right, with multiplication, division, and modulus having a higher precedence than addition and subtraction. For example, the expression

```
8 + 2 * 3
```

evaluates to 14 rather than 30. Of course, parentheses can be used to ensure the desired association and precedence—and to make the arithmetic expressions easier to read.

Listing 2-8. Operator association and precedence

```
#include <stdio.h>

void main() {
  int n1 = 4, n2 = 11, n3 = 7;
  printf("%i\n", n1 + n2 * n3);     /*  81 */
  printf("%i\n", (n1 + n2) * n3);   /* 105 */
```

```
  printf("%i\n", n3 * n2 % n1);     /*   1 */
  printf("%i\n", n3 * (n2 % n1));   /*  21 */
}
```

The *assoc* program (see Listing 2-8) shows how expressions can be parenthesized in order to get the desired association when mixed operations are in play. The use of parentheses seems easier than trying to recall precedence details, and parenthesized expressions are, in any case, easier to read.

C has variants of the assignment operator (=) that mix in arithmetic and bitwise operators. A few examples should clarify the syntax:

```
int n = 3;
n += 4;    /* n = n + 4 */
n /= 2;    /* n = n / 2 */
n <<= 1;   /* n = n << 1 */
```

2.4.2. Boolean Operators

The *boolean* or *relational* operators are so named because the expressions in which they occur evaluate to the boolean values *true* or *false*. Although any integer value other than zero is *true* in C, *true* boolean expressions in C evaluate to the default value for *true*, 1. Here are some sample expressions to illustrate the boolean operators:

```
/** equals and not-equals **/
2 == (16 - 14) /* true:  == is 'equals' */
2 != (16 / 8)  /* false: != is 'not equals' */

/** greater, lesser **/
!(2 < 3)       /* false: !  is 'negation' */
3 > 2          /* true:  >  is 'greater than' */
3 >= 3         /* true:  >= is 'greater than or equal to' */
```

```
3 < 2          /* false: <  is 'less than' */
3 <= 3         /* true:  <= is 'less than or equal to */

/** logical-and, logical-or **/
(2 < 3) && (4 < 5) /* true: && is logical-and */
(2 < 3) || (5 < 4) /* true: || is logical-or */
```

A few cautionary notes are in order. Note that the operators for equality (==) and inequality (!=) both have two symbols in them. The equality operator can be tricky because it is so close to the *assignment* operator (=). Consider this code segment, the stuff of legend among C programmers whose code has gone awry because of some variation of the problem:

```
int n = 2;
if (n = 1)
  printf("yep\n"); /** prints: presumably meant n == 1 **/
```

An assignment in C is an *expression* and so has a value—the value of the expression on the right-hand side of the = operator. Accordingly, the `if` test both assigns 1 to `n` and evaluates to 1, *true*; hence, the `printf` statement executes. Whenever a constant is to be compared against a variable, it is best to put the constant on the *left*. If the assignment operator = is then typed by mistake instead of the equality operator ==, the compiler catches the problem:

```
if (1 = n) /** won't compile **/
```

The logical *and* and logical *or* operators are efficient because they *short-circuit*. For example, in the expression

```
(3 < 2) && (4 > 2) /* only (3 < 2), the 1st conjunct, is
                      evaluated */
```

the second conjunct (`4 > 2`) is *not* evaluated: a conjunction is *true* only if each of its conjuncts is *true*, and the first conjunct (`3 < 2`) is *false*, thereby making the entire expression *false*.

The boolean operators occur regularly in loop and other tests. Simple examples have been seen already:

```
int i = 0;
while (i < 10) {  /* loop while i is less than 10 */
   /* ... */
   i += 1;      /* increment loop counter: i++ or ++i would
                work, too */
}
```

Richer examples are yet to come.

2.4.3. Bitwise Operators

As the name suggests, the *bitwise* operators work on the underlying bit-string representation of data. These operators thus deserve caution, as it may be hard to visualize the outcome of bit manipulation. Bitwise operations are fast, usually requiring but a single clock tick to execute. For example, an optimizing compiler might transform a source-code expression such as

```
n = n * 2;  /* n is an unsigned int variable: double n
            arithmetically */
```

to a left shift, shown here at the source level:

```
n = n << 1; /* double n by left-shifting one place */
```

Here are some more examples of the bitwise operators in expressions, using 4-bit values for readability:

```
~(0101) == 1010        /* invert bits: complement */

(0101 & 1110) == 0100  /* bitwise-and */
(0101 | 1110) == 1110  /* bitwise-inclusive-or */
(0101 ^ 1110) == 1011  /* bitwise-exclusive-or */

(0111 << 2) == 1100    /* left shift */
(0111 >> 2) == 0001    /* right shift */
```

The complement or *bit inversion* operator is tied to the unary minus operator considered earlier. Given an underlying 2's complement representation of signed integers, recall that the unary minus operator can be viewed as a combination of two operations: complement and increment by 1. Another example illustrates:

```
int n = 5;
if (-n == (~n + 1))
  printf("yep\n"); /* prints */
```

The shift operators require caution because overshifting in either direction is a misstep. As noted earlier, the compiler intervenes in case floating-point values are shifted left or right. At issue now are shifts of integer values. With *signed* integer values, left shifts can be risky because they may change the sign. Consider this example:

```
int n = 0x70000000;            /* 7 in binary is 0111 */
printf("%i  %i\n", n, n << 1); /* 1879048192  -536870912 */
```

The bit-level representation of `n` starts out `01110...`, with the leftmost bit as the sign bit `0` for *nonnegative*. The 1-bit left shift moves a `1` into the sign position, which accounts for change in sign from 1,879,048,192 to -536,870,912. Recall that, in left shifts, the vacated bit positions are filled with 0s.

Right shifts can be even trickier. Consider the *signed* integer value 0xffffffff in hex, which is all 1s in binary; in decimal, this is -1. Even in a 1-bit right shift, the sign could change to 0—if the shift is *logical*, that is, if the vacated bit is filled with a 0. If the shift is *sign preserving*, it is an *arithmetic shift*: the sign bit becomes the filler for the vacated positions. Whether a right is *logical* or *arithmetic* is platform dependent. In general, it is best to shift only *unsigned* integer values. Even in this case, of course, overshifting is possible; but at least the issue of sign preservation does not arise.

Listing 2-9. Reversing the endian-ness of a multibyte data item

```
unsigned int endian_reverse32(unsigned int n) { /* designed for
32 bits, or 4 bytes */
  return (n >> 24)          |  /* leftmost byte becomes
                               rightmost */
    ((n << 8) & 0x00FF0000) |  /* swap the two inner bytes */
    ((n >> 8) & 0x0000FF00) |  /* ditto */
    (n << 24);                 /* rightmost byte becomes
                               leftmost */
}
```

The *endian* code segment (see Listing 2-9) uses bitwise operators in a utility function that reverses the *endian-ness* of a 4-byte integer. Modern machines are still *byte addressable* in that an address is that of a single byte. For multibyte entities such as a 4-byte integer, an address thus points to a byte at one end or the other in the sequence of 4 bytes. Given this 4-byte integer

```
+----+----+----+----+
| B1 | B2 | B3 | B4 |  ## B1 is high-order byte, B4 is low-order byte
+----+----+----+----+
```

the integer's address would be either that of B1 (high-order byte) or that of B4 (low-order byte). Standard network protocols are *big endian*, with the integer's address that of the *big* (high-order) byte B1; Intel machines are *little endian*, with the integer's address that of the *little* (low-order) byte B4. (ARM machines are little endian by default but can be configured, as needed, to be big endian.) Given the preceding depiction, the *endian* program would reverse the byte order to yield:

```
+----+----+----+----+
| B4 | B3 | B2 | B1 |  ## B4 is high-order byte, B1 is low-order byte
+----+----+----+----+
```

A short code example illustrates, with integer n initialized to a hex value for clarity:

```
unsigned n = 0x1234abcd;
printf("%x  %x\n", n, endian_reverse(n)); /*
1234abcd  cdab3412 */
```

Recall that each hex digit is 4 bits. Accordingly, the leftmost byte in variable n is 12, and the rightmost is cd.

C has a header file *endian.h* that declares various functions for transforming little-endian formats to big-endian formats, and vice versa. These functions specify the bit sizes on which they work: 16 (2 bytes), 32 (4 bytes), and 64 (8 bytes).

WHAT IS AN *LVALUE* AND AN *RVALUE*?

An *rvalue* is one that does *not* persist. For example, in the statement

```
printf("%i\n", 444);  /* 444 does not persist, and is thus an
                      rvalue */
```

the rvalue `444` does not persist beyond the `printf` statement. By contrast, an *lvalue* does persist as the target of an assignment:

```
int n = 444;          /* 444 persists in n beyond the
                      assignment */
```

The variable `n` is the symbolic name of a memory location or CPU register, and a value assigned to `n` is thus an *lvalue*.

2.5. What's Next?

The examples so far have focused mostly on *scalar* variables: there is an identifier for a *single* variable, not a collection of variables. A typical example is

```
int n = -1234; /* n identifies a single variable */
```

C also supports *aggregates*, a *collection* of variables under a single name. Here is one example:

```
char* str = "abcd";     /* string literal abcd is a null-
                        terminated array of chars */
printf("%c\n", str[0]); /* string[0] = 1st of 5 variables, %c
                        for character */
```

The pointer variable `str` identifies a collection (in this case, an array) of *five* characters: the ones shown and the null terminator. The expression `str[0]` refers to the *first* of the variables that hold a character, lowercase *a* in this example. Pointer `str` thus identifies an aggregate rather than just a single variable.

Arrays and structures are the primary aggregates in C. Pointers also deserve a closer look because they dominate in efficient, production-grade programming. The next chapter focuses on aggregates and pointers.

WHAT'S THE RELATIONSHIP BETWEEN C AND C++?

C is a small, strictly *procedural* or *imperative* language. C++ is a large language that can be used in procedural style but also includes object-oriented features (e.g., classes, inheritance, and polymorphism) not found in C. C++, unlike C, has generic collection types. A C++ program can include orthodox C code, but much depends on the compiler; further, header files and the corresponding libraries may differ in name and location between the two languages. The two languages share history and features but are distinct.

CHAPTER 3

Aggregates and Pointers

3.1. Overview

This chapter focuses on arrays and structures, which are C's primary aggregate types. Arrays aggregate variables of the same type, whereas structures can do the same for variables of different types. Structures can be array elements, and a structure may embed arrays. Together these aggregate types make it possible for programmers to define arbitrarily rich data types (e.g., `Employee`, `Game`, `Species`) that meet application needs.

Pointers—address constants and variables—come into play naturally with both arrays and structures, and the code examples throughout the chapter get into the details. Among modern general-purpose languages, C (together with C++) stands out by giving the programmer so much control over—and, therefore, responsibility for—memory addresses and the items stored at these addresses. All of the chapters after this one have examples that, in one way or another, illustrate the power of pointers.

M. Kalin, *Modern C Up and Running*, https://doi.org/10.1007/978-1-4842-8676-0_3

3.2. Arrays

An *array* in C is a fixed-size collection of variables—of the *same* type—accessible under a single name, the array's identifier. A code example illustrates.

Listing 3-1. A simple array

```
#define Size 8

void main() {
  int arr[Size];              /* storage from the stack --
                                 uninitialized */
  int i;
  for (i = 0; i < Size; i++)  /* iterate over the array */
    arr[i] = i + 1;           /* assign a value to each
                                 element */
}
```

The *array* program (see Listing 3-1) shows the basic syntax for declaring an array and then uses a `for` loop to populate the array with values. An array has a *fixed* size, in this case specified by the macro `Size`. The array's name, in this case `arr`, is a *pointer constant* that holds the address of the array's first element, in this case the element that the `for` loop initializes to 1:

```
       +---+---+---+---+---+---+---+---+
arr--->| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |   ## array elements
       +---+---+---+---+---+---+---+---+
        [0] [1] [2] [3] [4] [5] [6] [7]    ## indexes
```

Arrays can be *indexed* to access elements by using the square brackets: legitimate indexes are 0 through the array's size - 1. The indexes are offsets from the start of the array: the first array element is at offset 0, the second at offset 1, and so on. For the preceding array, here are some of the addresses computed as offsets from the base address `arr`:

```
arr + 0 ---> 1st element  ## arr[0] is value of 1st element: 1
arr + 1 ---> 2nd element  ## arr[1] is value of 2nd element: 2
...
arr + 7 ---> 8th element  ## arr[7] is value of 8th element: 8
```

A second example builds on the first by introducing *pointer variables* and showing how C supports *pointer arithmetic* by having data types for pointers.

DOES C PROVIDE *BOUNDS CHECKING* ON ARRAYS?

No. The programmer is responsible for ensuring that array indexes are *in bounds* at runtime. The following code segment compiles without warning and likely blows up when executed because of the out-of-bounds index -9876.

```
int arr[4];      /* four elements */
int ind = -9876; /* not a good index: 0, 1, 2, and 3 are good
                 indexes */
arr[ind] = 27;   /* out-of-bounds, likely to blow up at
                 run-time */
```

3.3. Arrays and Pointer Arithmetic

C supports *typed pointers* so that the compiler can perform the required arithmetic when pointers are used to access memory locations. The compiler thereby takes on a task that would be error-prone if left to the programmer. Consider again the *array* program with its array of eight `int` elements and a sample index such as 2. The index expression `arr[2]` references the *third* element in the array, which is two elements over from where the array starts: `arr` is the base address, and 2 is the displacement or offset from this base address. However, machine-level addresses are of *bytes*, and an `int` is a 4-byte element. To reach the array's third element,

it is therefore necessary to move 2 × `sizeof(int)` bytes from where array `arr` starts, which is a move of 8 bytes in all. Yet the programmer refers to the third element as `arr[2]` (`int` level), not as `arr[8]` (byte level). It would be tedious and error-prone for programmers to work at the byte level in accessing array elements of multibyte types. Accordingly, C's typed pointers allow the programmer to work at the data-type level (e.g., `int` or `Employee`), while the compiler then works at the byte level.

Listing 3-2. Pointer variables and pointer arithmetic

```
#define Size (8)          /* Size is a macro that expands
                             into (8) */
void main() {
  int arr[Size];          /* storage from the stack --
                             uninitialized */
  int k = 1;
  int* ptr = arr;         /* point to the start of the array */
  int* end = arr + Size; /* points immediately beyond the end
                             of the array */

  while (ptr < end) {     /* beyond the end yet? */
    *ptr = k++;           /* assign a value to the array
                             element */
    ptr++;                /* increment the pointer */
  }
}
```

The *arrayPtr* program (see Listing 3-2), which revises the original *array* program, has three pointers at work:

- The array's name `arr`, a pointer *constant*, holds the address of the first element in the array.
- The pointer *variable* `ptr`, assigned to hold the address of the first element in the array.
- The pointer variable `end` points just beyond the last element in the array.

The following is a depiction of where `ptr` and `end` point before the looping begins:

```
       +---+---+---+---+---+---+---+---+---+---+
ptr--->| ? | ? | ? | ? | ? | ? | ? | ? | ? |<---end
       +---+---+---+---+---+---+---+---+---+---+
        [0] [1] [2] [3] [4] [5] [6] [7]     ## indexes
```

A pointer is allowed to point one element beyond the end of the array, although nothing should be stored at that location. In this example, the array's initialization now uses a `while` rather than a `for` loop, and the loop's condition compares the two pointers, `ptr` and `end`: looping continues so long as `ptr < end`. At the bottom of the loop, `ptr` is incremented by 1—by *one* `int`, which is 4 bytes. The pointer `ptr` is a variable, unlike the pointer constant `arr`, and so can have its value changed. Eventually `ptr` points to the same location as does `end`, which makes the loop condition *false*.

The initialization of each array element uses the *dereference operator*, the star:

```
*ptr = k++; /* k is 1,2,3,...,8 */
```

In the declaration of ptr, the star comes *after* the data type int. In the dereferencing of ptr, the star comes *before* the variable's name. It would be an error to change the code to

```
ptr = k; /** ERROR **/
```

because ptr then would take on values such as 1,2,3,...,8, which almost surely are not addresses within the program's address space. The aim is to initialize the array element to which ptr points, not ptr itself.

3.4. More on the Address and Dereference Operators

In the *addPtr* example, the pointer variable ptr is initialized to the array's name arr so that both arr and ptr point to the array's first element. An equivalent but less concise initialization uses the *address operator* &:

```
int* ptr = &arr[0]; /* alternative to: int* ptr = arr; */
```

The address operator computes an in-memory address, in this case the address of array element arr[0].

The *dereference operator* uses an address to access the contents stored at that address. If ptr points to any cell in the int array arr, then *ptr is the value stored at the address. The dereference operator can be used in the usual ways, for example, to read or to change a value:

```
int* ptr = &arr[3]; /* address of 4th element, which contains 4 */
*ptr = *ptr + 9;   /* equivalent to: arr[3] = arr[3] + 9 */
```

The examples so far have shown pointers that hold the addresses of char and int cells, but not pointers to other pointers. In principle, there can be *pointer to pointer to...*, although in practice, it is unusual to see more than two levels of indirection. The next example illustrates the case of a pointer to a pointer, and later examples motivate such a construct.

Listing 3-3. The address and dereference operators

```
#include <stdio.h>

void main() {
  int n = 1234;
  int* ptr1 = &n;         /* ptr1--->n */

  int** ptr2 = &ptr1;     /* ptr2--->ptr1 */
  printf("%i %p %p\n", n, ptr1, ptr2);   /* 1234 0x7ffee80dfb5c
                                            0x7ffee80dfb60 */

  **ptr2 = *ptr1 + 100;  /* increment n by 100 */
  printf("%i %i %i\n", n, *ptr1, **ptr2); /* 1334 1334 1334 */
}
```

The *ptr2ptr* program (see Listing 3-3) has an `int` variable `n` that stores 1234, a pointer `ptr1` that points to `n`, and a second pointer `ptr2` that points to `ptr1`. Here is a depiction, with fictional addresses written in hex above the storage cells and variable names below these cells:

```
              0xAB         0xEF       ## addresses
+------+    +------+    +------+
| 0xAB |--->| 0xEF |--->| 1234 |     ## contents
+------+    +------+    +------+
   ptr2        ptr1          n        ## variable names
```

Given this storage layout, any of the variables `n`, `ptr1`, and `ptr2` can be used to access (including to update) the value stored in variable `n`. For example, each of these statements updates `n` by one:

```
n += 1;       /* from 1234 to 1235 */
*ptr1 += 1;   /* from 1235 to 1236 */
**ptr2 += 1;  /* from 1236 to 1237 */
```

The index syntax used with arrays can be seen as *syntactic sugar*, as a short example shows:

```
int arr[] = {9, 8, 7, 6, 5};  /* compiler figures out the size */
int n = arr[2];               /* n = arr[2] = 7 */
```

The syntax `arr[2]` is straightforward and now is common across programming languages. In C, however, this syntax can be viewed as shorthand for

```
int n = *(arr + 2);  /* n = 7 */
```

The pointer expression `arr + 2` points to two `int` elements beyond the first in the array, which holds 7. Dereferencing the pointer expression `*(arr + 2)` yields the `int` contents, in this case 7.

The same point can be reinforced with some obfuscated C. Consider this code segment:

```
int arr[] = {9, 8, 7, 6, 5};
int i;
for (i = 0; i < 5; i++)
   printf("%i ", i[arr]); /** peculiar syntax **/
```

In the `printf` statement, the usual syntax for array access would be `arr[i]`, not `i[arr]`. Yet either works, and the compiler does not wince at the second form. The reason can summarized as follows:

```
arr[i] == *(arr + i)       /* syntactic sugar */
*(arr + i) == *(i + arr)   /* addition commutes */
*(i + arr) == i[arr]       /* more syntactic (but peculiar)
                              sugar */
```

3.5. Multidimensional Arrays

An array declared with a *single* pair of square brackets is *one-dimensional* and sometimes called a *vector*. An array declared with more than one pair of square brackets is *multidimensional*:

```
int nums[128];          /* one dimensional array */
int nums_table[4][32];  /* multidimensional array
                           (2-dimensional matrix) */
```

Arrays of any dimension are possible, but more than three dimensions is unusual. The array `nums_table` is two-dimensional. The arrays `nums` and `nums_table` hold the same number of integer values (128), but they do *not* have the same number of elements: array `nums` has 128 elements, each an `int` value; by contrast, array `nums_table` has four elements, each a *subarray* of 32 `int` values. The `sizeof` operator, when applied to an array's name, does the sensible thing: it gives the number of bytes required for all of the array elements, not the size in bytes of the array's name as pointer. In this case, for example, the `sizeof` array `nums` is the same as the `sizeof` array `nums_table`: 512 because there are 128 `int` values in each array and each `int` is 4 bytes.

Multidimensional arrays are yet another example of syntactic sugar in C. All arrays are implemented as one-dimensional, as the next code example illustrates.

Listing 3-4. Treating a multidimensional array as a one-dimensional array

```
#include <stdio.h>

void main() {
  int table[3][4] = {{1, 2, 3, 4}, /* row 1 */
                     {9, 8, 7, 6}, /* row 2 */
                     {3, 5, 7, 9}}; /* row 3 */
```

```
  int i, j;
  for (i = 0; i < 3; i++)    /** outer loop: 3 rows **/
    for (j = 0; j < 4; j++)  /** inner loop: 4 cols per row **/
      printf("%i ", table[i][j]);
  printf("\n");

  int* ptr = (int*) table;  /** ptr points to an int **/
  for (i = 0; i < 12; i++)  /** 12 ints (3 rows, 4 cols each) **/
    printf("%i ", ptr[i]);
  printf("\n");
}
```

The *table* program (see Listing 3-4) highlights critical features about how pointers work in C. The array name `table` is, as usual, a pointer constant, and this name points to the first byte of the first `int` in the *first element* in the array, where the first array element is a *subarray* of four `int` values, in this case 1, 2, 3, and 4:

```
              1st row         2nd row         3rd row       ## rows
          +---+---+---+---|---+---+---+---|---+---+---+---+
table--->| 1 | 2 | 3 | 4 | 9 | 8 | 7 | 6 | 3 | 5 | 7 | 9 |  ## contents
          +---+---+---+---|---+---+---+---|---+---+---+---+
           [0] [1] [2] [3] [0] [1] [2] [3] [0] [1] [2] [3]   ## column indexes
```

The data type of `table` is *pointer to an array of subarrays, each with four integer elements*. In memory, the array is laid out contiguously, with the `int` values in sequence, one table row (subarray) after the other.

The *table* program traverses the multidimensional array twice. The first traversal uses nested `for` loops: the outer `for` loop iterates over the *rows*, and the inner `for` loop iterates over the *columns* in each row. The C compiler lays out the `table` in *row-major* order: the first row with all of its

columns, then the second row with all of its columns, and so on. A Fortran compiler, by contrast, would lay out a multidimensional array in *column-major* order.

The second traversal of array `table` uses only a single `for` loop. The variable `ptr` is assigned the value of `table`, but with a cast: the cast `(int*)` is required because `ptr` is of type `int*`, whereas `table` is not. A revision to the *table* example goes into the details.

Listing 3-5. A function to print the two-dimensional table of three rows and three columns

```
void print(int (*arr)[4], int n) {
  int i, j;
  for (i = 0; i < n; i++)
    for (j = 0; j < 4; j++)
      printf("%i ", arr[i][j]);
  printf("\n");
}
```

To get a better sense of the `table` data type, imagine breaking out a `print` function for printing the two-dimensional `table` (see Listing 3-5). The first parameter in the `print` function could be written in different ways, including the one shown. Another way is this:

```
void print(int arr[ ][4], int n)
```

Both versions underscore that the first argument passed to `print`, in this case the two-dimensional array `table`, must be an *array of subarrays, with each subarray of size 4*. The second argument `n` to the `print` function specifies the number of *rows* in the array. From the `main` function in the *table* program, the call would be

```
print(table, 3); /* 3 rows */
```

The parameter `arr` in function `print` then points to the first row, and second parameter `n` gives the number of rows. The cast of `table` to `int*` in the assignment

```
int* ptr = (int*) table;
```

acknowledges to the compiler that pointer constant `table` and pointer variable `ptr` may point to the very same byte, but that the two differ in type. As an `int*` pointer, `ptr` can be used to iterate over the individual `int` values in the array, rather than over the four-element subarrays that make up each `table` row.

Consider the *pointer* expressions `table[0]`, `table[1]`, and `table[2]`. Each of these points to an array of three integers. Here is the output from a sample run that prints out the three addresses:

```
printf("%p (%lu) %p (%lu)  %p (%lu)\n",
    table[0], (long) table[0],  /* 0x7ffececccf30
                                   (140730827343600) */
    table[1], (long) table[1] , /* 0x7ffececccf40
                                   (140730827343616) */
    table[2], (long) table[2]); /* 0x7ffececccf50
                                   (140730827343632) */
}
```

The first and second addresses differ by 16 bytes, as do the third and fourth. The variable `table[0]` points to the first of the three rows in the table, and each row has four `int` values of 4 bytes apiece; hence, `table[1]` points 16 bytes beyond where `table[0]` points.

The syntax of multidimensional arrays gives a hint about how various pointer expressions are to be used. The `table` array, which holds `int` values, is declared with *two* sets of square brackets:

```
int table[3][4] = {...};
```

If index syntax is used to read or write an `int` value, then *two* square brackets must be used:

```
table[1][2] = -999; /* second row, third column set to -999 */
```

The first index picks out the *row*, and the second index picks out the *column* in the row. Any expressions involving `table`, but with *fewer than* two pairs of brackets, are *pointers* rather than `int` values. In particular, `table` points to the first subarray, as does `table[0]`; pointer `table[1]` points to the second subarray; and pointer `table[2]` points to the third subarray. A quick review exercise is to explain, in plain terms or through a code segment, the difference between the data type of `table` and the data type of `table[0]`. Both pointer expressions point to the *same byte*, but the two differ in type.

C arrays promote *efficient* modular programming. Consider again a function to print one-dimensional integer arrays of arbitrary sizes. As the *table* program shows, it is straightforward to treat an *n*-dimensional array as if it were one-dimensional. The *print_array* function might be declared as follows:

```
void print_array(int* arr, unsigned n); /* void print_array(int
                                           arr[], unsigned n); */
```

The obvious way to call `print_array` is to pass it, as the first argument, the array's name—a *pointer*:

```
int arr[100000];
/* fill the array */
print_array(arr, 100000); /* passing a pointer as the
                             1st arg */
```

To pass the array's name as an argument is thus to pass a *pointer* to the array, not a *copy* of it. Passing a copy of 100,000 4-byte integers would be expensive, maybe prohibitively so. It is possible to pass a copy of an array to a function, another issue for later analysis.

HOW ARE ARGUMENTS PASSED TO C FUNCTIONS?

C uses *call by value* exclusively in passing arguments to functions: the arguments are *copied* and then accessible in the called function through the parameter names. The compiler can optimize such calls in various ways, including placing arguments in CPU registers rather than on the stack. Addresses (pointers) as well are passed by value. For example, when an array's name is passed as an argument, a copy of this address is passed. Of course, both the copy and the original address can be used to access the very same array elements.

3.6. Using Pointers for Return Values

A function in C can take arbitrarily many arguments, but it can return *one* value at most. The restriction to just one returned value is not troubling, however. To begin, the single returned value could be a *list* of values, although this approach requires caution. Later code examples explore the option and go into best practices for returning collections. This section takes on a different approach: using a pointer *argument* to store a value that otherwise might be returned explicitly by a function:

```
int f() { return 100; }          /* explicitly returned */
void g(int* arg) { *arg = 100; } /* stored at a provided
                                    address */
```

The technique is common in C. A function's caller provides the *address* of some variable, and the callee then stores a value at this address. The effect is to return a value via the pointer. The next code example motivates this approach and also introduces in-line assembly code to check for integer overflow.

Listing 3-6. In-line assembly code to check for integer overflow

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

int safe_mult(int n1, int n2, int* product) {
  int flag = 0;       /* assume no overflow */
  *product = n1 * n2; /* potential overflow */

  asm("setae %%bl; movzbl %%bl,%0"
      : "=r" (flag)  /* set flag on overflow */
      :              /* no other inputs */
      : "%rbx");     /* scratchpad */
  return flag; /* zero is no overflow, non-zero is overflow */
}
```

The *safeMult* function (see Listing 3-6) introduces in-line assembly with a call to the library function `asm`. The architecture-specific assembly code is in AT&T style and targets an Intel machine; the code detects overflow in integer multiplication, returning a flag to indicate whether overflow occurred.

The syntax of the in-line assembly code needs a quick analysis. The percentage sign `%` used to identify a CPU register sometimes occurs twice, in this case to identify the 1-byte, special-purpose register `%%bl`. The double percentage signs are there to prevent the assembler from confusing this register identifier with something else. One percentage sign might do, but two are safer.

The argument to the `asm` function can be divided into two parts:

- The string

  ```
  "setae %%bl; movzbl %%bl,%0"
  ```

contains two instructions, with a semicolon separating them. The `setae` instruction puts the result of the overflow test in the 1-byte register `%bl`. This register now flags whether overflow occurs. The `movzbl` instruction then copies the contents of register `%bl` into a 32-bit register of the assembler's own choosing, designated as `%0`.

- The parts that begin with a colon (e.g., `: "=r" (flag)`) are *metadata*. For example, the C source code returns the overflow status with the `return` statement:

  ```
  return flag; /* zero is no overflow, non-zero is
  overflow */
  ```

  Recall that assembly routines return a value in the register `%rax` or its lower half `%eax`. The `"=r" (flag)` clause signals that `flag` in C is `%rax` in assembly code. If the assembler is in an optimizing mood, it should make `%rax` the register designated by `%0` `shown previously`: `%rax` serves as the overflow flag returned to the caller. The middle-colon section is empty here but in general could contain other inputs to the assembly code. The third-colon section recommends that the 64-bit register `%rbx` be used as scratchpad.

When the program executes (see the `main` function in the following), the output is

```
No overflow on 16 * 48: returned product == 768
Overflow on INT_MAX * INT_MAX: returned product == 1
```

The in-line assembly code does its job.

The focus now shifts to the C code, in particular to the `safe_mult` function. Here is the challenge:

- The `safe_mult` function needs to signal its caller whether overflow has occurred. The returned value is used for this purpose: zero (*false*) means *no* overflow, and nonzero (*true*) means overflow.
- How, then, is the product of the first two arguments to be returned? The approach taken here is to have `safe_mult` called with three arguments:

  ```
  int safe_mult(int n1, int n2, int* product); /*
  declaration */
  ```

  The parameters `n1` and `n2` are the numbers to be multiplied, and the parameter `product` points to where the result of the multiplication should be stored. The pointer argument `product` is the address of a variable declared in the caller `main`.

Listing 3-7. Using a pointer argument to hold a return value

```
void main() {
  int n;
  char* msg;

  /* no overflow */
  int flag = safe_mult(16, 48, &n);
  msg =  (!flag) ? "Overflow on 16 * 48" : "No overflow on
  16 * 48";
  printf("%s: returned product == %i\n", msg, n);
```

```
  /* overflow */
  flag = safe_mult(INT_MAX, INT_MAX, &n);
  msg = (!flag) ? "Overflow on INT_MAX * INT_MAX" : "No
  overflow on INT_MAX * INT_MAX";
  printf("%s: returned product == %i\n", msg, n);
}
```

The `main` function for the *safeMult* program (see Listing 3-7) makes two calls against the function. The first, with 16 and 48 as the values to be multiplied, does not cause overflow. The second call, however, passes `INT_MAX` as both arguments, with overflow as the expected and, because of `safe_mult`, the now detected overflow.

3.7. The `void*` Data Type and `NULL`

The term `void` is not the name of a data type, although C syntax implies as much:

```
void main() { /* body */ } /* void seems to be the
                           return type */
int some_function(void);   /* same as: int some_function(); */
```

This definition of `main` suggests that the function returns a `void` in the same way that another version of `main` returns an `int`; but the suggestion is misleading. The `void` is really shorthand for *returns no value* and so is not a data type in the technical sense. For instance, a variable cannot be declared with `void` as the type:

```
void n; /** ERROR: void is not a type **/
```

In the second example shown previously, the `void` in the declaration of `some_function` signals only that this function expects no arguments; the `void` once again is *not* a type, but another way of writing an empty argument list.

There is a very important data type in C that has `void` in its name: `void*`, or *pointer to void*. This type is a *generic pointer type*: any other pointer type can be converted to and from `void*` without explicit casting. Why is this useful? A short example provides one answer, and the next section provides another.

Consider this array of strings:

```
char* strings[ ] = {"eins", "zwei", "drei", "vier", "fuenf",
"sechs"};
```

The array happens to hold six strings, each of which is a `char*` in C. For example, the first array element is a pointer to the "e" in "eins". To write a loop that traverses this array without going beyond the end requires a count of how many elements are in the array; in this case, there are six.

There is a better, more robust, and more programmer-friendly way to build an array of strings:

```
char* strings[ ] = {"eins", "zwei", "drei", "vier", "fuenf",
"sechs", 0};
```

At first sight, this code looks wrong. An array aggregates elements of the *same* data type, and the last element here appears to be an integer value rather than a `char*` pointer. But the `0` here is `NULL`, a macro defined in the header file *stdlib.h* as follows:

```
#define NULL ((void*) 0) /* 0 cast as a pointer to void */
```

Because `NULL` is of type `void*`, it can occur in an array of *any* pointer type, including the `char*` element type in the `strings` array. By the way, the `0` as shorthand for `NULL` is the only numeric value that would work in this case. Were 987 used instead of 0, for instance, the code segment would not compile. C programmers, in order to save on typing, are fond of using `0` for `NULL`.

Traversing the revised array is now straightforward and illustrates idiomatic C programming:

```
int i = 0;
while (strings[i])                /* short for: while
                                     (strings[i] != NULL) */
  printf("%s\n", strings[i++]);   /* print current string, then
                                     increment i */
```

The loop condition is *true* until `strings[i]` is `NULL`, which is `0`: the value `0` in C is overloaded, and one of the overloads means *false* in a test context. The use of `NULL` to mark the end of pointer arrays is common in C.

A final note is in order. The `NULL` used in this most recent example is *not* the *null terminator* used to mark the end of an individual string. Recall that the string "eins" is represented in C as an array with 8-bit zero at the end as the terminator:

```
+---+---+---+---+--+
| e | i | n | s |\0|   ## \0 is 8-bit zero
----+---+---+---+--+
```

By contrast, the `NULL` that terminates the *strings* array is either a 32-bit zero or a 64-bit zero, depending on whether the machine uses 32-bit or 64-bit addresses. To be sure, the comparison

```
NULL == '\0' /* evaluates to true */
```

evaluates to *true*, but only because the compiler converts the 8-bit null terminator (zero) to the 32-bit or 64-bit zero.

In summary, zero has three specific uses in C beyond 0 as a numeric value:

- In a boolean context (e.g., an `if` or `while` condition), zero means *false*, and nonzero means *true*.

- In a string context, the 8-bit zero (`\0`) is the code for the nonprinting character that marks the end of a string: the *null terminator*.
- In a pointer context, zero is `NULL`, the address-size *null pointer* that points nowhere.

C programmers are fond of idioms that conflate these overloads of zero. The

```
while (strings[i])
```

test from the preceding example is one such idiom.

3.7.1. The `void*` Data Type and Higher-Order Callback Functions

The `void*` type plays an important role in library functions designed to work on arrays of any type. Consider, for example, library functions to initialize, sort, search, and otherwise process arrays. These functions should be *generic* in that they work on arrays of *any* data type. It would be impractical to fashion multiple sort functions, each targeted at a specific type. The task presumably would never be completed.

Among the generic library functions is `qsort`, which can sort an array of `Employee` instances, or `int` instances, or `double` instances, and so on. The first argument to `qsort` is a *pointer* that specifies where, in the array, the sort should begin, which is typically but not necessarily the first element: `qsort` can sort arbitrary *subarrays*, or the whole array, with only small changes to the arguments passed to this function. For now, the other arguments can be ignored, as the emphasis is on the type of first argument to `qsort`. This type is `void*` because it satisfies the requirement that `qsort` should work on any array of any type. Here is how the declaration of `qsort` begins:

```
void qsort(void* start,... /* 4 arguments in all */
```

A full sorting example fleshes out the details of the remaining three arguments.

Listing 3-8. Sorting an array with qsort

```
#include <stdio.h>
#include <stdlib.h> /* rand, qsort */
#define Size 12

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

int comp(const void* p1, const void* p2) {
  int n1 = *((int*) p1);  /* cast p1 to int*, then
  dereference */
  int n2 = *((int*) p2);  /* same for p2 */
  return n2 - n1;         /* descending order */
}

void main() {
  int arr[Size], i;
  for (i = 0; i < Size; i++) arr[i] = rand() % 100; /* values
                                                       < 100 */
  print_array(arr, Size); /* 83 86 77 15 93 35 84 92 49 21
                             62 27 */

  qsort(arr, Size, sizeof(int), comp); /* comp is a pointer to
                                          a function */
  print_array(arr, Size); /* 93 92 86 84 83 77 62 49 35 27
                             21 15 */
}
```

The *sort* program (see Listing 3-8) does the following:

1. Populates an `int` array with pseudorandomly generated values
2. Prints the array
3. Sorts the array in descending order using the library function `qsort`
4. Prints the sorted array

The `qsort` function has a *comparison semantics* used throughout modern programming languages. Here is the full declaration for `qsort`:

```
void qsort(void* start,
           size_t nmemb,
           size_t size,
           int (*comp) (const void*, const void*));
```

The arguments can be clarified as follows:

- The first argument, of type `void*`, points to where in the array the sorting should begin. This is typically, but not necessarily, the start of the array. The `qsort` function can sort only part of array, if required. Because the argument is of type `void*`, any type of array can be sorted using `qsort`.
- The second argument, of unsigned integer type `size_t`, specifies the number of elements to be sorted.
- The third argument (also of type `size_t`) is the `sizeof` each element.

- The fourth argument is a *pointer to a function* that matches this prototype:
  - Returns an `int` value.
  - Takes two arguments of type `const void*`, which are pointers to two elements that `qsort` needs to compare and, perhaps, move. The `const` indicates that the pointers are not used to change the values to which they point.

The critical fourth argument makes `qsort` a *higher-order function*, one that takes a (pointer to a) function as an argument.

A function's name, like an array's name, is a *pointer constant*. A function's name points to the first statement in a function's body; in assembly language, the function's name is thus a label.

The comparison function used in `qsort` can have any name so long as the function matches the prototype. In the *sort* program, the comparison function is named `comp`. The comparison function is a *callback*, a function that a programmer writes for some other function to call, in this case, `qsort` itself. In the course of doing the sort, `qsort` must do pairwise element comparisons in order to determine how to rearrange the array. The sort is destructive in that the sort occurs in place: the array being sorted is rearranged unless it is already sorted.

Here are the details for the comparison. Each argument passed to the comparison function points at an array element. Assume that the first argument points to array element *E1* and the second argument points to array element *E2*. The value returned from the comparison function then has the following semantics:

- If *E1* and *E2* are considered equal, 0 is returned.
- If *E1* is considered to precede *E2*, a negative value is returned (e.g., -1).

- If *E2* is considered to precede *E1*, a positive value is returned (e.g., +1).

These semantics are remarkably simple and flexible. The author of the comparison function determines the details. Here, for review, is the comparison function for the `sort` program:

```
int comp(const void* p1, const void* p2) {
  int n1 = *((int*) p1); /* cast p1 to int*, then
                            dereference */
  int n2 = *((int*) p2); /* same for p2 */
  return n2 - n1;        /* descending order */
}
```

The function's body could be reduced to a single `return` statement, but at the cost of clarity. Since the array being sorted has `int` elements, the `void*` arguments are cast to pointers of type `int*`. Each `int*` pointer then is dereferenced to get the `int` value pointed to. Variables `n1` and `n2` hold these values. Suppose that `n1` is 20 and that `n2` is 99. The returned value of

```
n2 - n1
```

is then 79, a *positive* value signaling that 99 should *precede* 20 in the sorted order. The sort is thus in descending order. If the returned value were changed to

```
n1 - n2
```

then the resulting sort would be in ascending order. If the `int` array had the same values throughout, then 0 would be returned for every comparison, leaving the array unchanged by the sort.

The usefulness of `void*` is undoubtedly evident to programmers from object-oriented languages such as Java and C#. In these languages, a *reference* (pointer) to `Object` can point to anything. Here is a segment of Java to illustrate:

```
Object ptr = new String("Hello, world!"); /* string */
ptr = 99;                                 /* integer: boxed as
                                             new Integer(99) */
ptr = new int[ ] {1, 2, 3, 4};            /* array of
                                             integers */
```

Generic types such as `void*` in C, and `Object` in Java, make languages flexible.

The second code example uses a `typedef` to describe the type of function suitable as an argument to another function. A `typedef` creates an alias for an existing type:

```
typedef unsigned boolean;  /* unsigned is existing type,
                              boolean is the alias */
boolean flag;              /* use the type in a variable's
                              declaration */
```

Pointers to functions, like other C pointers, have data types, and the `typedef` is useful in defining the appropriate type, a type that will satisfy the compiler. It is easy to get a pointer to a function; the function's name is just such a pointer. It can be challenging to pass an appropriate function pointer as an argument in another function.

WHAT'S AN ENUM?

An `enum` (*enumerated type*) gives names to integer values. The enumerated type itself can but need not be named:

```
enum { false, true };                   /* false is 0, true is
                                           1, and so on */
enum TruthValue { true = 1, false = 0 }; /* tagged and explicit
                                           assignments */
```

The enumerated values start at 0 and continue in series unless explicit values are given, as in the second example shown previously. In the second example, `false` would default to 2 if not explicitly assigned 0 as its value. Constructs such as `typedef` and `enum` promote readable code:

```
typedef unsigned boolean;
boolean continue_to_loop = true;
```

The next example uses a `typedef` to specify the prototype of a function passed as an argument to the higher-order `reduce` function. The `reduce` function takes two additional arguments: an array of integer values and the array's length.

Listing 3-9. Another example of pointers to functions

```
/* pointer to function with two arguments (int array and
length), returns an int */
typedef unsigned (*reducer)(unsigned list[], unsigned len);
                                    /* type name is reducer */

unsigned sum(unsigned list[], unsigned len) {
   unsigned sum = 0, i;
   for (i = 0; i < len; i++) sum += list[i];
   return sum;
}

unsigned product(unsigned list[], unsigned len) {
   unsigned prod = 1, i;
   for (i = 0; i < len; i++) prod *= list[i];
   return prod;
}
```

```
unsigned reduce(reducer func, unsigned list[], unsigned len) {
/* 1st arg: ptr to func */
   return func(list, len); /** invoking a function in the
   usual way **/
}
```

The *reducer* program (see Listing 3-9) has two functions, `sum` and `product`, that reduce a list of integers to a single value, in this case a sum and product, respectively. The third function is higher order and named `reduce`. This function takes a (pointer to a) function as its first argument, an array of values as its second, and the array's length as its third.

The `typedef` in the *reducer* program is the tricky part:

```
typedef unsigned (*reducer)(unsigned list[], unsigned len);
```

The data type alias is `reducer`, and it can point to any function that meets these conditions:

- The function takes two arguments: an array of `unsigned` integers and a single `unsigned` integer (the length) in that order.
- The function returns an `unsigned` integer.

The declaration of the `reduce` function uses the `typedef` data type in the first argument position:

```
unsigned reduce(reducer func, unsigned list[], unsigned len);
```

Applying a particular `reducer` function, in this case `sum` or `product`, through the function pointer `func` requires no special syntax:

```
unsigned n = func(list, len); /* invoking a function through a
pointer argument func */
```

Normally, a function is invoked using its name, a pointer *constant*; in this case, a function is invoked using a pointer *variable* instead, `func` of type `reducer`. Invoking `reduce` also is straightforward:

```
reduce(sum, nums, Size);     /* sum is a function */
reduce(product, nums, Size); /* product is a function */
```

Listing 3-10. The `main` function in the reducer program

```
#include <stdio.h>
#define Size 30

int main() {
   unsigned nums[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                      11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
                      21, 22, 23, 24, 25, 26, 27, 28, 29, 30};

   printf("Sum of list: %i\n", reduce(sum, nums, Size));
   /* 465 */
   printf("Product of list: %i\n", reduce(product, nums, Size));
   /* 1,409,286,144 */
   return 0;
}
```

The `main` function in the *reducer* program (see Listing 3-10) shows two calls to the `reduce` function: the first using `sum` as its first argument and the second using `product` as this argument.

The *reducer* program illustrates that higher-order functions are routine in C. Such functions, used judiciously, make programs easier to understand. The `reduce` function maps a list of integers to a single value, and the first argument—the function pointer—specifies the kind of mapping involved, in this case reducing the list to either a sum or a product.

3.8. Structures

Arrays aggregate variables of the *same* data type, whereas structures can aggregate variables of *different* types. The variables in a structure are known as its *fields*. There can be arrays of structures, and structures that embed arrays and even other structures. As a result, programmer-defined data structures can be arbitrarily rich.

The syntax of structures can be introduced in short code examples. Here's a start:

```
struct {
   int n;
   double k;
} s1;

s1.n = -999;
s1.k = 44.4;
```

The data type is `struct {...}`, and variable `s1` is of this structure type; hence, `s1` has two fields: an `int` named `n` and a `double` named `k`. The member operator, the period, is used to access the structure's *fields*, in this case the `int` field `n` and the `double` field `k`. The compiler is *not* bound to lay out storage for the fields in a way that matches the structure's declaration. Although field `n` occurs before field `k` in the structure declaration shown previously, this may not be the case after compilation. The member operator should be used to access the fields by name.

A second code segment adds a *tag* to the structure so that the structure type has a name:

```
struct TwoNums { /* TwoNums is the tag */
   int n;
   double k;
};
```

```
struct TwoNums s2; /* the data type is struct TwoNums: struct
                      plus the tag */
```

A third example shows the popular approach, which uses a `typedef` to name a structure type:

```
typedef struct {  /* tag is optional, could be same as typedef
                     name TwoNums */
   int n;
   double k;
} TwoNums;        /* TwoNums is now an alias for this
                     struct type */

TwoNums s3;       /* Note: the word 'struct' is not needed
                     anymore */
```

The name of a structure, unlike the name of an array, is *not* a pointer. Caution is thus required when structures are passed as arguments to functions.

Listing 3-11. Passing a structure as an argument

```
#include <stdio.h>
#define Size 100000

typedef struct { /* Declare the structure using a typedef for
                    convenience. */
  double nums1[Size]; /* 8 bytes per double */
  double nums2[Size]; /* 8 bytes per double */
  int nums3[Size];    /* 4 bytes per int */
  float nums4[Size];  /* 4 bytes per float */
  float nums5[Size];  /* 4 bytes per float */
  int n; /* for demo purposes */
} BigNumsStruct;
```

```
void good(BigNumsStruct* ptr) {
  printf("%lu\n", sizeof(ptr));        /* 8 on my machine */
  printf("%i %i\n", (*ptr).n, ptr->n); /* -9876 -9876 */
}

void bad(BigNumsStruct arg) {
  printf("Argument size is: %lu\n", sizeof(arg)); /* 2,800,008
                                                     bytes */
}

void main() {
  BigNumsStruct bns;
  bns.n = -9876;
  bad(bns);   /** CAUTION **/
  good(&bns); /* right approach: pass an address */
}
```

The *bigStruct* program (see Listing 3-11) declares a structure, five of whose fields are large arrays. The function `main` then creates a local variable `bns` of this structure type and passes the variable to function `bad`. Recall that C uses *call by value* in function calls; hence, a byte-per-byte *copy* of `bns` is passed to function `bad`, a copy that is about 2.8MB (*megabytes*) in size.

By contrast, `main` then calls function `good` by passing the address of `bns` rather than a copy of this `BigNumsStruct` instance. The address is 4 or 8 bytes, depending on whether the underlying machine uses 32-bit or 64-bit addresses.

The second `printf` in function `good` shows how C syntax supports two ways of accessing structure fields:

- The first way uses the member operator (the period) but is clumsy because the expression contains the pointer `ptr`:

  ```
  (*ptr).n
  ```

The parentheses are necessary because the period has higher precedence than the star. Without the parentheses, the deference operator would apply to `ptr.n`, but `n` is a *nonpointer* field.

- The second way uses the arrow operator (a minus symbol followed by a greater-than symbol):

  ```
  ptr->n
  ```

This syntax is cleaner and is idiomatic in C.

In the *bigStruct* program, the `sizeof of` the `BigNumsStruct` is reported to be 2,800,008 bytes. The arrays account for 2,800,000 of these bytes, and `int` field `n` requires only 4 bytes. What accounts for the extra 4 bytes? A simpler example explains.

Consider this structure:

```
struct {
   int n;    /* sizeof(int) == 4 */
   char c;   /* sizeof(char) == 1 */
   double d; /* sizeof(double) == 8 */
} test;
```

The minimum storage required for a variable such as `test` is 13 bytes, but most implementations would report `sizeof(test)` to be 16 rather than 13. Modern C compilers typically align storage for scalar variables on multibyte boundaries, for example, on 4-byte (32-bit) boundaries. The `char` field named `c` thus is implemented with four bytes rather than just one.

3.8.1. Sorting Pointers to Structures

An earlier discussion noted that *pointers to pointers* are common in C. The current discussion, on structures, is an opportunity to show how such pointers can be put to use.

Imagine an array of structure elements, perhaps of `Employee` instances, each of which is roughly 8KB (*kilobytes*) in size and all of which differ in whatever field (for instance, an ID field) might be used as a sort key. Suppose, then, that the `Employee` array is to be sorted by employee ID.

Sorting the `Employee` array with `qsort` would require moving 8KB chunks around in the array in order to get the desired sorted order. Such moves are inefficient, given the chunk size. A first principle of programming is not to move large data chunks unless the reasons are compelling.

There is another way, one that brings pointers to pointers into the picture. Given an array of relatively large structure elements, it is straightforward to create an *index array* for the `Employee` array, where the *index array* is a second array whose elements are pointers to elements in the first array:

```
   0x0004     0x1f44      0x3e84         ## addresses, 8KB
bytes or sizeof(Employee) apart
+-----------+-----------+-----------+
| Employee1 | Employee2 | Employee3 |... ## 8KB Employee elements
+-----------+-----------+-----------+

+--------+--------+--------+
| 0x0004 | 0x1f44 | 0x3e84 |...## index array for Employee array
+--------+--------+--------+
```

In this depiction, the elements in the top or data array are `Employee` instances, whereas the elements in the bottom or index array are `Employee*` pointers. In short, each index element points to an `Employee` element. The addresses in the index array are 8KB (*kilobytes*) apart because `sizeof(Employee)` is 8,000 bytes, and addresses are of bytes. Given the significant difference in size between elements in the `Employee`

array and the index array, it would be more efficient to sort the index than the Employee array. Indeed, several index arrays might be created and then sorted to obtain various orders: employees sorted by ID, by salary, by years in service, and so on. To print or otherwise process the Employee elements in the desired order, a program would traverse one of the indexes. The Employee elements would remain in their initial positions.

This approach does bring a challenge to the programmer, however. Consider the arguments passed to the qsort comparison function when an index is sorted on some Employee feature such as ID or years in service. Each such argument is of type const void*, which in this case is really of type Employee**: a pointer to a pointer to an Employee. The arguments to the comparison function thus must be dereferenced *twice* in order to access the Employee feature to be used in the comparison. A full code example goes into the details.

Listing 3-12. Sorting pointers rather than data

```
#include <stdio.h>
#include <stdlib.h> /* rand */
#define SizeS 1000
#define SizeA 100

typedef struct {
  double nums[SizeS]; /* 8 bytes per */
  int n;              /* for demo purposes */
} BigNumsStruct;

int comp(const void* p1, const void* p2) {
  BigNumsStruct* ptr1 = *((BigNumsStruct**) p1);
                                  /* p1 points to a pointer */
  BigNumsStruct* ptr2 = *((BigNumsStruct**) p2);
                                  /* p2 points to a pointer */
```

```
  return ptr1->n - ptr2->n;                          /* ascending
                                                        order */
}

void main() {
  BigNumsStruct big_nums[SizeA];
  BigNumsStruct* pointers[SizeA];

  int i;
  for (i = 0; i < SizeA; i++) {
    big_nums[i].n = rand();
    pointers[i] = big_nums + i;   /* base address (big_nums)
                                     plus offset (index i) */
}

qsort(pointers, SizeA, sizeof(BigNumsStruct*), comp);
                                        /** sort the pointers **/
for (i = 0; i < SizeA; i++)
   printf("%i\n", pointers[i]->n);
}
```

The *sortPtrs* program (see Listing 3-12) revises the earlier example of the `BigNumsStruct`. The size of this structure is reduced to a more realistic number, and a local array of such structures is declared, which means that storage for the array comes from the stack. The `int` field named `n` remains and now is initialized to a random value.

Although a `BigNumsStruct` is slimmer than before, its `sizeof` remains an impressive 8,008 bytes on my machine. By contrast, a *pointer* to such a structure instance requires only 8 bytes on the same machine. In the *sortPtrs* program, sorting the `big_nums` array would require moving 8KB (*kilobytes*) chunks, whereas sorting pointers to the elements in this array would require moving only 8-byte chunks. The resulting gain in efficiency is compelling. The `printf` loop at the end confirms that the `pointers` array has been sorted as desired, in ascending order by the `BigNumsStruct` field named `n`.

The cost for this efficiency is a complicated comparison function, again named `comp`. Recall that each argument in the comparison callback is of type `const void*`. Because an array of *pointers* is being sorted, the two arguments to `comp`, named `p1` and `p2`, are indeed pointers to pointers. Each of these pointers is therefore cast to its actual type, `BigNumsStrut**`: a pointer to a pointer to a `BigNumsStruct`. A dereference of each point provides what is needed: a pointer to a `BigNumsStruct`, which then can be used with the arrow operator to access the field `n`. Here, for review, is the body of the comparison function:

```
BigNumsStruct* ptr1 = *((BigNumsStruct**) p1); /* p1 points to
                                                  a pointer */
BigNumsStruct* ptr2 = *((BigNumsStruct**) p2); /* p2 points to
                                                  a pointer */

return ptr1->n - ptr2->n; /* access the field n, sort in
                             ascending order */
```

3.8.2. Unions

There is a specialized type of structure called a `union`, which is designed for memory efficiency. A short example highlights the difference between a `struct` and a `union`.

The following structure has two fields: a `double` and a `long`. The `sizeof(v1)`

```
struct {
  double d;
  long l;
} v1;
```

is 16: both the `double` and the `long` are 8 bytes in size.

By contrast, a `union` with exactly the same fields would require only half the bytes. The `sizeof(v2)`

```
union {
  double d;
  long l;
} v2;
```

is 8 bytes. A `union` provides enough storage for the *largest* of its fields, and all of the fields then share this storage. For example, the `struct` variable `v1` can store both a `double` and a `long` at the same time:

```
v1.d = 44.44;
v1.l = 1234L;
```

By contrast, the `union` variable `v2` stores either the one or the other:

```
v2.d = 44.44; /* the double is stored */
v2.l = 1234L; /* initializing the long overwrites the double */
```

3.9. String Conversions with Pointers to Pointers

Earlier examples illustrated very simple conversions involving basic data types. For example, even the statement

```
char c = 65; /* 65 is ASCII/Unicode for uppercase A */
```

involves a conversion: from the 32-bit `int` constant 65 to the 8-bit `char` value stored in variable `c`. Converting from one single value to another is routine in C: an explicit cast can be used for clarity, but in general, the compiler can be counted on to do the converting without complaint. For example, the compiler does not even warn against this conversion:

```
short n = 3.1415; /* 64-bit floating-point value stored in 16-
                     bit integer variable */
```

The conversion goes from a three-field, 64-bit floating-point source to a two-field, 16-bit signed-integer destination. In examples such as these, explicit casts can be used to enhance clarity, but this remains a recommendation rather than a requirement:

```
char c = (char) 65;
short n = (short) 3.1415;
```

The challenge arises in converting between strings, an *aggregate* rather than a *scalar* type, and other basic types. Because a string in C is an *array*, converting an array to a single integer or floating-point value is nontrivial. C provides library functions to do the heavy lifting.

The *stdlib.h* header file declares functions for converting strings to integers and floating-point values:

```
int atoi(const char* nptr);        /* string to 32-bit int */
long atol(const char* nptr);       /* string to 64-bit long */
long long atoll(const char* nptr); /* string to long long,
                                      probably 64-bits */
float atof(const char* nptr);      /* string to 32-bit float */
```

The `const` qualifier signals that the pointer argument is not used to change the string itself, only to convert the string to a numeric value. The *a* in `atoi` and the others is for ASCII, the default character encoding in C.

None of the *ato* functions are especially helpful in determining why an attempted conversion failed. To that end, the *stdlib.h* header file also includes functions with names that start out with *strto*, for example, `strtol` (string to `long` integer) and `strtod` (string to `double`). The *strto* functions check the string for inappropriate characters and have a mechanism for separating out the converted part of the source string, if any, from the rest. A code example clarifies.

Listing 3-13. Converting strings to numeric values

```
#include <stdio.h>
#include <stdlib.h> /* atoi, etc. */

void main() {
  const char* s1 = "27";
  const char* s2 = "27.99";
  const char* s3 = " 123";      /* whitespace to begin */
  const char* e1 = "1z2q";      /* bad characters */
  const char* e2 = "4m3.abc!#"; /* ditto */

  printf("%s + 3 is %i.\n",      s1, atoi(s1) + 3);
  /* 27 + 3 is 30. */
  printf("%s + 3 is %f.\n",      s2, atof(s2) + 3.0);
  /* 27.99 + 3 is 30.990000. */
  printf("%s to int is %i.\n",   s3, atoi(s3));
  /* 123 to int is 123. */
  printf("%s to int is %i.\n",   e1, atoi(e1));
  /* 1z2q to int is 1. */
  printf("%s to float is %f.\n", e2, atof(e2));
  /* 4m3.abc to float is 4.000000. */

  char* bad_chars = NULL;
  const char* e3 = "9876 !!foo bar";
  long num = strtol(e3, &bad_chars, 10);
  /* 10 is the base, for decimal */
  printf("Number: %li\tJunk: %s\n", num, bad_chars);
  /* Number: 9876 Junk: !!foo bar */
}
```

The *str2num* program (see Listing 3-13) has three examples of strings that convert straightforwardly. The pointers to these are `s1`, `s2`, and `s3`. The string to which `s3` points is the most interesting in that it begins with blanks;

but the *atoi* functions ignore the leading whitespace. The challenging cases are the strings to which `e1` and `e2` point, as these strings contain nonnumeric characters other than whitespace. (Numeric characters include the numerals, the plus and minus signs, and the decimal point.)

For strings with nonnumeric characters such as the sharp sign, the *ato* functions convert until the first such character is encountered and then stop. This is why function `atoi` converts the string "1z2q" to 1: the function converts as long as it can and then halts abruptly on the first inappropriate character. If a string starts with a nonnumeric character, then the *ato* functions return 0:

```
int n = atoi("foo123"); /* n == 0 after the conversion */
```

The *strto* functions are more powerful than their *ato* counterparts, and they use a pointer-to-pointer type to gain this power. Here is the declaration for `strtol`:

```
long int strtol(const char* nptr, char** endptr, int base);
```

The first argument is again a pointer to the *source* string, and the return value is a `long`. The last argument specifies the base to be used in the conversion: 2 for binary, 10 for decimal, and so on. The middle argument is the tricky one, as its type is pointer-to-pointer-to-`char`. Here, for review, is the code segment in the *str2num* program that sets up and then calls the `strtol` function:

```
char* bad_chars = NULL;
const char* e3 = "9876!!foo bar";
long num = strtol(e3, &bad_chars, 10);
```

The `strtol` function determines where to break the source string to which `e3` points: at the first `!` character. The library function then sets pointer `bad_chars` to this character. In an idiom analyzed earlier, the `strtol` function thus uses an *argument*, in this case the pointer-to-pointer variable `bad_chars`, in order to return a value—the first character (the `!`)

that cannot be used in the string-to-number conversion. The return value for `strtol` is, of course, the converted number. A pointer-to-pointer type allows the `strtol` function to return two pieces of information.

The *ato* and *strto* functions are convenient for converting strings to integer and floating-point types. There is also a more general approach. The `printf` function, for type-sensitive printing, has been used in many examples. This function prints to the *standard output*, the screen by default. The inverse function is `scanf`, which scans the *standard input* (the keyboard by default) for strings that then are converted into the specified type. Two variants of these functions are useful for converting from and to strings: `sprintf`, which prints to a buffer (`char` array) rather than to the standard output, and `sscanf`, which reads from a buffer instead of from the standard input. A code example clarifies.

Listing 3-14. A general approach to converting to and from strings

```
#include <stdio.h>

void main() {
  char* s1 = "123456";
  char* s2 = "123.45";
  int n1;
  float n2;

  /** string to other types: sscanf **/
  sscanf(s1, "%i", &n1); /* address of n1, not n1 */
  sscanf(s2, "%f", &n2); /* address of n2, not n2 */
  printf("%i %f\n", n1 + 3, n2 + 8.7f); /* 123459 132.149994 */

  /** other types to string: sprintf **/
  char buffer[64]; /* stack storage, buffer its address */
  sprintf(buffer, "%i", n1 + 3);
  printf("%s\n", buffer); /* 123459 */
}
```

The *scanPrint* program (see Listing 3-14) illustrates the basics of converting to and from strings using the printing and scanning functions. The *print* and *scan* functions differ markedly in their arguments. The *print* functions (`printf`, `sprintf`, and `fprintf` for printing to a file) take nonpointer *values* as the arguments after the format string. By contrast, the *scan* functions (`scanf`, `sscanf`, and `fscanf` for scanning data from a file) take *pointers* as the arguments after the format string. The scanning functions require a pointer to indicate where a scanned (and perhaps converted) value should be stored. For functions in both families, the format string specifies the desired type for either printing or scanning. As even this short code example shows, `sprintf` and `sscanf` provide a general-purpose solution to the problem of converting to and from strings.

Finally, the header file *ctype.h* has various functions for determining properties of individual characters. For instance, the library function `isdigit(c)` checks whether character `c` is a decimal digit, function `isprint(c)` checks whether character `c` is printable, and so on.

3.10. Heap Storage and Pointers

A program in execution (*process*) has access to three areas of memory:

- A *static area* that stores string literals, global variables, and executable code. The traditional name for the area that holds the executable code is *text*, as earlier assembly-code examples illustrate; the term *text* is meant to suggest *read-only*, but this static area can store *read/write* variables as well.
- The *stack*, which provides scratchpad storage for parameters and local variables. The stack acts as a backup for CPU registers, which are quite limited in number (e.g., roughly 16 on standard handheld, laptop, and desktop machines).

- The *heap*, which provides storage that the program explicitly allocates and, in the case of C, deallocates. Pointers come into play with heap storage.

The examples so far have not covered the third category, the heap. The compiler figures out how much storage is required for the read-only area and the stack; hence, the details about such storage are determined at compile time—no extra programmer intervention is required. By contrast, the programmer uses designated operators (e.g., `new` in many modern languages) or functions (e.g., `malloc` and its relatives in C) to allocate storage from the heap, an allocation traditionally described as *dynamic* because it is done explicitly at runtime.

The programmer plays a more active role with heap as opposed to stack storage. The compiler determines the mix of stack and CPU registers required for program execution, thereby off-loading this responsibility from the programmer. By contrast, the programmer manages heap storage through system calls to allocate and, in the case of C, to deallocate this storage. A review of stack storage through a code example sets the scene for a code-based analysis of heap storage.

Listing 3-15. Summing an array in C

```
#include <stdio.h>
#define Size 9

int sum_array(int arr[], unsigned n) {
  int sum = 0;
  unsigned i;
  for (i = 0; i < n; i++) sum += arr[i];
  return sum;
}
```

```
void main() {
  int nums[ ] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  int n = sum_array(nums, Size);
  printf("The sum is: %i\n", n); /* The sum is: 45 */
}
```

The *sumArray* program (see Listing 3-15) has two functions, `main` and `sum_array`, each of which needs stack storage for scratchpad. The `main` function has a local array of nine elements, each an `int`; these elements are stored on the stack. This function also has a local variable `n` to store the value returned from a call to the `sum_array` function. Depending on how optimizing the compiler happens to be, variable `n` could be implemented as a CPU register instead of as a stack location.

The `sum_array` function works with a *pointer* to the array declared and populated in `main`, but `sum_array` does need some local storage of its own: the integers `sum` and `i`, the loop counter. Both `sum` and `i` are scalars rather than aggregates, and so CPU registers would be ideal; but the stack is the fallback for the compiler.

The assembly code for the *sumArray* program is generated in the usual way:

```
% gcc -S -O1 sumArray.c ## capital letter O for optimization
level, 1 in this case
```

Here is a quick overview of how the assembly code handles summing the array. The assembly code

- Stores the array `nums` on the stack. The assembly code grows the stack by 56 bytes for this purpose, although only 36 bytes are needed for the nine `int` values.
- Stores the array's size in a CPU register for efficiency.

For readability, the resulting assembly code has been pared down; for instance, most of the directives are omitted. The first code display is the assembly code for `main`, and the following display is the assembly code for `sum_array`. To begin, however, a look at the syntax for pointers in assembly code will be useful.

Recall the assembly opcode `movq`, which copies 64 bits (a *quadword*) from a source to a destination:

```
movq $0, %rax   ## copy zero into %rax
```

A comparable C statement is

```
unsigned long n = 0; /* a long is 64 bits */
```

Consider a more complicated example:

```
movq $1, (%rax)
```

The parentheses are the *dereference* operator in assembly code. Accordingly, this statement implies that `%rax` holds an *address*, and 1 is to be copied to wherever `%rax` points, not into `%rax` itself. In C, a counterpart would be

```
*ptr = 1; /* copy 1 to where ptr points, not into ptr itself */
```

A common variant of pointer syntax in assembly language is

```
movq $1, 16(%rax)
```

The parentheses with an integer value to the left indicate *base-displacement* addressing: inside the parentheses is the *base address*, in this case the contents of `%rax`. To the left of the left parenthesis is the *displacement*, the number of bytes added to the base address. (The displacement can be positive or negative.) In C, a counterpart would be

```
*(ptr + 16) = 1; /* assuming ptr is of type char* because a
                    char is a byte */
```

With this background, the assembly code for the function `main` in the *sumArray* program should make sense.

Listing 3-16. The assembly code for `main` in the sumArray program

```
.LC0:                          ## address of format string
   .string "The sum is: %i\n" ## format string
main:
   subq $56, %rsp    ## grow the stack by 56 bytes (stack grows
                        high to low)
   movl $1, (%rsp)   ## store 1 to where the stack pointer
                        points (the TOP)
   movl $2, 4(%rsp)  ## store 2 four bytes _up_
   movl $3, 8(%rsp)  ## and so on
...
   movl $9, 32(%rsp) ## 9 is stored 32 bytes up from the
                        stack pointer
   movl $9, %esi     ## this 9 is Size: put in a CPU 32-bit
                        register %esi
   movq %rsp, %rdi   ## copy stack pointer in %rdi, which now
                        points to 9 in the array
   call sum_array    ## call the subroutine
   movl %eax, %edx   ## save the value returned from sum_array
   movl $.LC0, %esi  ## copy address of format string into %esi
   movl $1, %edi     ## copy 1 into %edi: number of values to
                        format, 1 in this case
   movl $0, %eax     ## clear %eax for the print routine
   call __printf_chk ## call print routine (special arg-
                        checking version of printf)
   addq $56, %rsp    ## restore the stack pointer by reclaiming
                        the 56 bytes
   ret               ## return to caller in exec family
```

The high points of the assembly code for the `main` block (see Listing 3-16), the assembly-language counterpart of the `main` function in C, can be summarized as follows:

- The block begins by growing the scratchpad storage on the stack: 56 is *subtracted* from the 64-bit stack pointer `%rsp`, which has the effect of growing the stack scratchpad by 56 bytes because the Intel stack grows from high to low addresses. Moving the stack pointer `%rsp` *down* by 56 bytes means, in other words, that there are now 56 newly available bytes *above* where the stack pointer currently points. Shrinking the scratchpad storage on the stack is done by *adding* to the stack pointer, as occurs in the second-to-last statement in the `main` block:

  ```
  addq $56, %rsp ## cleanup from the earlier subq
  %56, %rsp
  ```

- The nine-integer array elements 1,2,...,9 in the array `nums` from the C code are stored on the stack. Most of the values are stored *up* from the stack pointer. For example, 1 is stored at where the stack pointer currently points, 2 is stored 4 bytes *up* from this position at `4(%rsp)`, and so on. In general, the compiler stores arrays on the stack, even very small arrays. There are simply too few general-purpose registers to store arrays, and addressing array elements is simplified by having these elements be stored contiguously. Registers are used for scalar values, not for aggregates.

- The array's size, 9, is not stored on the stack, but rather in the 32-bit CPU register `%esi`. Recall that on a 64-bit machine, the name `%esi` refers to the lower-order 32 bits of the 64-bit register `%rsi`. The `sum_array` routine accesses the array's size from register `%esi`.

- The value returned from `sum_array` in 32-bit register `%eax` is copied to register `%edx`, the address of the format string is copied to register `%esi`, and the number of values to be formatted (in this case, one) is copied into register `%edi`. At this pointer, the `main` module is ready to call the print routine `printf_chk`, which does an integrity check on the arguments, where `chk` stands for "check." As the example shows, the underscore can be used even to start an identifier.
- After shrinking the stack back to its size before the call to `main`, the `main` routine returns to its caller. Recall that `main` in the C source does *not* return a value; hence, the assembly routine does not place a value in `%eax` immediately before returning.

Listing 3-17. The assembly code for `sum_array`

```
sum_array:
   testl %esi, %esi  ## is the array size 0?
   je .L4            ## if so, return to caller
   movl $0, %edx     ## otherwise, set loop counter to 0
   movl $0, %eax     ## initialize sum to 0
.L3:
   addl (%rdi,%rdx,4), %eax  ## increment the running sum by
                                the next value (sum += arr[i])
   addq $1, %rdx             ## increment loop counter by 1
                                (integer)
   cmpl %edx, %esi           ## compare loop counter with
                                array size
   ja .L3                    ## keep looping if size is bigger
                                (ja = jump if above)
```

```
   rep ret           ## otherwise, AMD-specific version of ret
                        for return
.L4:                 ## return 0 as the sum because array
                        is empty
   movl $0, %eax     ## copy 0 into returned-value register
   ret               ## return to caller
```

The `sum_array` routine in assembly code (see Listing 3-17) is complicated because of the control structure. The code basically handles two cases:

- If the array's size is zero (the array is empty), then return 0.
- Otherwise, initialize a loop counter (32-bit register `%edx`) to 0, and loop until the array's size is no longer greater than the loop counter. The running sum is stored in 32-bit register `%eax`, and `%eax` also serves as the returned-value register.

Several points about the code deserve mention. For one thing, the code sometimes references the 64-bit register `%rdx` but sometimes references only the lower 32 bits of this register under the name `%edx`. This can be confusing but works just fine because the upper-order bits in register `%rdx` have been zeroed out.

Another point of interest is the most complicated instruction in the `sum_array` routine:

```
addl (%rdi,%rdx,4), %eax   ## in C: sum += arr[i]
```

First, consider the instruction that follows the `addl` instruction:

```
addq $1, %rdx   ## in C: i = i + 1
```

This instruction updates the loop counter `%rdx` by one *integer*, not by 4 bytes. Accordingly, the `addl` instruction's first operand is the expression `(%rdi,%rdx,4)`. Register `%rdi` points to the *start* of the array; in the C code, this is the parameter `arr` in the function `sum_array`. The *offset* from this base address is `%rdx` × 4, where `%rdx` is the loop counter (in C, the index `i`) and 4 is `sizeof(int)`.

The assembly code confirms that the stack requirements for the *sumArray* program are determined at *compile time*. The stack management is thus automatic from the programmer's perspective: the programmer declares local variables and parameters, makes a function call, executes a *print* statement, and so on. The compiler manages the details when it comes to providing scratchpad storage on the stack and, in this example, in CPU registers as well.

This analysis of the *sumArray* program sets up a contrast between stack and heap storage. C has functions for allocating heap storage, with the `malloc` and the `calloc` functions as the primary ones. There is also a `realloc` function for growing or shrinking previously allocated heap memory. The `free` function deallocates the memory allocated by any of these functions. The general rule for avoiding *memory leaks* is this: for every `malloc` or `calloc`, there should be a matching `free`. The programmer is fully responsible for the calls to these functions. A first code example covers the basics.

Listing 3-18. Basic heap allocation and deallocation

```
#include <stdio.h>
#include <stdlib.h> /* malloc, calloc, realloc */
#include <string.h> /* memset */
#define Size 20

void dump(int* ptr, unsigned size) {
  if (!ptr) return; /* do nothing if ptr is NULL */
  int i;
```

```
  for (i = 0; i < size; i++) printf("%i ", ptr[i]);
  /* *(ptr + i) */
  printf("\n");
}

void main() {
  /* allocate */
  int* mptr = malloc(Size * sizeof(int)); /* 20 ints, 80
                                             bytes */
  if (mptr) /* malloc returns NULL (0) if it cannot allocate
            the storage */
    memset(mptr, -1, Size * sizeof(int)); /* set each byte
                                             to -1 */
  dump(mptr, Size);

  /* realloc */
  mptr = realloc(mptr, (Size + 8) * sizeof(int)); /* request
                                                     8 more */
  if (mptr) dump(mptr, Size + 8);
  /* deallocate */
  free(mptr);

  /* calloc */
  mptr = calloc(Size, sizeof(int)); /* calloc initializes the
                                       storage to zero */
  if (mptr) {
    dump(mptr, Size);
    free(mptr);
  }
}
```

The program *memalloc* (see Listing 3-18) shows the basic API for allocating and deallocating memory from the heap. The simplest and most basic function is `malloc`, which tries to allocate the number of bytes given as its single argument. The return type from `malloc` is the same for `calloc` and `realloc`:

- If the memory can be allocated, a pointer to the first byte is returned.
- If the memory cannot be allocated, `NULL` is returned.

The `malloc` function could be used to allocate as little as 1 byte but typically is used to allocate aggregates. In the case of `malloc`, the allocated storage is *not* initialized. The *memalloc* program therefore initializes the allocated memory to -1 by using the `memset` library function:

```
memset(mptr, -1, Size * sizeof(int)); /* mptr returned from
                                         malloc */
```

This function takes three arguments: a pointer to the storage to be initialized, the value to be stored in each *byte*, and the number of bytes to be initialized. The `memset` function is yet another library routine that works at the byte level.

The `calloc` function takes two arguments: the first is the number of *elements* to allocate (e.g., 10), and the second is the `sizeof` each element (e.g., 4 for an `int`). The `calloc` function thus can be used to allocate storage for multibyte types such as `int` and `double`. This function, unlike `malloc`, initializes the allocated storage to all 0s. In general, `malloc` is faster than `calloc` because `malloc` does no memory initialization.

The `realloc` function can be used to grow or shrink previously allocated storage. In this example, the function is used to grow the allocated storage by 8 × `sizeof(int)` bytes. If `realloc` succeeds, it leaves the previously allocated storage unchanged and adds or removes the requested number of bytes. In the *memalloc* program, `realloc` is called

to request an additional 32 bytes (8 `int` values) to a collection of 20 `int` values already initialized to -1; hence, the original bytes still have -1 as their value after the reallocation, but the added bytes have arbitrary values.

As the name suggests, the `free` function deallocates storage allocated with the `malloc` and `calloc` functions. The `realloc` function presupposes a previous call to one of these other functions. To avoid memory leaks, it is critical for a program to `free` explicitly allocated storage.

DOES C HAVE GARBAGE COLLECTION?

No. Library functions such as `malloc` and `calloc` allocate specified amounts of storage from the heap, but the programmer then is responsible for explicitly deallocating (*freeing*) this heap storage. Allocation without deallocation causes *memory leaks*, which can dramatically degrade system performance. Freeing no longer needed heap storage is, indeed, one of the major challenges in writing sound C programs.

3.11. The Challenge of Freeing Heap Storage

Recall the rule of thumb for freeing heap storage: for every `malloc` or `calloc`, there should be a `free`. Putting the rule into practice can be challenging, in particular when dealing with functions that return a pointer to a structure that, in turn, has, among its fields, pointers to heap storage. In short, the heap storage allocation may be *nested*. If the allocation is nested, then the freeing should be so as well. The documentation on library functions is worth reading carefully, in particular for functions that return a pointer to heap storage. There are different ways for memory-allocating library functions to guard against memory leakage, for example, by providing a utility function that does the freeing to whatever level is appropriate.

Two code examples get into the details of the challenge. The first example focuses on how to return an aggregate to a caller, and the second focuses on nested freeing.

Listing 3-19. Three ways of returning a string to a caller

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define BuffSize 128

void get_name1(char buffer[ ], size_t len) {  /* safest: buffer
                                                 passed in
                                                 as arg */
  strncpy(buffer, "Gandalf", len);            /* user-supplied
                                                 buffer */
}

void* get_name2() {                          /* ok, but invoker
                                                 must free */
  void* ptr = malloc(BuffSize + 1);
  if (!ptr) return 0;
  strcpy(ptr, "Sam");
  return ptr;
}

char* get_name3() {                          /* VERY BAD
                                                 (compiler
                                                 warning) */
  char buffer[BuffSize + 1];
  strcpy(buffer, "Frodo");
  return buffer;
}
```

The *getname* program (see Listing 3-19) contrasts three ways to return a string—an aggregate—from a function. The compiler generates an apt warning about one of the ways. Three functions represent the three different approaches. For each approach, imagine that a user is prompted for, and then enters, a name. To keep the code short, the example hard-wires the names. Here is a summary of the three approaches, with recommendations:

- The `get_name1` function represents the safest approach. The function takes two arguments: an array to hold the name and the array's length. The function then uses the library function `strncpy` to copy a name into this array. The `n` in `strncpy` specifies the *maximum* number of characters to be copied, thereby protecting against the notorious *buffer overflow* problem. A buffer overflow occurs if the array is not big enough to hold all of the elements placed in it. In the case of `get_name1`, the *invoker* of the function is responsible for providing a buffer at least as big as the `len` argument specifies. The first three lines of `main` illustrate a proper call to `get_name1`.

  A cautionary note is in order. Suppose that the first two lines of `main` change from

  ```
  char buffer[BuffSize + 1];
                    /* + 1 for null terminator */
  get_name1(buffer, BuffSize);
  ```

  to

  ```
  char* buffer; /* storage for a pointer, but not
  for any characters pointed to */
  get_name1(buffer, BuffSize);
                /* false promise: the buffer's
                   length is zero */
  ```

The *getname* program still compiles because the compiler treats these two data types as being equivalent:

```
char* buffer    ## the argument's type in main
char buffer[ ]  ## the first parameter's type in get_name1
```

Nonetheless, the program is likely to crash at runtime because there is no storage provided for the characters in the string; there is storage only for a single *pointer* to a `char`. Increasing the length of the string increases the likelihood of a crash.

- The `get_name2` function takes no arguments and instead allocates heap storage to store a string of `BuffSize` characters, where `BuffSize` is a macro defined as 64; a pointer to this storage is returned. The call to `malloc` requests an additional byte for the null terminator, so `BuffSize + 1` bytes in all. The `get_name2` function returns `ptr`, which holds the value returned from `malloc`. (If `malloc` returns `NULL`, so does `get_name2`.) This approach makes the caller, in this case `main`, responsible for *freeing* the allocated storage. There is a division of labor: one function allocates the required heap storage, but a different function (its invoker) must `free` these allocated bytes when they are no longer needed.
- The `get_name3` function is done badly, and the compiler points out the shortcoming. The function declares a *local* variable `buffer` of `BuffSize + 1` bytes. This, in itself, is fine. The function then returns the array's name—a *pointer* to the first `char` in the array. This is risky because the storage for the array comes from the *stack*, and that very area of the stack is open for reuse once `get_name3` returns. Some other function might place other data in this very area. The general principle is clear: *never return a pointer to local storage*.

Listing 3-20. Calling the three functions in the getname program

```
/** headers and macro above **/
void main() {
  char buffer[BuffSize + 1];          /* + 1 for null
                                         terminator */
  get_name1(buffer, BuffSize);
  printf("%s\n", buffer);

  void* retval2 = get_name2();
  printf("%s\n", (char*) retval2);    /* cast for the %s */
  free(retval2);                      /* safeguard against
                                         memory leak */

  const char* retval1 = get_name3(); /* not a good idea */
  printf("%s\n", retval1);           /* unpredictable output */
}
```

The `main` function for the *getname* program (see Listing 3-20) declares a `char` buffer, which is used in the call to function `get_name1`. The responsibility falls squarely on the caller to provide enough storage for the string to be stored. The second argument, `BuffSize`, guards against buffer overflow because the `char` array is of size `BuffSize + 1`, with the added byte for the null terminator.

The call to `get_name2` returns a pointer to the heap storage provided for the name. In this case, the `main` function does call `free`, but the logic is complicated: one function allocates, another function frees. The approach works, but it is error-prone.

The last call, to `get_name3`, provokes a compiler warning because a pointer to *local storage* is being returned to `main`. In this case, the storage for the name is local to the call frame for `get_name3`. Once the function `get_name3` returns to `main`, the call frame for `get_name3` should not be accessed. It is unpredictable whether this third approach works.

3.12. Nested Heap Storage

It is relatively straightforward to handle *nonnested* cases of allocating and freeing, as in the previous examples of heap storage. Here is a review of the pattern:

```
int* some_nums = malloc(5000 * sizeof(int));
/* ... application logic ... */
free(some_nums);
```

This code segment allocates heap storage for 5,000 `int` values, does whatever logic is appropriate, and then frees the storage. The challenge increases when, for example, structure instances are allocated from the heap—and such instances contain fields that are themselves pointers to heap storage. If the heap allocation is *nested*, the freeing must be nested as well.

As a common example of the challenge, C has various library functions that return a pointer to heap storage. Here is a typical scenario:

1. The C program invokes a library function that returns a pointer to heap-based storage, typically an aggregate such as an array or a structure:

   ```
   SomeStructure* ptr = lib_function(); /* returns pointer
                                           to heap storage */
   ```

2. The program then uses the allocated storage.

3. For cleanup, the issue is whether a single call to `free` will clean up all of the heap-allocated storage that the library function allocates. For example, the `SomeStructure` instance may have fields that, in turn, point to heap-allocated storage. A particularly troublesome case would be a dynamically allocated array of structures, each of which has a field pointing to more dynamically allocated storage.

The next code example (see Listing 3-21) illustrates the problem and focuses on how to design a library that safely provides heap-allocated storage to clients.

Listing 3-21. Nested heap storage

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

typedef struct {
  unsigned id;
  unsigned len;
  float* heap_nums;
} HeapStruct;
unsigned structId = 1;

HeapStruct* get_heap_struct(unsigned n) {
  /* Try to allocate a HeapStruct. */
  HeapStruct* heap_struct = malloc(sizeof(HeapStruct));
  if (NULL == heap_struct) /* failure? */
    return NULL;           /* if so, return NULL */

  /* Try to allocate floating-point aggregate within
  HeapStruct. */
  heap_struct->heap_nums = malloc(sizeof(float) * n);
  if (NULL == heap_struct->heap_nums) { /* failure? */
    free(heap_struct);                  /* if so, first free
                                           the HeapStruct */
    return NULL;                        /* then return NULL */
  }

  /* Success: set fields */
  heap_struct->id = structId++;
  heap_struct->len = n;
```

```
  return heap_struct; /* return pointer to allocated
  HeapStruct */
}

void free_all(HeapStruct* heap_struct) {
  if (NULL == heap_struct)  /* NULL pointer? */
    return;                 /* if so, do nothing */

  free(heap_struct->heap_nums); /* first free encapsulated
                                   aggregate */
  free(heap_struct);            /* then free containing
                                   structure */
}

int main() {
  const unsigned n = 100;
  HeapStruct* hs = get_heap_struct(n); /* get structure with N
                                          floats */

  /* Do some (meaningless) work for demo. */
  unsigned i;
  for (i = 0; i < n; i++) hs->heap_nums[i] = 3.14 + (float) i;
  for (i = 0; i < n; i += 10) printf("%12f\n", hs->heap_nums[i]);

  free_all(hs); /* free dynamically allocated storage */

  return 0;
}
```

The *nestedHeap* example (see Listing 3-21) centers on a structure `HeapStruct` with a pointer field named `heap_nums`:

```
typedef struct {
  unsigned id;
  unsigned len;
  float* heap_nums; /** pointer **/
} HeapStruct;
```

The function get_heap_struct tries to allocate heap storage for a HeapStruct instance, which entails allocating heap storage for a specified number of float variables to which the field heap_nums points. The result of a successful call to get_heap_struct can be depicted as follows, with hs as the pointer to the heap-allocated structure:

```
hs-->HeapStruct instance
        id
        len
        heap_nums-->N contiguous float elements
```

In the get_heap_struct function, the first heap allocation is straightforward:

```
HeapStruct* heap_struct = malloc(sizeof(HeapStruct));
if (NULL == heap_struct)  /* failure? */
  return NULL;            /* if so, return NULL */
```

The sizeof(HeapStruct) includes the bytes (four on a 32-bit machine, eight on a 64-bit machine) for the heap_nums field, which is a pointer to the float elements in a dynamically allocated array. At issue, then, is whether the malloc delivers the bytes for this structure or NULL to signal failure; if NULL, the get_heap_struct function returns NULL to notify the caller that the heap allocation failed.

The second attempted heap allocation is more complicated because, at this step, heap storage for the HeapStruct has been allocated:

```
heap_struct->heap_nums = malloc(sizeof(float) * n);
if (NULL == heap_struct->heap_nums) { /* failure? */
  free(heap_struct);                  /* if so, first free the
                                         HeapStruct */
  return NULL;                        /* and then
                                         return NULL */
}
```

The argument `n` sent to the `get_heap_struct` function indicates how many `float` elements should be in the dynamically allocated `heap_nums` array. If the required `float` elements can be allocated, then the function sets the structure's `id` and `len` fields before returning the heap address of the `HeapStruct`. If the attempted allocation fails, however, two steps are necessary to meet best practice:

1. The storage for the `HeapStruct` must be freed to avoid memory leakage. Without the dynamic `heap_nums` array, the `HeapStruct` is presumably of no use to the client function that calls `get_heap_struct`; hence, the bytes for the `HeapStruct` instance should be explicitly deallocated so that the system can reclaim these bytes for future heap allocations.
2. `NULL` is returned to signal failure.

If the call to the `get_heap_struct` function succeeds, then freeing the heap storage is also tricky because it involves two `free` operations in the proper order. Accordingly, the program includes a `free_all` function instead of requiring the programmer to figure out the proper two-step deallocation. For review, here's the `free_all` function:

```
void free_all(HeapStruct* heap_struct) {
  if (NULL == heap_struct)  /* NULL pointer? */
    return;                 /* if so, do nothing */

  free(heap_struct->heap_nums);  /* first free encapsulated
                                    aggregate */
  free(heap_struct);             /* then free containing
                                    structure */
}
```

After checking that the argument heap_struct is not NULL, the function first frees the heap_nums array, which requires that the heap_struct pointer is still valid. It would be an error to free the heap_struct first. Once the heap_nums have been deallocated, the heap_struct can be freed as well. If heap_struct were freed but heap_nums were not, then the float elements in the array would be leakage: still allocated bytes but with no possibility of access—hence, of deallocation. The leakage would persist until the nestedHeap program exited and the system reclaimed the leaked bytes.

A few cautionary notes on the free library function are in order. Recall the earlier sample calls:

```
free(heap_struct->heap_nums);  /* first free encapsulated
                                  aggregate */
free(heap_struct);             /* then free containing
                                  structure */
```

These calls free the allocated storage—but they do *not* set their arguments to NULL. (The free function gets a copy of an address as an argument; hence, changing the copy to NULL would leave the original unchanged.) For example, after a successful call to free, the pointer heap_struct still holds a heap address of some heap-allocated bytes, but using this address now would be an error because the call to free gives the system the right to reclaim and then reuse the allocated bytes.

Calling free with a NULL argument is pointless but harmless. Calling free repeatedly on a non-NULL address is an error with indeterminate results:

```
free(heap_struct);  /* 1st call: ok */
free(heap_struct);  /* 2nd call: ERROR */
```

3.12.1. Memory Leakage and Heap Fragmentation

As the previous code examples illustrate, the phrase memory leakage" refers to dynamically allocated heap storage that is no longer accessible. Here is a refresher code segment:

```
float* nums = malloc(sizeof(float) * 10); /* 10 floats */
nums[0] = 3.14f;                          /* and so on */
nums = malloc(sizeof(float) * 25);        /* 25 new floats */
```

Assume that the first malloc succeeds. The second malloc resets the nums pointer, either to NULL (allocation failure) or to the address of the first float among newly allocated 25. Heap storage for the initial ten float elements remains allocated but is now inaccessible because the nums pointer either points elsewhere or is NULL. The result is 40 bytes (sizeof(float) * 10) of leakage.

Before the second call to malloc, the initially allocated storage should be freed:

```
float* nums = malloc(sizeof(float) * 10); /* 10 floats */
nums[0] = 3.14f;                          /* and so on */
free(nums);                               /** good **/
nums = malloc(sizeof(float) * 25);        /* no leakage */
```

Even without leakage, the heap can fragment over time, which then requires system defragmentation. For example, suppose that the two biggest heap chunks are currently of sizes 200MB and 100MB. However, the two chunks are not contiguous, and process *P* needs to allocate 250MB of contiguous heap storage. Before the allocation can be made, the system must *defragment* the heap to provide 250MB contiguous bytes for *P*. Defragmentation is complicated and, therefore, time-consuming.

Memory leakage promotes fragmentation by creating allocated but inaccessible heap chunks. Freeing no-longer-needed heap storage is, therefore, one way that a programmer can help to reduce the need for defragmentation.

3.12.2. Tools to Diagnose Memory Leakage

Various tools are available for profiling memory efficiency and safety. My favorite is *valgrind* (`www.valgrind.org/`). The *leaky* program (see Listing 3-22) illustrates the problem and the *valgrind* solution.

Listing 3-22. The leaky program

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

int* get_ints(unsigned n) {
  int* ptr = malloc(n * sizeof(int));
  if (ptr != NULL) {
    unsigned i;
    for (i = 0; i < n; i++) ptr[i] = i + 1;
  }
  return ptr;
}
```

```
void print_ints(int* ptr, unsigned n) {
  unsigned i;
  for (i = 0; i < n; i++) printf("%3i\n", ptr[i]);
}

int main() {
  const unsigned n = 32;
  int* arr = get_ints(n);
  if (arr != NULL) print_ints(arr, n);

  /** heap storage not yet freed... **/
  return 0;
}
```

The function `main` calls `get_ints`, which tries to `malloc` 32 four-byte integers from the heap and then initializes the dynamic array if the `malloc` succeeds. On success, the `main` function then calls `print_ints`. There is no call to `free` to match the call to `malloc`; hence, memory leaks.

With the *valgrind* toolbox installed, the following command checks the *leaky* program for memory leaks:

```
% valgrind  --leak-check=full  ./leaky
```

In the following code segment, most of the output is shown. The number of the left, 207683, is the process identifier of the executing *leaky* program. The report provides details of where the leak occurs, in this case, from the call to `malloc` within the `get_ints` function that `main` calls.

```
==207683== HEAP SUMMARY:
==207683==   in use at exit: 128 bytes in 1 blocks
==207683==   total heap usage: 2 allocs, 1 frees, 1,152 bytes
allocated
==207683==
```

```
==207683== 128 bytes in 1 blocks are definitely lost in loss
record 1 of 1
==207683==   at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-
gnu/...-linux.so)
==207683==   by 0x109186: get_ints (in /home/marty/gc/leaky)
==207683==   by 0x109236: main (in /home/marty/gc/leaky)
==207683==
==207683== LEAK SUMMARY:
==207683==   definitely lost: 128 bytes in 1 blocks
==207683==   indirectly lost: 0 bytes in 0 blocks
==207683==   possibly lost: 0 bytes in 0 blocks
==207683==   still reachable: 0 bytes in 0 blocks
==207683==   suppressed: 0 bytes in 0 blocks
```

If function `main` is revised to include a call to `free` right after the one to `print_ints`, then *valgrind* gives the *leaky* program a clean bill of health:

```
==218462==  All heap blocks were freed -- no leaks are possible
```

3.13. What's Next?

C variables can be defined inside and outside of blocks, where a block is the by-now-familiar construct that begins with a left brace { and ends with a matching right brace }. Where a variable is defined determines, within options, where its value is stored, how long the variable persists, and where the variable is visible. Every variable has a *storage class* (with `auto` and `extern` as examples) that determines the variable's persistence and scope. Functions too have a storage class: either `extern` (the default) or `static`. The next chapter gets into the details of storage classes.

CHAPTER 4

Storage Classes

4.1. Overview

In C, a *storage class* determines where functions and variables are stored, how long variables persist, and where functions and variables can be made visible. For functions, the key issue is visibility (*scope*) because the lifetime of a function is the lifetime of the program that contains the function. For variables, both lifetime and scope are of interest to the programmer.

Storage classes also shed further light on the distinction between *declarations* and *definitions* in C. In large programs, with the constituent functions typically residing in different files, the distinction is especially important. Once again, code examples illustrate the basics and advanced features. The chapter ends with a discussion of type qualifier `volatile`, yet another aspect of C's close-to-the-metal personality.

4.2. Storage Class Basics

Here are two examples of where a storage class shows up in C code:

```
static int counter;                /* static is a storage-class
                                      specifier */
extern void main() { /* body */ }  /* extern is a storage-class
                                      specifier */
```


M. Kalin, *Modern C Up and Running*, https://doi.org/10.1007/978-1-4842-8676-0_4

C has four storage class specifiers: `extern`, `static`, `auto`, and `register`. It is rare for the last two to be used explicitly in modern C because the compiler, on its own, does what the specifiers call for. The first two specifiers, `extern` and `static`, remain relevant. A function can be either `extern` or `static` only; a variable can be any one of the four. A storage class also impacts the following:

- The *scope* or *visibility* of the storage. For example, C functions are `extern` by default, which means that they can be made visible to any other function in the same program. To be `extern` in C is to be potentially *global* in scope.
- The *lifetime* of the storage, which depends directly on where the storage is provided. The name *storage class* derives from the fact that different parts of the memory hierarchy are in play. For example, a local variable—that is, a variable defined *inside* a block—is `auto` by default. Storage for such a variable comes from the stack or a CPU register, and the variable's lifetime is the time span during which the containing block is active because some instruction within the block is still executing.

HOW DOES A VARIABLE DEFINITION DIFFER FROM A DECLARATION?

A *definition* implements, whereas a *declaration* describes. A *declared* function describes how the function is called and excludes the function's body; a *defined* function includes the body as well. For variables, the distinction matters only in the case of `extern` variables, where there is *one* definition but there can be more than one declaration. For variables of every other storage class, the definition and the declaration are effectively the same.

Here is a summary of the default storage classes for functions and variables:

- Functions are either `extern` or `static`, with `extern` as the default.
- Variables *defined* outside of all blocks are either `extern` or `static`, with `extern` as the default.
- Variables *defined* inside a block are either `auto`, or `register`, or `static`, with `auto` as the default.

In summary, neither `static` (functions or variables) nor `register` (variables only) is a default storage class. For a function or variable defined outside all blocks, `extern` is the default; for a variable defined inside a block, `auto` is the default.

On modern computers, C functions are stored in the *text* area of memory, and a function's lifetime is accordingly the lifetime of the program to which the function belongs. However, a `static` function or variable is not visible outside of its containing source file, whereas an `extern` function or variable can be made visible throughout a program—no matter the file that contains its definition.

In the case of variables, in particular large arrays, the storage classes `extern` and `static` raise issues of efficiency. If an array is `extern` or `static`, then the array's lifetime is the program's lifetime. In effect, the size of the array becomes part of the program's runtime memory footprint. It is best to keep arrays on the stack or the heap so that storage for the arrays persists only as needed.

4.3. The auto and register Storage Classes

The details of the auto and register specifiers can be clarified through a code example.

Listing 4-1. The auto and register specifiers

```
#include <stdio.h>
#include <stdlib.h> /* rand() */

int main() {
  /* i and n are visible from their declaration to the end
  of main */
  auto int i; /* auto is the default in any case */

  int n = 10; /* auto as well */

  for (i = 0; i < n; i++) {
    register int r = rand() % 100; /* if no register
                                      available, auto */
    printf("%i ", r);
  } /* r goes out of scope here */
  putchar('\n'); /* instead of the usual printf("\n") */
  return 0;
}
```

The *autoreg* program (see Listing 4-1) shows how the auto and register specifiers could be used. Recall that these specifiers are used for *variables* only, and only for variables declared *inside* a block. In this example, there are two blocks:

- The body of function `main` is the outer block, and `int` variables `i` and `n` are declared in this block. Each is visible from the point of its declaration until the end of the block, in this case the end of function `main`. In particular, local variables `i` and `n` are visible *inside* the `for` loop, a nested code block.
- The `for` loop's body is another block. Declared therein is the `register` variable `r`, which is visible only within the body of the `for` loop.

The declarations for variables `i` and `n` are equivalent, although only the one for variable `i` explicitly uses the `auto` specifier. Because `auto` is the default specifier for a variable declared inside a block, this specifier is almost never used—except for demonstration purposes, as in the *autoreg* program.

The `register` specifier, shown here in the declaration for variable `r`, also is rarely used in modern C, as clarified shortly. If the compiler cannot implement variable `r` with a CPU register, then the storage class reverts to the default, `auto`. The scope for `auto` and `register` variables is the same in any case: the containing block.

The `register` specifier has become outdated because an optimizing compiler tries to use a CPU register to store scalar values such as the ones stored in `r` during the `for` loop. It is more productive to flag the compiler for optimization (e.g., `gcc -O1...`) than to use the `register` specifier. The `auto` specifier also has become outdated because an optimizing compiler opts for CPU registers whenever possible and uses the stack as the fallback for scratchpad. From now on, the code examples dispense with explicit uses of `auto` and `register`.

4.4. The `static` Storage Class

The `static` specifier applies to both functions and variables. A variable can be declared as `static` either inside a block (with resulting *block scope*) or outside all blocks (with a scope from that point until the end of the file). The first code example deals with `static` variables.

DOES THE C COMPILER SUPPORT PROFILING?

Yes. The flag `-pg` enables profiling:

```
% gcc -pg profile.c   ## produces executable a.out (on
Windows: A.exe)
```

Running the program produces the file *gmon.out*, and the utility `gprof` then can be executed from the command line:

```
% gprof
```

A detailed profiling analysis is printed to the screen.

Although the C compiler includes support for profiling (see the sidebar), this code example shows how the `static` specifier can be used to keep track of how many times a particular function is invoked.

Listing 4-2. Using `static` variables to profile function calls

```
#include <stdio.h>

#define SizeF 109
#define SizeB 87

void foo() {
  static unsigned n = 0; /* initialized only once */
  if (SizeF == ++n) printf("foo: %i\n", n);
}
```

```
void bar() {
  static unsigned n; /* initialized automatically to zero */
  if (SizeB == ++n) printf("bar: %i\n", n);
}

void main() {
  unsigned i = 0, limit_foo = SizeF, limit_bar = SizeB;
  while (i++ < limit_foo) foo(); /* call foo() a bunch of
  times */

  i = 0;
  while (i++ < limit_bar) bar(); /* call bar() a bunch of
  times */
}
```

The *profile* program (see Listing 4-2) tracks the number of times that `main` calls two other functions, `foo` and `bar`. Each of the called functions has a local `static` variable named `n`. The compiler initializes a `static` variable to zero unless the program provides an initial value. Two points about these `static` variables are important in this example:

- Because each variable is declared *inside* a function, each variable has function scope only. Accordingly, the two distinct variables can have the same name, in this case `n`.
- Unlike an `auto` variable (stack based), a `static` variable (not stack based) maintains its state across function calls. For example, each time that the `foo` function is called, its variable `n` is incremented and retains this new value even when `foo` exits. An initialized `auto` variable would be reinitialized on every call to the function that encapsulates the variable.

A `static` variable has the lifetime of the program regardless of where the variable is declared, but its scope does differ depending on where the variable is declared. If declared inside a block, a `static` variable has block scope. If declared outside all blocks, a `static` variable has file scope: it is visible from the point of declaration until the end of the containing file.

To define a function as `static` is to restrict the function's scope to the file in which it resides. Functions are `extern` by default, which means that they are potentially visible throughout the compiled program, regardless of the source file that happens to contain them. Making a function `static` is as close as it comes to *private* in C: `static` functions might be described as *private to the file*. Scope is the *only* difference that matters between `extern` and `static` functions: the former can have program scope, whereas the latter can have file scope only.

4.5. The `extern` Storage Class

The source code for a large program is likely distributed among many files. A function housed in one file may need to call a function housed in another file. For example, a program that invokes a library function such as `printf` is thereby calling a function housed in another file—the library's delivery file. Furthermore, a program may require that the *same* variable—not just different variables with the same name—be accessible across files. But neither a `static` function nor a `static` variable can be made visible outside of its containing file. Such functions and variables have *program lifetime* due to their `static` character, but they have only *file scope* at most.

The `extern` storage class supports truly global scope, although the programmer needs to do some work to make this happen. The basic two steps for global scope go as follows:

- A variable or function is *defined*, implicitly or explicitly, as `extern` in *one* file. (A variable defined *outside* all blocks defaults to `extern`, and functions in general default to `extern`.) The term `extern` can but need not be used in the definition.
- This variable or function is then *declared* as `extern` in any other file that requires access.

The rule of thumb for making life easy on the programmer is to avoid the explicit `extern` in a *definition* (in particular for variables) and to use the explicit `extern` only in a declaration.

A code example should help to clarify the details. The example consists of two files, *prog2files1.c* and *prog2files2.c*. These will be considered in order.

Listing 4-3. One source file in the prog2files program

```
#include <stdio.h>

/* definition of the extern variable: keyword extern is absent,
but could be present if the variable were initialized in its
definition. */
int global_num = -999; /* would be initialized to 0
                       otherwise */

extern void doubleup(); /* declaration of a function defined in
                        another file */

extern void print() { /* extern could be dropped from this
                     definition */
  printf("global_num: %i\n", global_num);
}
```

```
/* set2zero can be invoked only by functions within
this file */
static void set2zero() {
  global_num = 0;
}

void main() { /* extern could be added, but not necessary */
  doubleup(); /* function in another file */
  doubleup(); /* call doubleup() again */
  print(); /* -3996 */

  set2zero(); /* function in this file */
  print(); /* 0 */
}
```

The *prog2files1.c* file (see Listing 4-3) does the following:

- Defines the `int` variable `global_num` outside all blocks. This makes the variable `extern`, although the specifier `extern` does not occur in the definition. The variable also is initialized to -999. Were the variable not initialized explicitly, the compiler would set its value to 0. There is subtle syntax at play here. If the specifier `extern` were used, then the variable would have to be initialized explicitly in order to distinguish its single *definition* from one of its many possible declarations. The safe approach is to omit the specifier `extern` from the definition and to use this specifier only in declarations. The second file in the *prog2files* program shows a declaration for `global_num` with the specifier `extern`.

- Declares the function `doubleup` as `extern`, thereby signaling that this function is *defined* elsewhere—in this case, in the other source file, *prog2files2.c*.
- Defines the function `print` using the specifier `extern`. The `extern` is not necessary because any defined function is `extern` by default unless explicitly specified to be `static`.
- Defines the function `main` as `extern`, but without using the specifier.

The `main` function, housed in the source file *prog2files1.c*, invokes the `doubleup` function twice—a function housed in the program's other source file, *prog2files2.c*. If the `doubleup` function were not declared in *prog2files1.c*, the compiler would complain. The `main` function also invokes the `static` function `set2zero`. Because `set2zero` is `static`, it must be invoked by a function such as `main` in the *same* source file, *prog2files1.c*.

Listing 4-4. The other source file in the prog2files program

```
/* declaration: keyword extern is required, and the variable
must not be initialized here because it then would be a
definition. */
extern int global_num;

void doubleup() {   /* definition: doubleup is declared
                       elsewhere, defined here */
  global_num *= 2;  /* the global_num defined elsewhere, but
                       accessed here */
}
```

The second source file *prog2files2.c* (see Listing 4-4) is deliberately simple. There are two points of interest:

- The variable `global_num` is *declared* with the specifier `extern` and not initialized. If the variable `global_nums` were initialized here, this would count as a *definition*, thereby breaking the rule that an `extern` variable (or function) must be defined exactly once in a program. The declaration for `global_num` occurs outside all blocks but could occur within the function `doubleup`. In any case, the declaration of `global_num` with the required specifier `extern` signals that this variable is *defined* elsewhere, which happens to be the other source file *prog2files1.c*.
- The function `doubleup` is defined here and is `extern` by default. This function is declared in the other source file with the specifier `extern`.

The source files in the *prog2files* program are compiled in the usual way:

```
% gcc -o prog2files prog2files1.c prog2files2.c   ## file names
could be in any order
```

For review, here again is the rule of thumb that sidesteps the legalese surrounding the specifier `extern`. This rule can be spelled out as two related recommendations:

- Never use the specifier `extern` in function or variable *definitions*, which must occur outside all blocks. The variables then can be initialized or not according to need.

- Use the specifier `extern` only in *declarations* of functions and variables. A variable cannot be initialized in a *declaration*, as this would transform the declaration into a definition.

WHAT DOES CONST MEAN IN C?

The qualifier `const` for *constant* originated in C++ and was brought into C. A few code segments clarify.

```
const int n = 17; /** n is constant or read-only **/
n = -999;         /** ERROR: won't compile -- n is read-only **/
```

There are workarounds through pointers, however.

```
int* ptr = &n; /* n is const */
*ptr = -999;   /** WARNING: bad idea, but works **/
```

The *const-ness* can be cast away from the pointer:

```
int* ptr = (int*) &n; /* (int*) cast is critical here, as &n is
(const int*) */
*ptr = -999;          /* no error, no warning */
```

Recall that the parameters to the `qsort` comparison function are `const void*`, in effect a promise that such pointers will not be used to modify the values pointed to.

4.6. The `volatile` Type Qualifier

A variable of any type, including pointers and `struct` types, can be qualified as `volatile`:

```
volatile int n; /* int could be left of volatile */
```

The `volatile` qualifier cautions the compiler against doing any optimization on a variable so qualified, in this case `n`. For example, there are situations in which an optimizing compiler should not implement a variable as a CPU register. Two sample situations are introduced in the following.

The first example deals with an interrupt service routine (ISR). As the name indicates, an ISR handles interrupts, which originate from *outside* the executing program. For example, imagine an ISR written in C to handle input from one of the machine's data ports, for example, the port for the keyboard. The programmer might define and initialize a variable `nextc` to store the next character read from the keyboard. An optimizing compiler, unaware that the data source for the variable is *outside* the executing program, may reason that `nextc` acts within the program like a *constant* best implemented in read-only storage; in other words, the compiler sees the initialization but does not see any updates to `nextc`. As a result, the compiler might deliver only this initial value to functions that read `nextc`. This optimization would undermine the ISR's task of reading arbitrary characters from the keyboard.

WHAT'S A MULTICORE MACHINE?

A *core* is a fabrication component that contains a *processing unit*: one or more CPUs (processors), registers, cache memory, and other architectural components. A multicore machine is therefore a *multiprocessor* machine, with one or more CPUs per core; hence, a multicore machine can support true parallelism.

The second example concerns multithreading, which Chapter 7 covers in detail. In a multithreaded program, multiple *threads of execution* (sequences of instructions) can communicate with one another through shared memory, for example, through a global variable *N* that is visible

across the threads because *N* is implemented as storage in main memory. On a multicore machine, however, the registers on a particular core would be visible only to a thread executing on the core's processor(s). The point deserves emphasis: if thread *T1* executes on core *C1*, then *T1* sees only the registers on *C1*. If the compiler were to implement global variable *N* as a register on core *C1*, then threads executing on some other core would not see *N*. In short, it is important that *N* be implemented in main memory if *N* is to be visible across the multiple threads in the process. The programmer could make this point to the compiler by qualifying global variable *N* as `volatile`, thereby recommending that the compiler not optimize by implementing *N* as a CPU register.

A program with no `volatile` qualifications may compile to the same executable as a version of the same program with many such qualifications. The `volatile` qualifier does not guarantee anything; instead, the qualifier is only a cautionary note that the programmer sends to an optimizing compiler.

Although the syntax for `volatile` is close to that for storage classes, `volatile` is *not* a storage class. The `volatile` qualifier has no connection whatsoever with how a variable, thus qualified, is stored.

DOES C COME WITH A DEBUGGER?

The standard compilers have a debugger with the usual support: breakpoints, stepping, viewing and resetting variables, and so on. Here is an example with the *fpoint.c* as the source file:

1. Compile with the `-g` flag:

```
% gcc -g -o fpoint point.c
```

2. Invoke the debugger on the compiled file:

```
% gdb fpoint
```

Inside the debugger, there is a help menu.

4.7. What's Next?

Every program in execution requires at least one processor (CPU) to execute its instructions and memory to store these instructions and the data that together make up the program. Except for special cases, a program uses I/O devices as well, which are accessible to a program as *files* of one sort or another. A *file* in this generic, abstract sense is just a collection of words, and a *word* is just a formatted collection of bits. For example, a camera in a smartphone and the lowly keyboard on a desktop machine are both files in this sense. The role of input and output operations is, of course, to allow a program to interact with the outside world. The next chapter gets into the details by highlighting C's flexible approach to input/output operations.

CHAPTER 5

Input and Output

5.1. Overview

Programs of all sorts regularly perform input/output (I/O) operations, and programmers soon learn the pitfalls of these operations: trying to open a nonexistent file, having too many files open at the same time, accidentally overwriting a file and thereby losing its data, and so on. Nonetheless, I/O operations remain at the core of programming.

C has two APIs for I/O operations: a low-level or *system-level* API, which is byte-oriented, and a high-level API, which deals with multibyte data types such as integers, floating-point types, and strings. The system-level functions are ideal for fine-grained control, and the high-level functions are there to hide the byte-level details. Although the two APIs can be mixed, as various code examples show, this must be done with caution. This chapter covers both APIs and examines options such as *nonblocking* and *nonsequential* for I/O operations.

Files and I/O operations are one way to support *interprocess communication* (IPC). Recall that separate processes have separate address spaces by default, which means that shared memory, although possible, requires setup for processes to communicate with one another. Local files, by contrast, can be used readily for IPC: one process can *produce* data that is streamed to a file, while another process can *consume* the data streamed from this file. A later section examines how to synchronize process access to shared files.

M. Kalin, *Modern C Up and Running*, https://doi.org/10.1007/978-1-4842-8676-0_5

The API for I/O operations extends to networking, in particular to *socket* connections between processes running on different machines. This chapter thus provides background for the next.

5.2. System-Level I/O

A short review of some basic concepts should be helpful in clarifying system-level I/O in C. A *process*, as a program in execution, requires shared system resources from at least two but typically from three categories:

- *Processors* to execute the program's instructions (at least one required)
- *Memory* to store the program's instructions and data (required)
- *Input/output devices* to connect to the outside world (optional but usual)

Some special-purpose utility processes (*background processes*) may require access to few, if any, I/O devices. For convenience, a *normal process* is one that uses resources from all three categories. When a normal process starts, the operating system automatically gives the process access to three files, where a *file* is a collection of words and a *word* is a formatted collection of bits (e.g., bits that represent printable characters such as *A* and *Z* in a character-encoding scheme such as ASCII). These three files have traditional names, and they are associated by default with particular I/O devices:

- The *standard input* defaults to the keyboard but can be *redirected* to some other device (e.g., a network connection).
- The *standard output* defaults to the screen but can be *redirected* to some other device (e.g., a printer).

- The *standard error* defaults to the screen but can be *redirected* to some other device (e.g., a log file on the local disk).

At the command line on modern systems, the less-than sign < redirects the standard input; the greater-than sign > redirects the standard output; and the combined symbols *2*> redirect the standard error. Examples are forthcoming, together with a clarification of why the numeral in *2*> is 2.

In system-level I/O, nonnegative integer values called *file descriptors* are used to identify, within a process, the files that the process has opened. Recall that files can be used for interprocess communication (IPC). If two processes were to open a file to share data using system-level I/O, then each process would have a *file descriptor* identifying the file; the file descriptor values would not have to be the same because the operating system maintains a global *file table* that tracks which processes have opened which files.

Table 5-1. *File descriptor and* `FILE*` *overview*

Name	File descriptor	Macro	`FILE*`
standard input	0	`STDIN_FILENO`	`stdin`
standard output	1	`STDOUT_FILENO`	`stdout`
standard error	2	`STDERR_FILENO`	`stderr`

Table 5-1 summarizes the basics about the three files to which a normal process automatically gets access. For other files, access is achieved through a successful call to an *open* function: in low-level I/O, the basic function is named `open`, and in high-level I/O, the basic function is named `fopen`. The table now can be clarified further:

- In system-level I/O, a program can use the three reserved file descriptors (0, 1, and 2) for I/O operations. A short example follows. The integer values themselves can be used, or the macros (defined in *unistd.h*) shown in the third column.
- In high-level I/O, the header file *stdio.h* includes three pointers to a `FILE` structure, which contains pertinent information about an opened file. The pointer `stdin` is the high-level counterpart of file descriptor 0, `stdout` is the high-level counterpart of file descriptor 1, and `stderr` is the high-level counterpart of file descriptor 2.

A first code example draws these introductory points together.

Listing 5-1. Some basic I/O operations using the system-level API

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

#define BuffSize 4

void main() {
  const char* prompt = "Four characters, please: ";
  char buffer[BuffSize]; /* 4-byte buffer */

  /* write returns -1 on error, count of bytes written on
  success */
  write(STDOUT_FILENO, prompt, strlen(prompt));
  ssize_t flag = read(0, buffer, sizeof(buffer)); /* 0 ==
  stdin */

if (flag < 0)
  perror("Ooops...");   /* this string + a system msg
                           explaining errno */
```

```
else
  write(1, buffer, sizeof(buffer));
  /* 1 == stdout */
putchar('\n');
}
```

The *ioLL* program (see Listing 5-1) is a first look at low-level or byte-oriented I/O. The program uses two of the three automatically supplied *file descriptors*: 0 for the standard input (keyboard) and 1 for the standard output (screen). The key features of the program can be summarized as follows:

- The program writes a prompt, implemented as a string literal, to the standard output. The `write` function takes three arguments:
  - The first argument specifies the *destination* for the `write`, in this case the standard output. The *file descriptor* value 1 could be used here instead of the macro `STDOUT_FILENO`.
  - The second argument is the *source* of the bytes, in this case the address of the first character `F` in the prompt string.
  - The third argument is the number of bytes to be written, in this case the value of `strlen(prompt)`. The characters are, by default, encoded in ASCII; hence, `strlen` effectively returns the number of bytes to be written.

The `read` function likewise expects three arguments:

- The first argument specifies the *source* from which the bytes are read, in this case the standard input (0), the keyboard by default.

- The second argument specifies where the bytes should be stored, in this case the `char` (byte) array named `buffer`.
- The third argument specifies the number of bytes to be read into the buffer, in this case four.

Like many of the low-level I/O functions, `read` returns an `int` value: the number of bytes read, on success, and -1, on error. If an error occurs, an error code is available in the global variable `errno`, which is declared in the header file *errno.h*. The `perror` function prints a human-readable description of this error. This function takes a single string argument so that the user can add a customized error message to which `perror` appends a system error message. If only the system error message is of interest, `perror` can be called with `NULL` as its argument.

The program concludes with another call to `write`, this time using 1 to designate the standard output. The bytes to be written come from the array `buffer`, and the number of bytes is computed as `sizeof(buffer)`, which returns the number of bytes *in the array*, not the size of the pointer constant `buffer`.

The buffer does *not* include extra space for a null terminator: the program does not treat the input from keyboard as a *string*, but rather as four independent bytes. The `write` function takes the same approach: no string terminator is needed because the last argument to `write` specifies exactly how many bytes should be written, in this case four.

A short experiment underscores the level at which the functions `read` and `write` work. The experiment is to replace

```
char buffer[BuffSize];
```

with

```
int buffer; /* sizeof(int) is 4 */
```

or, indeed, with a variable of any data type whose size is at least 4 bytes. The `read` call now changes to

```
ssize_t flag = read(0, &buffer, sizeof(buffer));
                              /* &buffer == address of buffer */
```

The 4 bytes are to be put into a single `int` variable, which now acts like a 4-byte buffer. The `write` statement requires only a minor but critical change:

```
write(1, &buffer, sizeof(buffer));  /* need buffer's address */
```

The address operator must be applied to `buffer`, which is now just a scalar `int` variable.

This experiment underscores that system-level I/O does not honor multibyte types. For example, the bytes read into the `int` variable `buffer` could be any characters whatsoever. Here is a screen capture of a sample run of the revised *rwLL* program:

```
% ./ioLL
Four characters, please: !$ef
!$ef
```

These characters are not numerals, of course. The low-level `read` and `write` functions treat these simply as 8-bit bytes stored together in a 4-byte variable named `buffer`.

5.2.1. Low-Level Opening and Closing

The next two code examples introduce the byte-oriented `open` and `close` functions. The *sysWrite* program writes an array of `int` values, 4 bytes apiece, to a disk file, and the *sysRead* program reads the bytes from the same file in two different ways. The file descriptors 0 (standard input), 1 (standard output), and 2 (standard error) identify files that are opened automatically when a process begins execution; hence, there is no need for

the program to call open on these three. For other files, however, a call to open is required, and a matching call to close is sound practice. (When a program terminates, the system closes any files that the program may have opened.) The open function, like so many in the standard libraries, takes a variable number of arguments.

Listing 5-2. Writing to a local file with system-level I/O

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

#define FILE_NAME "nums.dat"

void main() {
  /* Open a file for reading and writing. */
  int fd = open(FILE_NAME,                    /* name */
                O_CREAT | O_RDWR,             /* create, read/
                                                 write */
                S_IRUSR | S_IWUSR | S_IXUSR | /* owner's
                                                 rights */
                S_IROTH | S_IWOTH | S_IXOTH); /* others'
                                                 rights */

  if (fd < 0) { /* -1 on error, positive value on success */
    perror(NULL);
    return;
  }

  /* Write some data. */
  int nums[ ] = {9, 7, 5, 3, 1}; /* int[ ] type */
  ssize_t flag = write(fd, nums, sizeof(nums));
  if (flag < 0) { /* -1 on error, count of written bytes on
                     success */
```

```
    perror(NULL);
    return;
  }

  /* Close the file. */
  flag = close(fd);
  if (flag < 0) perror(NULL);
}
```

The *sysWrite* program (see Listing 5-2) tries to open a file on the local disk, creating this file if necessary. The program sets the access rights for the file's owner and for others. The program then writes five integers to the file and closes the file. There is error-checking on all three of these I/O operations.

In this example, the call to the open function has three arguments, but the open function also can be called with only the first two arguments. The arguments in this case are as follows:

- The first argument is the name of the file to open. In this case, the full path is not used; hence, the file will be created in the directory from which the *sysWrite* program is launched.
- The second argument consists of flags, perhaps bitwise *or-ed* together as in this case. The pair

  ```
  O_CREAT | O_RDWR
  ```

  signals that the file should be created, if necessary, and opened for both *read* and *write* operations.
- The third argument consists of bitwise *or-ed* values that specify access permissions on the file. In this example, the file's *owner* has *read/write/execute* permissions, as do *others*. In a production environment, the access permissions of *owner* and *others* might differ.

If the call to open succeeds, a file descriptor is returned. Its value is the *smallest* positive value not currently in use by the process as a file descriptor. Since the file descriptor for the standard error (2) is in use, the smallest available value in this case would be 3. A print statement could be added to confirm that the value of fd is, indeed, 3.

If the call to open fails, -1 is returned to signal some error or other. (The next code example shows a sample perror message.) The call to write again has the three required arguments: the *destination* for the written bytes, the *source* of these bytes, and the *number* of bytes to write. Here is the relevant code segment:

```
int nums[ ] = {9, 7, 5, 3, 1}; /* int[ ] type */
ssize_t flag = write(fd, nums, sizeof(nums));
                         /* ssize_t is a signed integer type */
```

No looping is needed to write the array's contents because the third argument, sizeof(nums), is the number of bytes in the array as a whole. In this example, the bytes are written as integer values because the array's elements are stored in memory as int instances. In short, the target file *nums.dat* contains binary data, not text. Checking the size of the file *nums.dat* confirms that it holds 20 bytes, 4 bytes apiece for the 5 integers written to this file.

The *sysWrite* program opens a file by specifying access rights for the file's owner and for others. In general, these rights are divided into three categories: owner, group, and other. The macros such as S_IRUSR and S_IWUSR are assigned values such that their bitwise *or-ing* yields unique values. For example:

```
S_IRUSR | S_IWUSR == 384 ## decimal
```

whereas

```
S_IRUSR | S_IRWXU == 448 ## decimal
```

The bitwise *or-ings* can be as complicated as needed. It is common in Unix-like systems to set file permissions from the command line with octal values that reflect the bitwise *or-ing* of the values shown. For example:

```
S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH == 0664 ## octal
```

Table 5-2. *Access permissions*

Octal code	Symbolic code	Meaning
0001	S_IXOTH	Others can execute.
0002	S_IWOTH	Others can write.
0004	S_IROTH	Others can read.
0007	S_IRWXO	Others can do anything.
0010	S_IXGRP	Group can execute.
0020	S_IWGRP	Group can write.
0040	S_IRGRP	Group can read.
0070	S_IRWXG	Group can do anything.
0100	S_IXUSR	Owner can execute.
0200	S_IWUSR	Owner can write.
0400	S_IRUSR	Owner can read.
0700	S_IRWXU	Owner can do anything.

Table 5-2 summarizes the access permissions on files. In the left column, the values are octal. In C programs, an integer constant that starts with a `0` is interpreted as being in base-8, just as one starting with `0x` or `0X` is interpreted as being in base-16. It is common to use the octal values in command-line utilities such as *chmod*, but the symbolic constants are the way to go in programs. Note, by the way, that the permission values are such that any bitwise *or-ing* still yields a unique value. Also, mistakes such as

```
S_IWUSR | S_IXGRP | S_IWUSR /* S_IWUSR occurs twice */
```

are harmless.

Listing 5-3. Reading from a local file with system-level I/O

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>

#define FILE_NAME "nums.dat"

void main() {
  int fd = open(FILE_NAME, O_RDONLY); /* open for
  reading only */
  if (fd < 0) { /* -1 on error, > 2 on success */
    perror(NULL); /* "No such file or directory" if nums.dat
    doesn't exist */
    return;
  }

  int read_in[5]; /* buffer to hold the bytes */
  ssize_t how_many = read(fd, read_in, sizeof(read_in));
  if (how_many < 0) {
    perror(NULL);
    return;
  }

  close(fd); /* no error check this time */

  int i;
  int n = how_many / sizeof(int); /* from byte count to number
                                     of ints */
  for (i = 0; i < n; i++) printf("%i\n", read_in[i] * 10);
                                     /* 90 70 50 30 10 */
}
```

The *sysRead* program (see Listing 5-3) reads five 4-byte `int` values from the same file that the *sysWrite* program populates with these integers. In the *sysRead* program, the file is opened for *read-only*. The available macro flags for a call to `open`, together with their values, are

```
#define O_RDONLY  0x0000  /* open for reading only */
#define O_WRONLY  0x0001  /* open for writing only */
#define O_RDWR    0x0002  /* open for reading and writing */
```

The source code documentation shows the `perror` message if the file *nums.dat* does not exist.

Once the file is opened, the `read` function requires a buffer in which to place the bytes, in this case the `read_in` array that can hold five `int` elements, or 20 bytes in all. The `read` function, like the others seen so far, returns -1 in case of error; 0 on end of file; and otherwise the number of bytes read.

A *read* operation is the inverse of a *write* operation, and the arguments passed to `read` and `write` reflect this relationship. The first argument to `read` is a file descriptor for the *source* of bytes, whereas this argument specifies the *destination* in the case of `write`. The second argument to `read` is the *destination* buffer, whereas this argument specifies the *source* in a `write`. The last argument is the same in both: the number of bytes involved.

The *sysRead* program uses the high-level `printf` function to print the `int` values. Each value is multiplied by 10 to confirm that `int` instances have been read into memory from the source file. Recall that a successful `read` returns the number of bytes, in this case stored in the local variable `how_many`; hence, `how_many` is divided by `sizeof(int)` to get the number of 4-byte integers, in this case five.

Together the *sysWrite* and *sysRead* programs illustrate how local disk files can support basic interprocess communication. The programs would

need to be amended so that, for example, the *sysRead* program would wait for the *nums.dat* file to be created and populated with integer values before trying to read from that file. A later code example covers file locking for synchronizing access to shared files.

5.3. Redirecting the Standard Input, Standard Output, and Standard Error

Redirecting the standard input, the standard output, and the standard error with programs launched from the command line is straightforward. A simplified version of an earlier program illustrates. This approach brings the advantage of using one and the same program for reading and writing arbitrarily many files, but without editing and then recompiling the source code.

Listing 5-4. Redirecting I/O

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

#define BuffSize 8

void main() {
  char buffer[BuffSize]; /* 8-byte buffer */
  ssize_t flag = read(0, buffer, sizeof(buffer)); /* 0 ==
                                                     stdin */
  if (flag < 0) {
    perror("Ooops...");
    return;
  }

  char ws = '\t';
```

```
  write(1, buffer, sizeof(buffer));    /* 1 == stdout */
  write(1, &ws, 1);                    /* ditto */
  write(2, buffer, sizeof(buffer));    /* 2 == stderr */
  putchar('\n');
}
```

The *ioRedirect* program (see Listing 5-4) expects to read 8 bytes from the standard input and then echoes these bytes to the standard output and the standard error. If the bytes are ASCII character codes, the program is easy to follow. Here is a screen capture of a sample run; my comments start with ##:

```
% ./ioRedirect      ## on Windows, drop the ./
12345678            ## typed in from the keyboard, echoed on
                       the screen
12345678 12345678   ## 1st 8 to standard output, 2nd 8 to
                       standard error
```

The file *infile* contains a single line:

```
abcdefgh
```

To redirect the standard input to this file, the command is

```
% ./ioRedirect < infile ## < redirects the standard input
```

The output now is

```
abcdefgh abcdefgh
```

To redirect the standard output to the file *outfile*, the command is

```
% ./ioRedirect > outfile
```

The eight characters entered on the keyboard now appear once on the screen (default for the standard error) and once in the local disk file *outfile*. By the way, if *outfile* already exists, then the redirection purges this file and then repopulates it; hence, caution is in order.

Redirection to the standard error differs only slightly. Recall that 2 is the file descriptor for the standard error:

```
% ./ioRedirect 2> logfile
```

Redirections can be combined as needed, for example:

```
% ./ioRedirect < infile 2> logfile
```

Assuming that *infile* is the same as before, the contents of *logfile* are

```
abcdefgh  abcdefgh
```

5.4. Nonsequential I/O

The examples so far have dealt with *sequential* I/O: bytes are read in sequence and written in sequence. It is convenient at times, however, to have *random* or *nonsequential* access to a file's contents. A short code example illustrates the basic API.

Listing 5-5. Random or nonsequential file access

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>

#define FILE_NAME "test.dat"
```

```
void main() {
  const char* bytes = "abcdefghijklmnopqrstuvwxyz";
  int len = strlen(bytes);
  char buffer[len / 2];
  char big_N = 'N';

  /* Open the file and populate it with some bytes. */
  int fd = open(FILE_NAME,
                O_RDWR | O_CREAT,             /* flags */
                S_IRUSR | S_IWUSR | S_IXUSR); /* owner's
                                                 rights */
  write(fd, bytes, len);

  off_t offset = len / 2;          /* twelve bytes in is
                                      character n */
  lseek(fd, offset, SEEK_SET);     /* SEEK_SET is the start of
                                      the file */
  write(fd, &big_N, sizeof(char)); /* overwrite 'n' with 'N' */
  close(fd);

  fd = open(FILE_NAME, O_RDONLY);
  lseek(fd, offset, SEEK_SET);
  read(fd, buffer, len / 2);
  close(fd);
  write(1, buffer, len / 2); /* Nopqrstuvwxyz */
  putchar('\n');
}
```

The *nonseq* program (see Listing 5-5) skips the error checking to minimize the clutter, thereby keeping the focus on the nonsequential file access. The program first writes 26 bytes (the lowercase characters in the English alphabet) to a file. After closing the file, the program reopens the file to do an `lseek` operation that sets up another `write` operation, this

time a `write` of just one byte. As the name indicates, the function `lseek` performs a *seeking* operation, which can change the current file-position marker. A closer look at `lseek` clarifies.

The library function `lseek` takes three arguments. They are, in order:

- A file descriptor
- A byte offset from a designated position in the file
- The start position for the offset, with three convenient macros to define the usual positions:
  - `SEEK_SET` is the *start* position in the file.
  - `SEEK_CUR` is the *current* position in the file.
  - `SEEK_END` is the *end* position in the file.

The `lseek` function returns -1 in case of an error, or the offset to indicate success. The returned offset could be saved for later use. The offsets for `lseek` are like indexes in a `char` array: an offset of 0 is the position of the first byte in the file from the *seek position*, and an offset of 1 is the position of the second byte in the file from the *seek position*, and so on. In this example, the offset is 13, the position of the ASCII character code for lowercase *n*. An `lseek` operation beyond the current end of a file does not expand the file's size; a subsequent *write* operation would be required to do so.

Once the current position has been reset with `lseek`, the program overwrites the lowercase *n* with an uppercase *N*. The file then is closed again only to be reopened one more time. There is another `lseek` to the position of the now uppercase *N* and a `read` operation to get the bytes for *N* through *z* into the `char` array named `buffer`. For confirmation, `buffer` is printed to the standard output.

5.5. High-Level I/O

System-level I/O is low level because it works with bytes, the `char` type in C; by contrast, high-level I/O can work with multibyte data types such as integers, floating-point numbers, and strings. To take but one convenient example, the API for the high-level I/O makes it straightforward to convert between, for example, integers and strings. High-level I/O can work at the byte (`char`) level, but this kind of I/O is especially useful above the byte level.

The names are similar for some functions in the high-level and the low-level API. For example, there is a low-level `open` function and the high-level `fopen` function, as well as the low-level `close` and the high-level `fclose` functions. There is an `fread` function in the high-level API that matches up with the `read` function in the low-level API. The functions differ in syntax, of course, but also in how they work at the byte level. The low-level functions work only at the byte level, whereas the high-level API can work directly with multibyte types such as `int` and `double`.

There is crossover. For example, the high-level `fdopen` function takes a low-level file descriptor as an argument but returns the high-level type `FILE*`, the return type for various high-level library functions. Consider this contrast for opening and closing a file on the local disk:

```
int fd = open("input.dat", O_RDONLY);  /* low-level: -1 on
                                          failure */
FILE* fptr = fopen("input.dat", "r");  /* high-level: NULL on
                                          failure */
```

The corresponding function calls to close the opened file would be

```
close(fd);    /* fd is an int value */
fclose(fptr); /* fptr is a FILE* value */
```

In general, a file opened with the low-level `open` function is closed with the low-level `close` function. In a similar fashion, a file opened with `fopen` is closed with the `fclose` function. By the way, there is a limit on how many files a process can have open at a time; hence, it is critical to close files when keeping them open is no longer important.

In the low-level API, the integer values 0, 1, and 2 identify the *standard input*, the *standard output*, and the *standard error*, respectively. In the high-level API, the `FILE*` pointers `stdin`, `stdout`, and `stderr` do the same. The data type of interest in high-level I/O is `FILE*`, not `FILE`. It would be highly unusual for a program to declare a variable of type `FILE`, but typical for a program to assign the value returned from a high-level I/O function to a variable of type `FILE*`.

The following code segment summarizes the contrast between low-level and high-level I/O, with variable `fd` as a file descriptor and variable `fptr` as a pointer to `FILE`:

```
int buffer[5];                          /* 5 ints == 20 bytes */
read(fd, buffer, sizeof(int) * 5);      /* byte level read: read
                                           20 bytes */
fread(buffer, sizeof(int), 5, fptr);    /* int level read: read
                                           5 ints */
```

The low-level `read` function reads a specified number of bytes and stores them somewhere—in this case, in a 20-byte buffer that happens to be an `int` array of size five. By contrast, the high-level `fread` function can read *multibyte chunks*, in this case five `int` values, which are 4 bytes apiece.

Some in the C community believe that `FILE` should have been named `STREAM`, and it is common to describe *high-level I/O* as *stream-based I/O*. In a technical sense, C has two ways for a program to connect to any file, including the standard input, a local disk file, and so on:

- Through a *file descriptor*, an integer value that identifies the opened file.
- Through a *stream*, a channel that connects a *source* and a *destination*: the file could be either the source (*read* operation) or destination (*write* operation).

To study the API for the high-level I/O is, in effect, to study various ways of managing I/O streams. The forthcoming examples do so.

Listing 5-6. Basics of high-level I/O

```
#include <stdio.h>
#define FILE_NAME "data.in"

void main() {
  float num;
  printf("A floating-point value, please: ");
  int how_many_floats = fscanf(stdin, "%f", &num);
                              /* last arg must be an address */
  if (how_many_floats < 1)
    fprintf(stderr, "Bad scan -- probably bad characters\n");
  else
    fprintf(stdout, "%f times 2.1 is %f\n", num, num * 2.1);

  FILE* fptr = fopen(FILE_NAME, "w");  /* write only */
  if (!fptr) perror("Error on fopen"); /* fptr is NULL (0) if
                                          fopen fails */
  int i;
  for (i = 0; i < 5; i++)
    fprintf(fptr, "%i\n", i + 1);
  fclose(fptr);

  fptr = fopen(FILE_NAME, "r");
  int n;
```

```
  puts("\nScanning from the input file:");
  while (fscanf(fptr, "%i", &n) != EOF)  /* EOF == -1 == all 1s
                                            in binary */
    printf("%i\n", n);
  fclose(fptr);
}
```

The *scanPrint* program (see Listing 5-6) covers some basics of high-level I/O, beginning with scanning a file for input. The statement

```
int how_many_floats = fscanf(stdin, "%f", &num);
```

highlights some distinctive features of the high-level API. The function `fscanf`, with `f` for *file*, is structured as follows:

- The first argument specifies the *source* from which to scan for input, in this case `stdin`. The shortcut function `scanf` is hard-wired to read from the standard input, but `fscanf` explicitly names the source as its first argument. The first argument to `scanf` is the second argument to `fscanf`, the format string:

  ```
  int how_many_floats = scanf("%f", &num); /* scanf
  instead of fscanf */
  ```

- The second argument to `fscanf` is the *format string*, which specifies how scanned bytes are to be converted into an instance of some type, including a multibyte type such as the 4-byte `float`. The format string can contain arbitrarily many formatters.

- The third argument is the destination *address*, that is, the address of where the formatted bytes are to be stored. In this example, the third argument is `&num`. The scanning functions in general, including `fscanf`, return

the number of *properly formatted* instances of the specified data type, in this case `float`. The format string requests that only a single `float` be formatted; hence, the returned value is either 0 (failure) or 1 (success).

WHY IS THE ADDRESS OPERATOR & SO CRITICAL IN THE *SCANNING* FUNCTIONS?

A typical call to `scanf` is

```
int num;           /* num is a local variable, and so contains
                   random bits */
scanf("%i", &num); /* read an int, store it at the address of n */
```

If the address operator & were missing from &`num` in the `scanf` call, the contents of `num` would be interpreted as an *address*, and it is highly unlikely that these random bits make up an address within the executing program's address space. If `num` is a local variable, for example, its contents are random bits from the stack or a register.

The *scanPrint* program prompts the user to enter a floating-point value. If inappropriate characters such as *abc.de* are entered instead, the program prints an error message to that effect. The `fprintf` function is used to print to the standard error:

```
if (how_many_floats < 1)
  fprintf(stderr, "Bad scan -- probably bad characters\n");
```

Otherwise, the scanned `float` value is multiplied to confirm that the conversion from bytes to a `float` instance indeed succeeded. The `printf` function is hard-wired for printing to the standard output, just as the `scanf` function is hard-wired for scanning from the standard input. In general, error messages should have the standard error as their destination; hence, the *scanPrint* function uses `fprintf` with `stderr` as the first argument.

The last loop in the program is a `while` loop, and the loop's condition is a common one in programs that use high-level I/O to read from files:

```
while (fscanf(fptr, "%i", &n) != EOF) /* EOF == -1 == all 1s in
binary */
```

The value returned from `fscanf` in particular, and the related scanning functions in general, is tricky:

- If `fscanf` is successful in reading and converting, it returns the number of such successes. This number could be zero, which does *not* represent an input error, but rather a *data conversion* failure.
- If an *end-of-stream* condition occurs before a successful scan-and-convert, the function returns -1 (the value of the macro `EOF`). The high-level API also includes the function `feof()`, which returns *true* (nonzero) to signal end of file and *false* (zero) otherwise.
- If an *input* error occurs (e.g., the data source is absent), `fscanf` also returns -1.

At issue, then, is how to distinguish between `EOF`, a normal eventuality when reading from a stream, and an outright error. The library function `ferror` returns nonzero (*true*) to indicate an error condition in the stream, and the global variable `errno` contains an error code under the same condition; as usual, the `perror` function can be used to print a corresponding error message. For the programmer, however, the difference may not matter: `fscanf` returns a negative value to signal, in effect, that a scan-and-convert operation on a stream has failed. The `ferror` function and the `errno` variable then can be used, if needed, to get more information on why the failure occurred.

A final point about EOF is in order. The EOF value (32 1s in binary) marks the end of a *stream*, and streams can differ in their sources. If the source is a file on a local disk, then the EOF is generated when a *read* operation tries to read beyond the last byte stored in the file. If the source is a *pipe*, a one-way channel between two processes, then the EOF is generated when the pipe is closed on the sending side. An EOF thus should be treated as a *condition*, rather than as just another data item. To be sure, a program recognizes the EOF condition by reading the 32 bits that make up the EOF value; but these 32 bits differ in meaning from whatever else happens to be read from the stream.

High-level I/O is appropriately named, for this level focuses on the multibyte data types that are dominant in high-level programming languages. There may be times at which any program must drop down to the byte level, but the usual level is awash with integers, strings, floating-point values, and other instances of multibyte types. C works well at either I/O level. Other technical aspects of high-level I/O will be explored in forthcoming examples, which provide context for exploring this API.

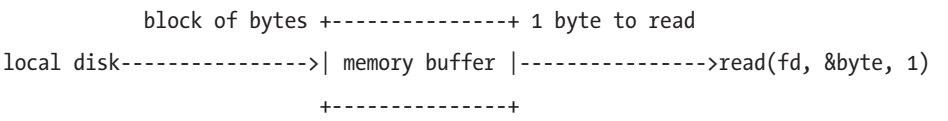
5.6. Unbuffered and Buffered I/O

There is yet another way to contrast low-level and high-level I/O: low-level I/O operations are said to be *unbuffered*, whereas the high-level ones are said to be *buffered*. It is important, however, to consider carefully what it means for low-level I/O to be *unbuffered*. A *buffer* in this context is a system-supplied, in-memory storage area between the executing program, on the one side, and the data source, on the other side.

Consider a code segment that reads a single byte:

```
char byte;
read(fd, &byte, 1); /* fd identifies a local disk file */
```

For reasons of efficiency, no modern operating system would fetch a single byte from disk into memory. Instead, the system would fetch a *block* of bytes into a memory buffer and then deliver the single byte from this buffer to the program:



To call low-level I/O *unbuffered* is *not* to deny system buffering under the hood. Instead, the point is that the low-level *API* supports the reading of just one byte, regardless of exactly how that byte might have been delivered to the program that invokes the `read` function with a third argument of 1.

The high-level `fread` function is essentially a wrapper around the low-level `read` function. Each can read a single byte:

```
char byte;
read(0, &byte, 1);          /* one byte from standard input */
fread(&byte, 1, 1, stdin);  /* ditto */
```

There are also high-level functions such as `fgetc` that *seem* to read a single byte, as the `c` for `char` in the function's name suggests. But the return type for `fgetc` and related high-level functions is `int`, not `char`. The `fgetc` function, like its high-level cousins, returns `EOF` to signal the end-of-stream condition, and `EOF` is a 4-byte `int` value. In situations other than `EOF`, the `fgetc` function returns a byte packaged in an `int` whose high-order 24 bits are zeroed out; the byte of interest occupies the low-order 8 bits.

Listing 5-7. A program contrasting `read` and `fgetc`

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

void main() {
  int i = 0, n = 8;
  char byte;

  /* unbuffered */
  while (i++ < n) {
    read(0, &byte, 1);   /* read a single byte */
    write(1, &byte, 1);  /* write it */
  }

  /* buffered */
  i = 0;
  while (i++ < n) {
    int next = fgetc(stdin); /* char read in a 4-byte int */
    fputc(next, stdout);     /* char written as a 4-byte int */
  }
  putchar('\n');
}
/* stdin is: 12345678abcdefgh */
```

The *buffer* program (see Listing 5-7) contrasts byte-fetching in the low-level and the high-level APIs. The low-level `read` stores the byte in a `char` variable, and `sizeof(char)` is guaranteed to be 1 byte. By contrast, the high-level `fgetc` function returns a 4-byte `int`. From the command line, the program can be tested against the *in.dat* file, whose contents are shown in the comment at the bottom:

```
% buffer < in.dat
```

Otherwise, all 16 characters should be entered at once from the keyboard, and only then should the *Return* key be hit.

The traditional contrast between *buffered* and *unbuffered* I/O can be misleading, as emphasized in the previous discussion. It is more useful to focus on program requirements. If a program needs to work directly with bytes, then the low-level API is designed to do precisely this. If a program deals mostly with multibyte types but occasionally drops down to the byte level, then the high-level API, which includes wrappers such as `fread` for low-level functions, is the sensible alternative.

5.7. Nonblocking I/O

Nonblocking I/O has become a popular technique for boosting performance. For example, a production-grade web server is likely to include nonblocking I/O in the mix of acceleration techniques. The potential boost in performance is likewise a challenge to the programmer: nonblocking I/O is simply trickier to manage than its blocking counterpart.

As the name indicates, *nonblocking* I/O operations do not block—that is, wait—until a *read*, *write*, or other I/O operation completes. Consider this code segment in system-level I/O:

```
int n;                      /* 4 bytes */
read(fd, &n, sizeof(int)); /* blocking read operation */
printf("%i\n", n);          /* next statement after
                               blocking read */
```

The file descriptor `fd` might identify a local file on the disk but also a less reliable source of bytes such as a network connection. If the `read` operation in the second statement blocks, then the `printf` statement immediately thereafter does *not* execute until the `read` call returns, perhaps because of an error.

If the `read` call were nonblocking, the code segment would need a more complicated approach. A nonblocking call returns immediately, and there are now various possibilities to consider, including the following:

- The `read` call got all of the expected bytes, in this case four.
- The `read` call got only some of the expected bytes and perhaps none at all.
- The `read` call encountered an error or end-of-stream condition.

The program now needs logic to handle such cases. Consider the second case. If one call to a nonblocking `read` gets only some of the expected bytes, then these bytes need to be saved, and another `read` attempted to get the rest. Perhaps a loop becomes part of the *read* logic: loop until all of the expected bytes arrive or an error occurs. At the very least, it seems that the `printf` statement would need to occur inside an *if* test that checks whether enough bytes were received to go on with the `printf`.

IS NONBLOCKING I/O THE SAME AS ASYNCHRONOUS I/O?

The use of the terms *blocking/nonblocking* and *synchronous/asynchronous* varies enough to rule out a simple *yes* or *no* answer. My preference is for the *blocking/nonblocking* pair because they seem more intuitive. That said, code examples are the best way to clarify exactly what these terms mean in practice.

5.7.1. A Named Pipe for Nonblocking I/O

The next code example uses the nonblocking *read* operation as representative of nonblocking I/O operations in general. For the example to be realistic, it should have two features:

- The data consumed in a nonblocking *read* operation should arrive randomly; otherwise, the nonblocking reads might behave exactly as blocking *reads* would have.
- After an attempted nonblocking *read* operation, the program should have meaningful work to do before the next *read* operation: the appeal of nonblocking I/O is that it frees up a program to do something else besides just waiting for an I/O operation to complete.

Accordingly, the code example consists of two programs: one writes in a pseudorandom fashion to a *named pipe*, and the other reads from this pipe. A *pipe* is a connection between processes, and *one way* in that one end of the pipe is for *writing*, and the other is for *reading*. There are both unnamed (or anonymous) and named pipes, both of which are used widely across modern systems for interprocess communication. A later example covers unnamed pipes.

Unix-like systems, and Cygwin for Windows, have command-line utilities that make it easy to demonstrate named pipes. The steps are as follows:

1. Open *two* terminal windows so that two command-line prompts are available. The working directory should be the same for both command-line prompts.

2. In one of the terminal windows, enter these two commands (my comments start with ##):

```
% mkfifo tester  ## creates special file named tester,
                    which implements the pipe
% cat tester     ## type the pipe's contents to the
                    standard output
```

   To begin, nothing should appear in the window because nothing has been written yet to the named pipe.

3. In the second terminal window, enter the following command:

```
% cat > tester ## redirect keyboard input to the pipe
hello, world!  ## then hit Return key
bye, bye       ## ditto
<Control-C>    ## terminate session with a Control-C
```

   Whatever is typed into this terminal window is echoed in the other. Once Control-C is entered, the regular command-line prompt returns in both windows: the pipe has been closed.

4. For cleanup, remove the file that implements the named pipe:

```
% rm tester
```

As the name *mkfifo* suggests, a named pipe also is called a *fifo* for *first in, first out* (FIFO). A named pipe implements the FIFO discipline so that the pipe acts like a normal queue: the first byte into the pipe is the first byte out, and so on. There is also a library function named `mkfifo`, which is used in the next code example.

Listing 5-8. A named pipe writer

```
#include <fcntl.h>
#include <unistd.h>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>

#define MaxLoops 12000   /* outer loop */
#define ChunkSize 16     /* how many written at a time */
#define IntsPerChunk 4   /* four 4-byte ints per chunk */
#define MaxZs 250        /* max microseconds to sleep */

void main() {
  const char* pipeName = "./fifoChannel";
  mkfifo(pipeName, 0666);   /* read/write for user/group/
                               others */
  int fd = open(pipeName, O_CREAT | O_WRONLY);   /* open as
                                                    write-only */

  sleep(2); /* give user a chance to start the fifoReader */
  int i;
  for (i = 0; i < MaxLoops; i++) {    /* write MaxWrites
                                         times */
    int j;

    for (j = 0; j < ChunkSize; j++) { /* each time, write
                                         ChunkSize bytes */
      int k;
      int chunk[IntsPerChunk];
```

```
      for (k = 0; k < IntsPerChunk; k++)
        chunk[k] = rand();
      write(fd, chunk, sizeof(chunk));
    }
    usleep((rand() % MaxZs) + 1); /* pause a bit for realism */
  }
  close(fd);                    /* close pipe: generates an
                                   end-of-file */
  unlink(pipeName);             /* unlink from the
                                   implementing file */

  printf("%i ints sent to the pipe.\n", MaxLoops * ChunkSize *
  IntsPerChunk);
}
```

The *fifoWriter* program (see Listing 5-8) creates and then writes sporadically to the named pipe called *fifoChannel*. Two statements at the start do the setup:

```
mkfifo(pipeName, 0666);  /* read/write for user/group/others */
int fd = open(pipeName, O_CREAT | O_WRONLY);  /* open as
                                                 write-only */
```

The first statement calls the library function `mkfifo` with two arguments: the name of the implementing file and the access permissions in octal. The second statement invokes the by-now-familiar `open` function, specifying that the file underlying the named pipe be created if necessary; the *fifoWriter* is restricted to *write* operations because of the `O_WRONLY` flag.

The *fifoWriter* then pauses for two seconds to give the user a chance to start the other program, the *fifoReader*. The *fifoWriter* needs to start first because it creates and opens the named pipe; but the two-second pause is there only for convenience. The *fifoWriter* program then loops

MaxLoops times (currently 12,000), writing multibyte chunks rather than single bytes to the pipe. A chunk is an array of four 4-byte int values. After writing the bytes to the pipe, the program pauses a pseudorandom number of microseconds, thereby making the *write* operations somewhat unpredictable. In all, the *fifoWriter* writes 768,000 int values to the pipe.

The program does cleanup at the end. The file descriptor fd is used to close the pipe, which generates an end-of-file signal for the *reader* side. The call to the unlink function unlinks the *fifoWriter* program from the implementation file *fifoChannel*. When all of the processes connected to the pipe unlink, the system is free to remove the file. In the current example, there is only a single *writer* process to the pipe and a single *reader* process from the pipe; hence, only two unlink operations are required.

Listing 5-9. A named pipe reader

```
#include <fcntl.h>
#include <unistd.h>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>

unsigned is_prime(unsigned n) { /* not pretty, but efficient */
  if (n <= 3) return n > 1;
  if (0 == (n % 2) || 0 == (n % 3)) return 0;

  unsigned i;
  for (i = 5; (i * i) <= n; i += 6)
    if (0 == (n % i) || 0 == (n % (i + 2))) return 0;

  return 1;
}
```

```
void main() {
  const char* file = "./fifoChannel";
  int fd = open(file, O_RDONLY | O_NONBLOCK); /* non-
                                                blocking */
  if (fd < 0) return; /* no point in continuing */
  unsigned primes_count = 0, success = 0, failure = 0;

  while (1) {
    int next;
    int i;

    ssize_t count = read(fd, &next, sizeof(int));
    if (0 == count)
      break;                    /* end of stream */
    else if (count == sizeof(int)) { /* read a 4-byte int
                                      value */
      success++;
      if (is_prime(next)) primes_count++;
    }

    else                              /* includes errors, and <
                                         4 bytes read */
      failure++;
  }

  close(fd);     /* close pipe from read end */
  unlink(file);  /* unlink from the underlying file */
  printf("Success: %u\tPrimes: %u\tFailure: %u\n",
  success,   primes_count, failure);
}
```

The *fifoReader* program (see Listing 5-9) reads from the named pipe that the *fifoWriter* creates and then populates with chunks of `int` values. The program configures the pipe for nonblocking *read* operations with the `O_NONBLOCK` flag passed as an argument to the `open` function:

```
int fd = open(file, O_RDONLY | O_NONBLOCK); /* non-blocking */
```

The utility function `fcntl` also could be used to set the nonblocking status, as illustrated shortly. The program tries to read `int` values from the pipe:

```
ssize_t count = read(fd, &next, sizeof(int)); /* 4-byte int
                                                 values */
```

Recall that the *fifoWriter* writes an array of four `int` values at a time and does so sporadically. Because the `read` operation in the *fifoReader* is nonblocking, three cases are singled out for application logic:

- If the `read` function returns 0, this signals an *end-of-stream* condition in the named pipe: no further bytes are coming from the one and only *writer*, and so the *fifoReader* breaks out of its infinite loop.
- If the `read` function yields exactly 4 bytes, then the program checks whether the integer value is a prime; this check represents the *do something* step before attempting the next *read* operation.
- If the `read` function fails to read exactly 4 bytes, or detects an error condition of any kind, then the program records the failure. The *fifoReader* program does not distinguish between partial reads (e.g., 2 bytes instead of the expected 4) and miscellaneous but nonfatal errors.

The *fifoReader*, like the *fifoWriter*, cleans up by closing the named file and unlinking from the implementation file. The *fifoReader* generates a short report at the end. On a sample run, the output (formatted for readability) was

```
Success: 768,000 Primes: 37,682 Failure: 31,642,062
```

Recall that the thirty-one million or so failures cover partial reads (`read` returns less than `sizeof(int)`) and nonfatal errors. In the end, the *fifoReader* does manage to read all of the 768,000 4-byte integer values that the *fifoWriter* writes to the pipe; but the *fifoReader* has plenty of unsuccessful reads as well: the *fifoWriter* sleeps between *write* operations, which gives the *fifoReader* ample opportunity to attempt nonblocking *read* operations doomed to fail because no unread bytes remain in the channel. In short, the output from the *fifoReader* is not surprising.

The *fifoReader* program has a dismal record of successful reads: about 2% of its *read* operations succeed in getting desired 4-byte `int` values, and the remaining *read* operations fail. The next chapter introduces an *event-driven* approach to *read* operations. This new approach first checks a channel for available bytes before even attempting a *read* operation.

The *fifoReader* program uses a flag passed to the `open` function to set the nonblocking status. The standard libraries include an `fcntl` utility, declared in the header file *fcntl.h*, that can do the same. The `fcntl` function has many uses and a correspondingly long documentation.

Listing 5-10. A function to set the nonblocking feature

```
unsigned set_nonblock(int fd) {
  int flags = fcntl(fd, F_GETFL);          /* get the current
                                              flag values */
  if (-1 == flags) return 0;               /* on error, return
                                              false */
```

```
  flags |= O_NONBLOCK;                          /* add non-
                                                   blocking */
  return -1 != fcntl(fd, F_SETFL, flags); /* 1 == success, 0 ==
                                                   failure */
}
```

The *setNonBlock* example (see Listing 5-10) shows how a file descriptor can be used to change the status from *blocking* to *nonblocking*. The `set_nonblock` function takes a file descriptor as its only argument and returns either *true* (1) or *false* (0) to signal whether the attempt succeeded. The function first gets the flags currently set (e.g., `O_CREAT` and `O_RDONLY`); if an error occurs here, *false* is returned. Otherwise, the function adds the `O_NONBLOCK` flag and then uses the `fcntl` function for updating. If the update succeeds, `set_nonblock` returns *true*, and *false* otherwise.

5.8. What's Next?

Network programming centers on the socket API, where a *socket* is an endpoint in a point-to-point connection between two processes. If the processes are running on physically distinct *hosts* (machines), a *network socket* is in play. If the processes are running on the same host, a *domain socket* could be used instead. (Domain sockets are a popular way for large systems, such as database systems, to interact with clients.) The very same I/O API used to interact with disk files works with sockets as well. Sockets, unlike pipes, are bidirectional.

This chapter has focused on I/O operations on a single machine. The next chapter broadens the study to include I/O operations across machines, and the chapter also explores an event-driven alternative to the nonblocking I/O introduced in this chapter.

CHAPTER 6

Networking

6.1. Overview

Network programming brings challenges beyond the details of yet another API. Networks can be brittle, as connections go down for reasons that may be hard to determine. Performance can vary widely because of network load. Programs must be sufficiently robust to deal with such issues and to anticipate the many others that come with the territory. Debugging network applications is harder, in general, than debugging ones that involve only a single machine. Given the challenges of network programming, it is no surprise that library functions in its support can seem subtle, complicated, and even overwhelming. This chapter uses relatively short but realistic examples to illustrate the challenges and sound ways to address them. After a few more introductory points, the discussion moves to a series of code examples.

Table 6-1. *The basic protocol stack*

Acronym	Meaning	Comments
HTTP	Hyper Text Transport Protocol	Web servers and their clients
TCP	Transmission Control Protocol	Connection-oriented, reliable
UDP	User Datagram Protocol	Connectionless, best-try
IP	Internet Protocol	Addressing: symbolic and numeric

M. Kalin, *Modern C Up and Running*, https://doi.org/10.1007/978-1-4842-8676-0_6

Table 6-1 lists the protocols of interest in the forthcoming examples. The *protocol stack* shown in the table has IP at the bottom and HTTP at the top: IP handles network addressing, and HTTP manages conversations between web servers and their clients. The HTTP protocol sits atop TCP, which is *connection-oriented*: the protocol sets up a connection between the endpoints before any data are transmitted. This *connection-oriented* feature contrasts with the *best-try* character of UDP. Under UDP, a sender sends a *datagram* to a receiver, but the receiver does not acknowledge automatically the receipt of the transmitted packet. Further, there is no error sent to the sender if the datagram gets lost. TCP adds error reporting, acknowledgment, and other services to the underlying UDP layer. HTTP, in turn, specializes the features inherited from TCP. The *web socket* protocol so popular in interactive web-based applications is built on top of TCP as well and has less overhead than HTTP.

The socket API has settings that reflect the different protocol layers shown in Table 6-1. Each of the protocols supports some level of configuration, which is done through a mix of utility functions and flags. The socket API must be complicated, in short, because the underlying protocol stack is so.

The library functions exposed in the socket API have been fine-tuned, reworked, and even obsoleted over time. Again, this is to be expected: the protocols themselves have changed. For example, the IP protocol comes in versions such as IPv4 and IPv6. The move from IPv4 to IPv6 is a major one in that Internet addresses go from 32 to 128 bits. The code examples address this versioning issue.

6.2. A Web Client

The first code example, a web client, is divided into two source files for convenience. The file *web_client.c* contains the high-level application logic: connect to a web server, send a request, and print the response.

The file *get_connection.c* contains the low-level networking details such as determining the type of connection (UDP or TCP) to the server, the amount of time a *read* operation should wait on a response before timing out, and so on. (The next sidebar describes a *Makefile* for compiling the files into an executable.)

Listing 6-1. A basic web client

```
#include <unistd.h>  /* low-level I/O */
#include <string.h>
#include <stdio.h>
#include <stdlib.h>  /* exit */
#include <errno.h>

#define BuffSize   2048  /* bytes */

extern int get_connection(const char*, const char*);
                                        /* declaration */

void main() {
  const char* host = "www.google.com";    /* symbolic IP
                                          address */
  const char* port = "80";                /* standard port for
                                          HTTP connections */
  const char* request = "GET / HTTP/1.1\nHost: www.google.
  com\r\n\r\n";
  ssize_t count;
  char buffer[BuffSize];

  /* connect */
  int sock_fd = get_connection(host, port);
  if (sock_fd < 0) {
    fprintf(stderr, "Can't connect\n");
    exit(-1);
  }
```

```
  /* send request */
  if (write(sock_fd, request, strlen(request)) < 0) {
    fprintf(stderr, "Can't write request\n");
    exit(-1);
  }

  /* get and write response */
  unsigned read_count = 0, total_bytes = 0;
  memset(buffer, 0, BuffSize); /* clear the buffer for reading */

  while (1) {
    count = read(sock_fd, buffer, sizeof(buffer));

    if (EWOULDBLOCK == errno || 0 == count) break;
    /* EWOULDBLOCK on timeout */
    if (-1 == count) continue;  /* continue on non-fatal
    error */

    write(1, buffer, count);
    read_count++; total_bytes += count;
  }
  close(sock_fd);
  fprintf(stderr, "\n\n%u bytes read in %u separate reads.\n",
  total_bytes, read_count);
}
```

The file *web_client.c* (see Listing 6-1), one of the two source files in the *webclient* program, uses the familiar `read` and `write` functions to communicate with a web server, in this case a Google HTTP server. A socket, just like a file on the local disk, has a file descriptor as its identifier. In addition to the `read` and `write` functions, the socket API also has `send` and `recv` functions, which take four arguments instead of the three in `read` and `write`. The fourth argument, in both cases, allows for configuration through various flags.

The *webclient* program initializes two strings, `host` and `port`, which specify the symbolic IP address for the Google server and the port number: *www.google.com* and 80, respectively. The port number 80 is the default for HTTP connections, just as 443 is the default port for HTTPS connections. Instead of the symbolic IP address, the program could have used the IPv4 *dotted-decimal* address *216.58.192.132*, each of whose four parts is 8 bits in size. The IP address and port number are sent as arguments to the `get_connection` function, which returns either the file descriptor for a socket (*success*) or -1 (*failure*). In case of failure, the *webclient* program exits after an error message.

WHAT'S A MAKEFILE?

The *webclient* program consists of two source files. It can be tedious to compile multiple files into an executable. The *make* utility, available on most Unix-like systems and through Cygwin, automates the process. Here is a bare-bones Makefile (with *Makefile* as its name), which the *make* utility reads by default:

```
webclient: web_client.c get_connection.c
        gcc -o webclient web_client.c get_connection.c
```

The first line lists the target (*webclient*) and its dependencies, with a colon as the separator. The dependencies consist of the two source files in any order. The second line begins with a *single* tab *character*, not blanks. This line is the command to be executed, in this case a familiar *gcc* command. At the command line, invoke the *make* utility:

```
% make     ## reads Makefile, follows the instructions
```

Far more extravagant examples of Makefile are available on the Web.

The *webclient* program has a third string literal, which holds the request. In more readable form, the request is

```
GET / HTTP/1.1          ## start line: verb (GET), noun (URI /),
                           HTTP version (1.1)
Host: www.google.com    ## required header element in HTTP 1.1
```

The first line is the HTTP *start line*, consisting of the HTTP *method* (*verb*) named `GET`: a GET request is a *read* request, whereas a POST request is a *create* request (e.g., a POSTed order form is a request to create an order). After the start line come arbitrarily many *header elements*, or *headers* for short. These are key/value pairs, with a colon as the separator. Under HTTP 1.1, the *host* header, which specifies the device address to which the request is being sent, is required; but a half-dozen or so headers is typical. The headers section ends with two carriage-return/newline combinations. (Two newlines are likely to work.) A GET request has no HTTP *body*, and so is complete as shown. In the start line, the URI (Uniform Resource Identifier) is the single slash, which web servers typically interpret as the identifier for their *home page*. In effect, then, the start line and the *host* header make up a *read* request for Google's home page.

The `write` function, with the socket's descriptor as its first argument, is used to send the request to the server. As usual in network programming, there is a check for an error condition: the `write` could fail for any number of reasons; if it does so, there is no point in continuing. Next comes a loop to read the server's response. There are some subtle issues to consider, as a closer analysis of client's connection to the web server will indicate.

Web servers typically keep client connections *alive* so that repeated request/response pairs can use the original connection. The motive, of course, is efficiency. Also, a web server is likely to *chunk* its response, that is, break the requested document (in this case, Google's HTML home page) into parts, transmitting each of these separately. The *webclient*

program has a *read* buffer of about 2KB (*kilobytes*). On a sample run, the program printed out this report:

```
48431 bytes read in 34 separate reads.
```

The Google home page is a hefty 48K bytes, and these were fetched in 34 separate *read* operations. The chunks of data from the various *read* operations vary in size.

How much time should be allowed between responses from the server? This is a question without an obvious answer. Whatever the answer, the socket API supports a *timeout* on a blocking *read* operation, which is in use here. For review, here are the three critical lines in the `while` loop that reads the Google response:

```
count = read(sock_fd, buffer, sizeof(buffer));
if (EWOULDBLOCK == errno || 0 == count) break;   /* EWOULDBLOCK
                                                    without a
                                                    timeout */
if (-1 == count) continue;                 /* continue on non-fatal
                                           error */
```

If the blocking *read* operation times out, there is a signal with an aptly named error code `EWOULDBLOCK`, which says that the *read* operation would have continued to block except for the interrupting signal. If the blocking times out, the program assumes that no further response bytes are coming.

Recall that `read` returns 0 on an end-of-byte-stream condition. In this case too, there is a `break` out of the `while` loop. If any other nonfatal error should occur (the -1 test), then execution of the `while` loop continues: the `continue` statement moves control directly to the loop condition, in this case bypassing the `write` operation to the standard output.

6.2.1. Utility Functions for the Web Client

The utility functions for the *webclient* program are broken out into their own file. These functions handle the networking details such as the protocol to be used, the address information of the web server, and the amount of time the client should wait for bytes from the server.

Listing 6-2. Utility code for the web client

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int get_connection(const char* host, const char* port) {
  struct addrinfo hints, *result, *next;
  int sock_fd, flag;
  memset(&hints, 0, sizeof(struct addrinfo)); /* zero out the
  structure */
  hints.ai_family = AF_UNSPEC;       /* IPv4 or IPv6 */
  hints.ai_socktype = SOCK_STREAM;   /* connection-
                                        based, TCP */
  hints.ai_flags = 0;         /* various possibilities here */
  hints.ai_protocol = 0;      /* any protocol */

  if ((flag = getaddrinfo(host, port, &hints, &result)) < 0)
  {  /* error? */
```

```
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(flag));
    /* messages */
    exit(-1);                                    /* failure */
  }

  /* Iterate over the list of addresses until one works. */
  for (next = result; next; next = next->ai_next) {
    sock_fd = socket(next->ai_family, next->ai_socktype,
    next->ai_protocol);
    if (-1 == sock_fd) continue;                 /* failure */
    if (connect(sock_fd, next->ai_addr, next->ai_addrlen) !=
    -1) break; /* success */
    close(sock_fd);                  /* close and try again */
  }

  if (!next) {
    fprintf(stderr, "can't find an address\n");
    exit(-1);
  }
  freeaddrinfo(result); /* clean up storage no longer needed */

  /* Set a timeout on read operations. */
  struct timeval timeout;
  timeout.tv_sec = 2;  /* seconds */
  timeout.tv_usec = 0;
  if (setsockopt(sock_fd, SOL_SOCKET, SO_RCVTIMEO,
                 (char*) &timeout, sizeof(timeout)) < 0) {
    fprintf(stderr, "setsockopt failed\n");
    exit(-1);
  }
  return sock_fd;
}
```

The code in the file *get_connection.c* (see Listing 6-2) handles the networking details. This code also illustrates various points made throughout earlier chapters. At the center is the data type `struct addrinfo`, which encapsulates information about an IP address. The program declares a variable `hints` of this type and then initializes the structure's fields with information that provides hints to the library function `getaddrinfo`. One hint is that the program could deal with either an IPv4 or an IPv6 address (`AF_UNSPEC` for *address family unspecified*), and a second hint is that the program wants a *reliable* connection (`SOCK_STREAM` vs. `SOCK_DGRAM`), which is typically TCP based. Two other fields are initialized to zero, indicating that the *webclient* program defers to the library function to make the default choices.

A pointer to the `hints` structure is one of the arguments to library function `getaddrinfo`. Here is a summary of the four arguments passed to this function:

- The `host` argument is *`www.google.com`*, the symbolic IP address.
- The `port` argument is 80 as a string, the standard server-side port number for accepting HTTP connections.
- The third argument is `&hints`: a pointer to the `hints` structure, rather than a copy of it.
- The last argument is the pointer `results` of type `struct addrinfo*`: the library function sets this pointer to the address of a structure that contains the information about available addresses for the Google server.

A successful call to `getaddrinfo` may contain several addresses for the Google server; hence, a `for` loop is used to iterate over the options, picking the first one that supports a connection. Two key library functions are in play in the loop:

- The `socket` function returns a file descriptor on success.
- The `connect` function uses the file descriptor and address information to attempt a connection to a host.

The `socket` and `connect` functions both return -1 on failure. Once the program confirms that a usable network address is in hand, the program frees the dynamically allocated storage to which `result` points. The library function `freeaddrinfo` does whatever *nested* freeing may be needed, and so this function rather than the regular `free` function should be used.

The last configuration in this utility code involves setting a timer on the socket. The relevant type is `struct timeval`, and the library function is `setsockopt`. In this example, the timer applies only to *read* operations because of the `SO_RCVTIMEO` (*receive timeout*) flag. The timeout can be set in a mix of seconds and microseconds; in this example, the socket is configured to time out after two seconds of waiting.

After fetching Google's home page from `www.google.com`, the *webclient* program prints the HTML document to the standard output. If the program is run, there likely will be a pause of two seconds or so after the printing but before the program exits. There is no magic in the *two-second* timeout, of course; the example invites experimentation.

6.3. An Event-Driven Web Server

In an earlier example, the *fifoReader* (recall Listing 5-9) did nonblocking *read* operations on a named pipe. The *fifoWriter* sporadically populated this pipe with 16-byte chunks, each chunk consisting of four 4-byte `int` values. The *fifoReader*, in turn, tried to read 4 bytes, or one `int` value, at a time. Most of the *read* operations by the *fifoReader* failed to deliver the integer values, although all of the `int` values eventually were read. Indeed, only about 2% of the *read* operations yielded the expected `int` value—a

failure rate of 98%! The approach taken in the code example was crude and inefficient and designed only to introduce the nonblocking API. The *fifoReader* tried, on every loop iteration, to read whatever happened to be available in the named pipe. But the *fifoWriter* paused a random amount of time between *write* operations so that there was a discontinuous byte stream from the *writer* to the *reader*. The odds were overwhelmingly against successful nonblocking *read* operations by the *fifoReader*.

A different approach can improve the efficiency of *read* operations and also make application logic easier to follow. The approach involves a division of labor:

- A library function monitors a channel to detect whether there are bytes to read.
- The application can query the monitor function before even attempting a *read* operation: if the monitor detects nothing to read, the application does not bother to attempt a *read* operation.

Under this approach, the odds of successful *read* operations should improve dramatically. Moreover, there is no need to use nonblocking *reads*, as the monitor itself blocks until it detects bytes to be read.

Various C libraries have emerged, over time, for performing the *monitoring* task, with *epoll* and *kqueue* as some recent examples. A good place for an overview and analysis is the C10K project at `www.kegel.com/c10k.html`. The forthcoming *webserver* program code introduces the `select` library function, which has a long history in C.

Before moving on to the web server program, however, it may be helpful to look at a simpler example of how `select` works. The next code example uses the `select` function to check whether there are bytes to read from the standard input. If so, a single byte is read and then written to the standard output; if not, an appropriate message is printed.

Listing 6-3. Introducing the `select` function

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <unistd.h>

void main() {
  fd_set fds;         /* set of file descriptors */
  struct timeval tv;
  int flag;
  char byte;

  FD_ZERO(&fds);      /* clear the set of fds */
  FD_SET(0, &fds);    /* 0 == standard input */
  tv.tv_sec = 5;
  tv.tv_usec = 0;

  flag = select(FD_SETSIZE,  /* how many file descriptors */
                &fds,        /* file descriptors for readers */
           NULL,        /* no writers */
           NULL,        /* no exceptions */
           &tv);        /* timeout info */

  if (-1 == flag)
    perror("select error");
  else if (flag) {          /* flag == 1 == true */
    read(0, &byte, 1);      /* read the byte */
    puts("data read");
  }

  if (flag)
    printf("The byte value is: %c\n", byte);
}
```

The *selectStdin* program (see Listing 6-3) declares a variable of type `fd_set`, which represents a set of file descriptors. The macro `FD_ZERO` clears the set by zeroing out the variable, and the macro `FD_SET` adds a file descriptor to the set—in this case, the file descriptor 0 for the standard input is added. A timeout of five seconds is then configured using the `struct timeval` variable `tv`.

The library function `select` holds center stage in the example. The function, which blocks until the specified timeout occurs, is called with five arguments:

- The first argument, `FD_SETSIZE`, is the *count* of the file descriptors in the set, in this case 1. Normally, there would be multiple file descriptors in the set.
- The second argument `&fds` is the address of *readers* set.
- The third and fourth arguments, both `NULL`, are the addresses of the *writers* and *exceptions* sets, respectively. In this example, only the *readers* set has a member, and then only one.
- The fifth and final argument is the timeout configuration, a pointer to the `struct timeval` structure. If the timeout argument is `NULL`, the `select` function waits (blocks) indefinitely.

The `select` function returns -1 on error. If there is a byte to read within the timeout period of five seconds, `select` returns *true* to confirm the fact, and the program then tries to read the byte. If the timeout occurs first, there is no attempt to read because this would be wasted effort.

6.3.1. The *webserver* Program

The forthcoming *webserver* example puts the `select` function to practical use. The program has three source files and a Makefile for convenience. Two of the source files contain utility functions, whereas the code in the third file implements the application logic. This logic can be summarized now and analyzed in detail after the code displays. The summary ignores technical details taken up later.

- For convenience, the server awaits connections on port 3000 rather than on the default port of 80. Port numbers greater than 1023 do not require special administrative privileges. There is a *backlog* of 100, which means that up to 100 clients can be connected at the same time. The server can be built and started from the command line in the usual way:

  ```
  % make
  % ./web_server    ## on Windows: % web_server
  ```

  The server runs indefinitely, and so the program should be shut down with Control-C or the equivalent.

- The server uses a set of file descriptors (`fd_set`). To start, the only file descriptor in the set identifies the original socket, an *accepting* socket that awaits connecting clients. The file descriptor for this socket remains in the `fd_set` from start to finish, but other file descriptors—ones that represent *read/write* channels to clients—are added to and removed from the set of file descriptors.

- Clients attempt to connect to the web server and then to send requests. From the web server's perspective, the clients are in one of two states:
  - A *connecting* client is trying to connect and has not yet sent a request for the server to read.
  - A *requesting* client has connected and is thus able to send a request.
- If a *connecting* client succeeds in connecting, the file descriptor for the socket is placed in the `fd_set` that the `select` function monitors. The client's request now can be read when it arrives.
- If a *requesting* client is selected, its request is read, and a response is written: the response echoes back the request. After responding to a client, the server removes the client from the `fd_set`.

6.3.2. Utility Functions for the Web Server

The *webserver* program breaks out the utility functions into two separate files. These functions handle the many low-level details from getting the original file descriptor for the socket to logging information about a connecting client and through sending a response back to a client.

Listing 6-4. Utility functions for the web server

```
#include <netinet/in.h>
#include <string.h>
#include <stdio.h>
#include <arpa/inet.h>

#define BuffSize 256
```

```
void log_client(struct in_addr* addr) {
  char buffer[BuffSize + 1];
  if (inet_ntop(AF_INET, addr, buffer, sizeof(buffer)))
  /* NULL? */
    fprintf(stderr, "Client connected from %s...\n", buffer);
}

void get_response(char request[ ], char response[ ]) {
  strcpy(response, "HTTP/1.1 200 OK\n");
  /* start line */
  strcat(response, "Content-Type: text/*\n");
  /* headers... */
  strcat(response, "Accept-Ranges: bytes\n");
  strcat(response, "Connection: close\n\n");
  strcat(response, "Echoing request:\n");
  /* body of response */
  strcat(response, request);
}
```

The *servutils2.c* file (see Listing 6-4) contains two utility functions. The `log_client` function has one argument, a pointer to a `struct in_addr` (*Internet address*). This structure contains information about the client, including the client's IP address. The `log_client` function calls the library function `inet_ntop` (*Internet name to protocol*) with the structure pointer as an argument; the library function generates a human-readable string and puts the string in the caller-supplied buffer. If the web server is running on *localhost* (127.0.0.1), and a request comes from this same machine, then the message would be

```
Client connected from 127.0.0.1...
```

The get_response function creates an HTTP-compliant response consisting of an HTTP *start line*, four HTTP *headers*, and the HTTP body, if any, that came with the request. (Recall that a POST request has a body, whereas a GET request does not.) This response is sufficient for development and initial testing.

Listing 6-5. Core utilities for the webserver

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define Backlog  100

void report_and_exit(const char* msg) {
  fprintf(stderr, "%s\n", msg);
  exit(-1); /* EXIT_FAILURE */
}

int get_servsocket(int port) {
  struct sockaddr_in server_addr;

  /** create, bind, listen **/
  /* create the socket, make it non-blocking */
  int sock_fd = socket(PF_INET, SOCK_STREAM, 0); /* internet
  family, connection-oriented */
  if (sock_fd < 0)
    report_and_exit("socket(...)");

  /* bind to a local address: implementation details */
  memset(&server_addr, 0, sizeof(server_addr));
  server_addr.sin_family = AF_INET;
  server_addr.sin_addr.s_addr = INADDR_ANY;
```

```
  server_addr.sin_port = htons(port); /* host to network
  endian */
  if (bind(sock_fd, (struct sockaddr*) &server_addr,
  sizeof(server_addr)) < 0)
    report_and_exit("bind(...)");

  /* listen for connections */
  if (listen(sock_fd, Backlog) < 0) report_and_
  exit("listen(...)");
  return sock_fd;
}
```

The principal function in the *servrutils.c* file (see Listing 6-5) is `get_servsocket`, which takes a port number as its single argument. The function performs the classic three steps for setting up a web server: *create*, *bind*, and *listen*. Here are some details:

1. *Create* a socket with the library function `socket`. In this example, the socket is in the IP protocol family (`PF_INET`) and is connection based (`SOCK_STREAM`).

2. *Bind* the socket to a local port number, in this case port 3000. A `server_addr` structure is used to store the required information. The port number is passed as an argument to the `htons` library function, which converts local *endian-ness* to network *endian-ness*. Recall that Intel machines are *little endian*, whereas network protocols are *big endian*. The library function of interest here is `bind`.

3. *Listen* for up to `Backlog` clients at a time, where Backlog is 100. If 100 clients are connected already to the server, then any would-be client gets a *Connection refused* message. The library function is `listen`.

If there are no errors in the three steps, the `get_servsocket` function returns the identifying file descriptor. Otherwise, the web server exits.

WHAT'S CURL?

The *curl* command-line tool (`https://curl.haxx.se`) can fetch data through URLs. The tool is cross-platform and works with an impressive number of protocols. The *curl* tool is used later to test the web server.

Listing 6-6. A web server with `select`

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <netinet/in.h>
#include "servutils.h" /* function declarations */

#define BuffSize 250

int main() {
  const int port = 3000;
  char request[BuffSize + 1];
  memset(request, 0, sizeof(request));
  struct sockaddr_in client_addr;
  socklen_t len = sizeof(struct sockaddr_in);

  fd_set active_set, temp_set;          /* temp_set becomes
                                           a copy of
                                           active_set */
  FD_ZERO(&active_set);                 /* clear the
                                           active_set */
```

```
int sock_fd = get_servsocket(port);   /* get the original
                                         socket fd */
FD_SET(sock_fd, &active_set);         /* add it to the set */
fprintf(stderr, "Server awaiting connections on port
%i.\n", port);

while (1) {
  temp_set = active_set; /* make a working copy, as active_
  set changes */
  if (select(FD_SETSIZE, &temp_set, NULL, NULL, NULL) < 0)
  /* activity? */
    report_and_exit("select(...)");

  int i;
  for (i = 0; i < FD_SETSIZE; i++) {       /* handle the
                                              current fds */
    if (!FD_ISSET(i, &temp_set)) continue; /* member of
                                              the set? */

    if (i == sock_fd) { /** original accepting socket **/
      int client_fd = accept(sock_fd,
                             (struct sockaddr*) &client_addr,
                             &len);
      if (-1 == client_fd) continue; /* try again */
      log_client(&client_addr.sin_addr);
      FD_SET(client_fd, &active_set); /* add this fd to
                                         select list */
    }
    else {              /** read/write socket **/
      int bytes_read = read(i, request, BuffSize);
      if (bytes_read < 0) continue;
```

```
        /* Send a response. */
        char response[BuffSize * 2]; /* twice as big to
        be safe */
        memset(response, 0, sizeof(response));
        get_response(request, response);
        int bytes_written = write(i, response,
        strlen(response));
        if (bytes_written < 0) report_and_exit("write(...)");
        close(i);

        FD_CLR(i, &active_set); /* remove from active set */
      }
    }
  }
  return 0;
}
```

The *webserver* program (see Listing 6-6) uses the `select` function and its supporting macros such as `FD_SET` and `FD_CLR` to read client requests and to write back responses. The salient points can be summarized as follows:

- The primary setup is a call to the utility function `get_servsocket`, which returns the file descriptor for the socket, if successful; otherwise, the *webserver* exits as there is no point in going on. For reference, this first socket is the *accepting* socket because its job is to accept client connections. The *accepting* socket is not used as a channel to read requests and write responses. Among the sockets used in the application, there is a strong separation of concerns: one socket accepts client connections, whereas all of the others act as read/write channels between the web server and its clients.

- The *accepting* socket's file descriptor is added, using the `FD_SET` macro, to the `fd_set` variable named `active_set`. This file descriptor is the one permanent member of the `active_set`.
- After a client connects, this socket's file descriptor is added to the `active_set`; after a client receives its response, the same file descriptor is removed from the `active_set`.
- The program has two loops: an outer `while` loop that iterates indefinitely and an inner `for` loop that iterates over a *copy* of the `active_set` named the `temp_set`. The copy is important because of what happens in a loop iteration. During a `for` loop iteration, file descriptors may be added to and removed from the `active_set`: added if a new client connects and removed if a client receives a response. At the top of the outer `while` loop, the `active_set` is thus copied into the `temp_set`, and the iteration is over this temporary copy, which does not change during `for` loop execution.
- The second statement in the `while` loop is a *blocking* call to `select`, which monitors only the *read* set named `temp_set`. There is no monitoring of *writers* and *exceptions* (the third and fourth arguments), and the `select` does not have a timeout: the `select` should *not* return unless there is client activity of some kind—connecting or requesting.

- Once the `select` function returns, the inner `for` loop iterates over the file descriptors in the `temp_set`. For each member of this set, there are two possibilities:
  - The file descriptor is of the single *accepting* socket; hence, a client connection is pending. The program uses the library function `accept` to finalize the connection and to get the connecting socket's descriptor. This file descriptor is added to the `active_set` to enable *read/write* operations later. For reference, this socket is the *client socket*.
  - The file descriptor is of a client socket used for *read/write* operations. In this case, the client's request is read and then echoed back as a response. Examples follow shortly. Once the response has been sent, the socket's descriptor is passed as an argument to `close`, which effectively breaks the connection. This descriptor also is *removed* from the `active_set`. The conversation with the client is short and sweet: the client sends one request and gets one response in return.

6.3.3. Testing the Web Server with *curl*

There are various ways to test the *webserver* program. For example, the earlier *webclient* program might be used, but this program is not sufficiently flexible to go beyond preliminary testing. The *curl* utility, by contrast, is well suited for the task. As an example, the *curl* command

```
% curl localhost:3000?msg=Hello,world!
```

generates the following response, with comments following ##:

```
Echoing request:
GET /?msg=Hello,world! HTTP/1.1   ## GET request with a
                                     query string
User-Agent: curl/7.35.0           ## user program is curl
Host: localhost:3000              ## localhost on port 3000
Accept: */*                       ## accept any MIME type/
                                     subtype combination
```

By contrast, the *curl* command

```
curl --data "name=Fred Flintstone&occupation=handyman"
localhost:3000
```

generates this response:

```
Echoing request:
POST / HTTP/1.1                              ## POST, not GET
User-Agent: curl/7.35.0
Host: localhost:3000
Accept: */*
Content-Length: 40                           ## in bytes for
                                                HTTP body
Content-Type: application/x-www-form-urlencoded
## POSTed form
                                ## two newlines end the headers
name=Fred Flintstone&occupation=handyman     ## body of
                                                POST request
```

The *webserver* is an *iterative* rather than a *concurrent* server: the server handles one request at a time, completing the response to a given request before turning to the next request. In more technical terms, the *webserver* program executes as a single process with a single thread of execution

and thus uses neither of the standard concurrency mechanisms—multiprocessing and multithreading. For development and testing, an iterative server is acceptable and even preferable because it is relatively easy to debug the connect/request/response trio. Modern languages typically have libraries for development web servers (e.g., the Ruby WEBrick library), and these web servers are typically iterative. However, any production-grade web server is going to be concurrent. The next chapter focuses on concurrency. The next section in this chapter moves from HTTP to HTTPS to analyze wire-level security in web connections.

6.4. Secure Sockets with OpenSSL

The *S* in HTTPS is for *secure*. Various security layers are suitable for sitting atop HTTP, including SSL (Secure Sockets Layer, from Netscape) and TLS (Transport Layer Security, derived from SSL). SSL and TLS are distinct but sometimes lumped together as SSL/TLS.

Among the production-grade and most popular implementations of SSL and TLS is OpenSSL (`www.openssl.org/`). OpenSSL also includes a full library for cryptography: functions for message digests, digital signatures, digital certificates, encryption/decryption, and more. OpenSSL can be installed as a development environment—header files and implementation libraries. Once OpenSSL is installed, the header files and libraries are typically in *openssl* subdirectories such as in */usr/include/openssl* and */usr/lib/openssl*, respectively.

HTTPS provides *wire-level* or *transport-level* security, as opposed to *users/roles* security in which a user provides an identity (e.g., a login name) and a credential (e.g., a password) to confirm the identity. The wire-level security comprises three major services: peer authentication (mutual challenge), confidentiality (data encryption/decryption), and reliability (message sent equals message received). These are clarified in order.

Consider a scenario in which Alice and Bob exchange messages over a channel:

```
        messages
Alice<------------>Bob
```

How does Alice know that it is Bob, and not an impostor, at the other end? The same goes for Bob. The eavesdropper Eve might be in the middle (*man-in-the-middle attack*), pretending to be both Alice and Bob, thereby intercepting all of the messages sent in one direction or the other. Alice and Bob need a procedure (*peer authentication*) so that each can authenticate the other's identity before any significant messages are sent between them.

Peer authentication, as used in HTTPS, requires a *key pair* apiece for Alice and Bob: a digital *public key* (distributable to anyone) and a digital *private key* (secret to its owner). The public key is an *identity*. For example, Amazon's public key identifies Amazon, and Alice's public key identifies her. A public key can be embedded in a digital certificate, with a certificate authority (CA) vouching for this key through the CA's own digital signature on the same certificate. For example, a CA such as VeriSign or RSA vouches with its own digital signature that the public key on Alice's digital certificate indeed identifies Alice. The vouching may come with a fee, of course.

Here is a scenario for *peer authentication* between Alice and Bob:

1. Alice sends a signed certificate request containing her name, her public key, and some additional information to a CA such as VeriSign or RSA. Assume that the public key is unique.

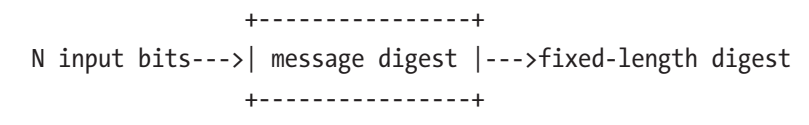
2. The CA creates a message *M* from Alice's request, signing the message *M* with the CA's own private key, thereby creating a separate signature message *DSIG*.

3. The CA returns to Alice the message *M* with its signature *DSIG*. Together *M* and *DSIG* form the core of Alice's certificate. The certificate has a *from* and a *to* date together with some other information.
4. Alice sends her newly minted certificate to Bob, and the certificate contains Alice's public key.
5. Bob verifies the signature *DSIG* using the CA's public key. If the signature is verified, Bob accepts the public key in the certificate as Alice's public key, that is, as her identity.
6. Bob repeats Alice's steps.

There is, of course, a fly in this ointment. If Eve manages to get a copy of Alice's digital certificate and also manages to intercept an authentication request from Bob to Alice, then Eve becomes indistinguishable from Alice. To guard against this possibility, Bob might request from Alice several digital certificates, each with a different signer and with different validity dates. There also are certificates with more than one CA as a signer. When it comes to peer authentication, there are precautions rather than guarantees.

WHAT'S A MESSAGE DIGEST?

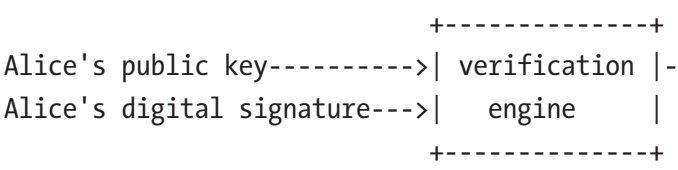
A *message digest*, also called a *hash*, is a fixed-length digest of input bits:



For example, SHA-1 (Secure Hash Algorithm 1) generates a 160-bit digest of any input bits. Duplicate digests from different inputs are possible, but unlikely. A digest is *one-way secure*: it is relatively easy to compute the digest, but it is computationally intractable to go from the digest back to the original input bits—even if the digest algorithm is known.

One more fly in the ointment deserves mention. As noted earlier, a digital certificate contains a CA's digital signature to vouch for the public key on the certificate. What is a *digital signature*, and how is one to be verified?

A *digital signature* is a message digest (see the sidebar) encrypted with the *private* key from a key pair. To create her own digital signature, Alice would create a message digest of information about her (e.g., name, city of residence, employer's name, and so on) and then encrypt this digest with her *private* key. This signature then can be verified with the *public* key from the same pair. If Bob has Alice's *public* key, Bob can verify Alice's digital signature:



Validating a CA's digital signature requires the CA's public key: a CA's public key is available on the CA's own digital certificate, which in turn has a digital signature as a voucher. Thus begins the verification regress. At some point, of course, the regress stops because a digital signature is accepted as valid.

Listing 6-7. A sample X.509 digital certificate

```
Certificate:
   Data:
     Signature Algorithm: md5WithRSAEncryption
     Issuer: C=ZA, ST=Western Cape, L=Cape Town, O=Thawte
        Consulting cc,
     ...
     CN=Thawte Server CA/emailAddress=server-certs@thawte.com
     Validity
        Not Before: Aug  1 00:00:00 1996 GMT
        Not After : Dec 31 23:59:59 2028 GMT
     Subject Public Key Info:
        Public Key Algorithm: rsaEncryption
        RSA Public Key: (1024 bit)
          Modulus (1024 bit):
            00:d3:a4:50:6e:c8:ff:56:6b:e6:cf:5d:b6:ea:0c:
            ...
            3a:c2:b5:66:22:12:d6:87:0d
          Exponent: 65537 (0x10001)
     ...
     Signature Algorithm: md5WithRSAEncryption
        07:fa:4c:69:5c:fb:95:cc:46:ee:85:83:4d:21:30:8e:ca:d9:
        ...
        b2:75:1b:f6:42:f2:ef:c7:f2:18:f9:89:bc:a3:ff:8a:2
        3:2e:70:47
```

The *dcert* display (see Listing 6-7) shows parts from a sample digital certificate, with Thawte as the CA. The public key algorithm is RSA, the industry standard. The certificate also gives details about the digital signature.

With web sites as opposed to web services, peer authentication typically becomes *one-way* authentication: the browser, as the client application, challenges the web server to establish its identity through one or more digital certificates, but the web server usually does not challenge the browser. For web services, by contrast, the challenge may be mutual.

The second HTTPS service is *confidentiality*, achieved through the encryption of sent messages and the corresponding decryption of received messages:

```
      msg  +----------+ encrypted msg  +----------+ msg
Alice----->| encrypt |---------------->| decrypt |----->Bob
           +----------+                 +----------+
```

Here is a depiction of how encryption and decryption work:

```
                +------------+ encrypted bits  +------------+
plainbits------>| encryption |---------------->| decryption |--->plainbits
encryption key->| engine     |         +------>| engine     |
                +------------+         |       +------------+
                       decryption key--+
```

There are two general approaches to encryption/decryption, depending on whether the same key is used for both operations:

- In the *symmetric* approach, the *same* key is used to encrypt and decrypt. The upside is that this approach is very efficient, about a thousand times faster than the alternative explained in the following. The downside is the *key distribution problem*: How is the key to be distributed to both Alice and Bob?

- In the *asymmetric* approach, one key is used to encrypt, but a *different* key is used to decrypt. The upside is that this approach solves the key-distribution problem. For example, Alice can encrypt a message using Bob's *public* key, but only Bob can decrypt this message because he has the one and only copy of his *private* key. The downside is that this approach is about a thousand times slower than the *symmetric* approach.

HTTPS uses a clever combination of the two approaches:

1. After the client and the server have agreed upon a *cryptographic suite* of algorithms, and the client has received at least one acceptable digital certificate from the server during the authentication phase, the client generates a *premaster secret*, bits that will be used on both sides to generate a *session key*.
2. The client encrypts the premaster secret with the server's *public* key and sends the encrypted bits over the wire.
3. The server (and presumably the server alone) can decrypt these encrypted bits using the server's *private* key.
4. During the rest of the conversation between client and server, the *session key* is used both to encrypt and decrypt bits; hence, the *symmetric* approach is now used for efficiency.

The third major HTTPS service, message reliability, checks whether the sent message is the same as the received message:

```
      sent message      received message
Alice-------------->...------------------>Bob   ## Sent message
                                                = received
                                                message?
```

Recall that the client and the server have settled on a cryptographic suite, which includes a message digest (hash) algorithm. The sender computes a hash of the message to be sent and sends the hash as well. The receiver recomputes the hash locally, using the same algorithm, and then checks whether the received hash matches the locally computed one. Assume that the locally computed hash is correct. If the two hashes do not match, then something in the sent message (the original message and/or the sender's hash) has been corrupted in transit; the message and a hash need to be sent again.

The *wcSSL* program is an HTTPS client that exhibits the security features discussed previously. The OpenSSL libraries do a nice job of wrapping the usual HTTP client functions—create a socket, open a connection, engage in a conversation, close the connection—within security-enabled counterparts. The resulting flow of control is easy to follow. For readability, the source code for *wcSSL* program is divided among three files. A Makefile is included.

The three source files in the *wcSSL* program are as follows:

- The header file *wcSSL.h* has the required *include* directives for the standard libraries and for OpenSSL. This file also declares five utility functions defined in the file *wcSSLutils.c*.
- The source file *wcSSLutils.c* defines five utility functions, which are clarified shortly.
- The source file *wcSSL.c* contains the high-level logic. The code tries to open an HTTPS connection to Google; calls a stub function to verify the Google certificate; sends a request over the now encrypted channel; and prints the response, which again is the Google home page.

Listing 6-8. The header file wcSSL.h

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <openssl/bio.h>
#include <openssl/ssl.h>
#include <openssl/x509.h>
#include <openssl/x509_vfy.h>

extern void report_exit(const char* msg);
extern void load_SSL();
extern int verify_dc(int ver, X509_STORE_CTX* x509_ctx);
extern void view_cert(SSL* ssl, BIO* out);
extern void cleanup(BIO* out, BIO* web, SSL_CTX* ctx);
```

The five functions declared in the header file *wcSSL.h* (see Listing 6-8) can be clarified as follows:

- The `report_exit` function prints an error message before exiting. The error (e.g., a socket connection cannot be opened) makes it impossible to continue.
- The `load_ssl` function calls various OpenSSL functions, which in turn load the required OpenSSL modules.
- In production mode, the `verify_dc` function would check the certificate(s) sent from Google during the HTTPS handshake. The details of verification can differ widely depending on how a system stores trusted digital certificates. One straightforward approach is to have a persistent *store* of trusted certificates on the client machine or local network. For instance, there might be a local file with a copy of a trusted Google

certificate in either a text format such as PEM (*Privacy-Enhanced Mail*) or a binary format such as DER (*Distinguished Encoding Rules*). OpenSSL has utilities to convert from one standard format to another. In any case, a Google certificate downloaded during the *peer authentication* phase would be compared against a stored copy, using OpenSSL functions designed for the purpose. If there is no such local copy, then the certificate's digital signature from a CA could be verified instead. The current example omits these details by having the `verify_dc` function simply return *true* (1). The `verify_dc` function is thus a *stub* that needs to be fleshed out for production.

Unix-like systems typically include a directory such as */etc/ssl/certs*, which contains accepted digital certificates. This directory thus acts as the local truststore for such certificates.

- The `view_cert` function prints the subject line from the certificate to confirm its arrival.
- The `cleanup` function calls OpenSSL utility functions to free heap storage.

These five functions are defined and *wcSSLutils.c* and called in the main program file *wcSSL.c*.

Listing 6-9. The utilities file cwSSLutils.c

```
#include "wcSSL.h"

void report_exit(const char* msg) {
  puts(msg);
  exit(-1);
}
```

```
void load_SSL() {  /* load various OpenSSL libraries */
  OpenSSL_add_all_algorithms();
  ERR_load_BIO_strings();
  ERR_load_SSL_strings();
  SSL_load_error_strings();
  if (SSL_library_init() < 0) report_exit("SSL_library_init()");
}

int verify_dc(int ver, X509_STORE_CTX* x509_ctx) {  /* stub
function */
  /* In production, a full verification would be needed. */
  return 1;
}

/* Extract the subject line for the certificate, then free
storage. */
void view_cert(SSL* ssl, BIO* out) {
  X509* cert = SSL_get_peer_certificate(ssl);
  if (NULL == cert) report_exit("SSL_get_peer_
  certificate(...)");

  X509_NAME* cert_name = X509_NAME_new();
  cert_name = X509_get_subject_name(cert);
  BIO_printf(out, "Certificate subject:\n");
  X509_NAME_print_ex(out, cert_name, 0, 0);
  BIO_printf(out, "\n");
  X509_free(cert);
}

void cleanup(BIO* out, BIO* web, SSL_CTX* ctx) {
  if (out) BIO_free(out);
  if (web) BIO_free_all(web); /* handles nested frees */
  if (ctx) SSL_CTX_free(ctx); /* ditto */
}
```

The `load_SSL` function in the *wcSSLutils.c* file (see Listing 6-9) calls four functions from the OpenSSL API in order to load various SSL modules. The `load_SSL` then calls a fifth OpenSSL function `SSL_library_init` to do whatever SSL initialization is required. Any error in the initialization would make it impossible to continue; hence, the *wcSSL* client exits if an error occurs.

The `view_cert` function gets the X509-formatted certificate from Google, extracts some information, and then prints this information. X509 is versioned and remains the dominant format for digital certificates; hence, OpenSSL includes many functions with `X509` in the name. Once information about the certificate is printed, in this case only the subject line, the heap storage for the certificate is freed. The `X509_free` utility function does whatever nested freeing is required; hence, this function and *not* the library function `free` should be called.

Throughout the *wcSSL* program, there are calls to various OpenSSL functions with `BIO` (*Basic Input/Output*) in the name. The `BIO` library is roughly a wrapper around the standard `FILE` type, and the `BIO` API mimics the `FILE` API. However, the `BIO` functions have access to the all-important SSL *context*, which is discussed shortly.

In working with the OpenSSL libraries, it is best practice to use the `BIO` functions for any input/output operations that involve web content. Accordingly, the *wcSSL* program uses the standard `puts` function in `report_exit` but otherwise sticks with the `BIO` input/output functions. For instance, the `BIO_puts` function is used to send the request, over an encrypted channel, to the Google web server.

Listing 6-10. The main source file wcSSL.c

```
#include "wcSSL.h"
#define BuffSize 2048

int main() {
  const char* host_port = "www.google.com:443";
```

```
const char* request = "GET / \r\nHost: www.google.com\r\
nConnection: close\r\n\r\n";
BIO* out = BIO_new_fp(stdout, BIO_NOCLOSE); /* standard
                                              output */
SSL* ssl = NULL;
/* primary data structure for SSL connect */

load_SSL();
const SSL_METHOD* method = SSLv23_method(); /* protocol
version */
if (NULL == method) report_exit("SSLv23_method()");
SSL_CTX* ctx = SSL_CTX_new(method);
/* global context for client/server */
if (NULL == ctx) report_exit("SSL_CTX_new(...)");

BIO* web = BIO_new_ssl_connect(ctx);  /* BIO is roughly FILE,
but with SSL baked in */
if (NULL == web) report_exit("BIO_new_ssl_connect(...)");
if (1 !=  BIO_set_conn_hostname(web, host_port)) report_
exit("BIO_set_conn_host(...)");
BIO_get_ssl(web, &ssl); /* the security layer atop HTTP */
if (NULL == ssl) report_exit("BIO_get_ssl(...)");

if (BIO_do_connect(web) <= 0) report_exit("BIO_do_
connect(...)");     /* connect */
if (BIO_do_handshake(web) <= 0) report_exit("BIO_do_
handshake(...)"); /* handshake */
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, verify_dc);
if (!SSL_get_verify_result(ssl)) report_exit("SSL_get_
verify(...)");  /* verify cert */
view_cert(ssl, out);                       /* look at cert */

BIO_puts(web, request);   /* the GET request */
```

```
  int len = 0;
  do {                    /* read chunks from Google server */
    char buff[BuffSize] = { };
    len = BIO_read(web, buff, sizeof(buff));
    if (len > 0) BIO_write(out, buff, len);
  } while (len > 0 || BIO_should_retry(web));

  cleanup(out, web, ctx);  /* free heap storage */

  return 0;
}
```

The main file for the *wcSSL* program is *wcSSL.c* (see Listing 6-10). Rather than analyze each OpenSSL function call separately, it may be more useful to group the calls, focusing on what each group is meant to accomplish. The following describes three groups in turn:

- The *init* group specifies the SSL version to be used, in this case with the OpenSSL call `SSLv23_method`. This function constructs an `SSL_CTX` instance, which is the global context for all of the remaining OpenSSL calls. The `SSL_CTX` tracks the state of the SSL session, from setup through cleanup; this context is the last item to be freed in the program.
- The *socket* group then uses the `SSL_CTX` instance (`ctx` is the variable) to create an SSL layer atop HTTP. The secure channel is named `web` in this program and is the secure counterpart of a file descriptor. Writing the request to and reading the response from Google uses the `web` variable. The standard `socket` call now occurs under the hood, in the OpenSSL libraries.

- The *connect* group establishes a connection, performing the handshake operations that include authentication. In this case, the authentication is *one way* rather than *peer* because the Google server does *not* challenge the *wcSSL* program (the client) for a certificate; but the call to the OpenSSL `BIO_do_handshake` function does result in a challenge to the Google server. The `SSL_CTX` is used again, this time to declare a *callback* function (in this case, `verify_dc`) that is to verify the Google certificate. Fine-tuning is possible here and would be appropriate in a production environment. In this example, the interest is in verifying that a certificate arrived, rather than in its validity. Google sends three certificates in response to the challenge.
- The *request/response* group uses the OpenSSL function `BIO_puts` to send the GET request to Google and the `BIO_read` function to read the response. The `BIO_write` function writes the response to the standard output. The `BIO_read` and `BIO_write` functions are the counterparts of the standard `read` and `write` functions, but the `BIO` functions have access to the `SSL_CTX`.
- The *cleanup* group uses OpenSSL functions to free heap storage allocated in the course of setting up and using the HTTPS connection.

To confirm that a certificate arrived from Google, the *wcSSL* program prints the subject line:

```
Certificate subject:
C=US, ST=California, L=Mountain View, O=Google Inc, CN=www.
google.com
```

As noted earlier, there is a Makefile to build the *wcSSL* program. The program also can be built with this command, which is part of the Makefile:

```
% gcc -o wcSSL wcSSL.c  wcSSLutils.c  -lssl -lcrypto -I.
```

The two *link* libraries (the two `-l` flags) are the OpenSSL library and the standard cryptography library, respectively. In the flag at the end `-I.`, the `I` is for *include* files, and the period represents the current working directory, which means that only this directory should be searched for any *include* files. In general, any search path could be specified for *include* files.

6.5. What's Next?

Concurrency and parallelism are distinct but related concepts. A *concurrent* program handles multiple tasks within the *same* time span. For example, a concurrent web server might handle, say, 20 client requests within a second or so. Concurrency is possible even on an old-fashioned, single-CPU machine through time-sharing: one task gets the CPU for a certain amount of time, and then its processing is preempted so that another task can have a turn, and so on. A concurrent program becomes a truly parallel one if the tasks are delegated to separate processors so that all of tasks can be processed *literally* at the same time. There is also instruction-level parallelism on modern machines; this parallelism involves the execution of instructions that perform machine-level operations in parallel. The next chapter fleshes out the details of concurrency and parallelism with code examples.

CHAPTER 7

Concurrency and Parallelism

7.1. Overview

A *concurrent program* handles more than one task at a time. A familiar example is a web server that handles multiple client requests at the same time. Although concurrent programs can run even on a single-processor machine of bygone days, these programs should show a marked gain in performance by running on a multiprocessor machine: different tasks can be delegated to different processors. A *parallel program* in this sense is a concurrent program whose tasks can be handled literally at the same time because multiple processors are at hand.

The two traditional and still relevant approaches to concurrency are *multiprocessing* and *multithreading*. Applications such as web servers and database systems may mix the approaches and throw in acceleration techniques such as nonblocking I/O. Multiprocessing has a relatively long history and is still widespread. For example, early web servers supported concurrency through multiprocessing; but even state-of-the-art web servers such as Nginx are multiprocessing systems.

Recall that a *process* is a program in execution and that each process has its own address space. Two processes could share a memory location, but this requires setup: shared memory is *not* the default. Separate address

M. Kalin, *Modern C Up and Running*, https://doi.org/10.1007/978-1-4842-8676-0_7

spaces are appealing to the programmer, who does need to worry about memory-based race conditions when writing a multiprocessing program. A typical *race condition* arises when two or more operations, at least one of which is a *write*, could access the same memory location at the same time. Of interest now is that separate processes, by default, do not share access to a memory location, which is requisite for such a race condition.

What is the downside of multiprocessing? When the operating system preempts a not-yet-finished process, a process-level *context switch* occurs: the operating system gives the processor to another process for its execution. The preempted process must be scheduled again to complete its execution. A process-level context switch is expensive because the operating system may have to swap data structures such as page tables (virtual-to-physical address translators) between memory and disk; in any case, there is nontrivial bookkeeping to track the state of both the preempted and the newly executing process. It is hard to come up with an exact figure, but a process-level context switch takes about 5ms to 15ms (milliseconds), time that is not available for other tasks.

Recall too that a *thread* (short for *thread of execution*) is a sequence of executable instructions. Every process has at least one thread; a process with only one thread is *single threaded*, and a process with more than one thread is *multithreaded*. Operating systems schedule *threads* to processors; to schedule a process is, in effect, to schedule one of its threads. On a multiprocessor machine, multiple threads from the same process can execute at the very same time. A thread-level context switch—preempting one thread in a process for another in the *same* process—is not free, but the cost is very low: nanoseconds rather than milliseconds. Multithreading is efficient.

In a simplifying move, Linux systems turn process scheduling into thread scheduling by treating even a multithreaded process as if it were single threaded. A multithreaded process with N threads then requires N scheduling actions to cover the threads. Threads within a multithreaded process remain related in that they share resources such as memory

address space. Accordingly, Linux threads are sometimes described as lightweight processes, with the *lightweight* underscoring the sharing of resources among the threads within a process.

What is the downside of multithreading? Threads within a process have the *same* address space; hence, multithreaded programs are susceptible to memory-based race conditions. On a multiprocessor machine, for instance, one thread might try to *read* memory location N at the very instant that another thread is trying to *write* N. The outcome is indeterminate. The burden of preventing race conditions falls on the programmer, not the operating system. Multithreaded programs, especially ones with variables shared among the threads, are a challenge even for the experienced programmer.

7.2. Multiprocessing Through Process Forking

The standard library functions provide options for multiprocessing, but the `fork` function is the most explicit. The first code example covers the basics of a `fork` call using *unnamed* pipes; an earlier example (recall Listings 5-8 and 5-9) covered named pipes. A look at unnamed pipes from the command line serves as preparation.

At the command line, the vertical bar | represents an unnamed pipe: to the left is the *pipe writer* and to the right is the *pipe reader*. Each is a process. Here is a contrived example using the sleep and echo utilities available on Unix-like systems and through Cygwin:

```
% sleep 5 | echo "Hello, world!"
```

The greeting *Hello, world!* appears on the screen; then, after about five seconds, the command-line prompt returns, signaling that both the sleep and echo processes have exited. The pipe is closed automatically when the

reader and *writer* terminate. There is multiprocessing here, but it does no useful work; instead, the example shows how the unnamed pipe works.

In normal usage, the *writer* process on the left writes bytes to the pipe, and the *reader* process on the right *blocks* until there are bytes to read. By closing the *write* end of a pipe before exiting, the *writer* process thereby generates an end-of-stream condition. The *reader* process closes the *read* end before exiting as well. Once the reader and the writer process exit, the pipe shuts down.

The preceding example is contrived because the sleep process does not write any bytes to the pipe and the echo process does not read any bytes from the pipe. Nonetheless, there is multiprocessing. The sleep process on the left does just that, and for five seconds. In the meanwhile, the echo process immediately writes its greeting to the screen because this process need not wait for bytes from the pipe. The echo process exits after printing its message. The sleep process then exits, the pipe goes away, and the command-line prompt reappears.

The first code example focuses on the basics of `fork`. The second example then uses the `pipe` library function in a multiprocessing example with an unnamed pipe.

Listing 7-1. Introducing the fork function

```
#include <sys/types.h> /* just in case... */
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void main() {
  signal(SIGCHLD, SIG_IGN);    /* prevents zombie */
  int n = 777;                 /* both parent and child have
                                  a copy */
```

```
  pid_t pid = fork();
  if (-1 == pid) {              /* -1 signals an error */
    perror(NULL);
    exit(-1);
  }

  if (0 == pid) {           /** child **/
    n = n + 10;
    printf("%i\n", n);      /** 787 ***/
  }
  else {                    /** parent **/
    n = n * 10;
    printf("%i\n", n);      /** 7770 */
  }
}
```

The *basicFork* program (see Listing 7-1) opens with a call to the `signal` function. This is a precaution to prevent *zombie* processes, as clarified in an upcoming section. The `int` variable `n` is declared and initialized to 777. If the subsequent call to the library function `fork` succeeds, both the child and the parent process get their own separate copy of variable `n`; hence, each process manages *different* variables with the same name.

The library function `fork` tries to create a new process. If the attempt succeeds, the newly created process becomes the *child* of the original process, which is now a *parent*. The `fork` function returns an integer value; for portability, the recommended type is `pid_t`, where `pid` stands for *process identifier*. The tricky part of the `fork` call is that, if successful, it returns one value to the parent—but a *different* value to the child. A short digression into the *process id* explains.

Every process has a nonnegative integer value as its identifier (*pid*). There is a library function `getpid` to retrieve the *pid*, and a related function `getppid` to retrieve the *parent process identifier* (*ppid*). Every process except the first has a *ppid*, which is guaranteed to be the same as the parent's *pid*.

If the `fork` call fails to spawn a child process, it returns -1 to signal the error. If `fork` succeeds, it returns

- 0 to the child
- The child's *pid* to the parent

Once forked, the child process executes a copy of the very same code as the parent—the code that comes *after* the call to `fork`. Accordingly, a test is typically used (in this case, the `if` test) to distinguish between code intended for the child and code intended for the parent. In this example, the child executes the `if` block, printing 787; the parent executes the `else` block, printing 7770. The order in which the prints occur is indeterminate. If the program runs on a multiprocessor machine, this *concurrent* program can execute in a truly *parallel* fashion.

The second code example uses an *unnamed* pipe for interprocess communication. The parent again calls `fork` to spawn a child process, and the two processes then communicate through the pipe: the parent as the *writer* process and the child as the *reader* process. The discussion also explains *zombie* processes and how to *reap* them.

Listing 7-2. The basics of the fork function

```
#include <sys/wait.h> /* wait */
#include <stdio.h>
#include <stdlib.h>   /* exit functions */
#include <unistd.h>   /* read, write, pipe */
#include <string.h>

#define ReadEnd  0
#define WriteEnd 1
```

```
void report_and_die() {
   perror(NULL);
   exit(-1);    /** failure **/
}

void main() {
  int pipeFDs[2]; /* two file descriptors */
  char buf;       /* 1-byte buffer */
  const char* msg = "This is the winter of our discontent\n";
  /* bytes to write */

  if (pipe(pipeFDs) < 0) report_and_die();
  pid_t cpid = fork();              /* fork a child process */
  if (cpid < 0) report_and_die();   /* check for failure */

  if (0 == cpid) {    /*** child ***/     /* child process */
    close(pipeFDs[WriteEnd]);              /* child reads,
                                              doesn't write */

    while (read(pipeFDs[ReadEnd], &buf, 1) > 0)   /* read until
                                                     end of byte
                                                     stream */
      write(STDOUT_FILENO, &buf, sizeof(buf));    /* echo to
                                                     the standard
                                                     output */
    close(pipeFDs[ReadEnd]);   /* close the ReadEnd:
                                      all done */
    _exit(0);                  /* exit fast */
  }
  else {              /*** parent ***/
    close(pipeFDs[ReadEnd]);  /* parent writes, doesn't read */

    write(pipeFDs[WriteEnd], msg, strlen(msg));   /* write the
                                                     bytes to
                                                     the pipe */
```

```
    close(pipeFDs[WriteEnd]);     /* done writing:
                                     generate eof */
    wait(NULL);                   /* wait for child to exit */
    exit(0);                      /* exit normally */
  }
}
```

The *pipeUN* program (see Listing 7-2) uses the `fork` function for multiprocessing and the `pipe` function for creating an unnamed pipe so that the processes can communicate. To begin, here is an overview of the library function `pipe`:

- The `pipe` function takes an `int` array of two elements as its single argument: the first element (index 0) is the file descriptor for *read* operations, and the second element (index 1) is the file descriptor for *write* operations.
- The function returns -1 to signal failure and 0 to signal success.
- Note that the `pipe` function creates an *unnamed* pipe, whereas the `mkfifo` function creates a *named* pipe.

The `fork` function is used to create the *reader* process, although this spawned process could have been the *writer*. The process that does the forking is the *parent*, and the forked process is the *child*. The child process, an almost exact duplicate of the parent, is said to *inherit* from the parent. For example, a forked child process inherits open file descriptors from the parent. Recall that once forked, the child process executes the very same code as the parent process, unless an *if* test or the equivalent is used to divide the code that each process executes. A closer look at the example clarifies.

Here, for quick review and with added detail, are the values that the `fork` function can return:

- A returned value of -1 indicates an error: the `fork` failed to spawn a child process. This could occur for various reasons, including a full process table. The *process table* is a data structure that the operating system maintains in tracking processes.
- If the `fork` call succeeds, it returns *different* values to the child and the parent processes:
  - 0 is returned to the *child*.
  - The child's process identifier (*pid*) is returned to the parent.

The *pipeUN* program uses an `if else` construct to distinguish between the parent and the child. Keep in mind that *both* processes execute this test:

```
if (0 == cpid) {    /*** child ***/
```

The `else` clause is thus for the parent to execute. Because the child process is the *reader*, it immediately closes the `WriteEnd` of the pipe; in a similar fashion, the parent process as the *writer* immediately closes the `ReadEnd` of the pipe. Both file descriptors are *open* because of the call to `pipe`. By closing one end of the pipe, each process exhibits the *separation-of-concerns* pattern.

The *writer* process then writes bytes to the pipe, and the *reader* process reads these bytes one at a time. When the *writer* process closes the pipe's *write* end, an end-of-stream marker is sent to the *reader*, which responds by closing the pipe's *read* end. At this point, the pipe closes down.

7.2.1. Safeguarding Against Zombie Processes

In the *pipeUN* program, the parent process writes a full string to the pipe and then waits for the child process to terminate with the call to library function `wait`; the child reads the string byte by byte. The `wait` call is a precaution against creating a permanent zombie process: a *zombie* is a process that has terminated but which still has an entry in the process table. If zombies are not *reaped* from the process table, this table can fill—and thus prevent the forking of any other process. Although a forked child is largely independent of its parent process, the operating system does notify the parent when the child terminates. If a child terminates *after* its parent, and there is no safeguard against zombies, the child can remain a zombie.

In the *pipeUN* example, it is unpredictable whether the parent or the child will terminate first, and so the parent—the process being notified—makes the precautionary call to `wait`: if the child has already exited, the call has no effect; otherwise, the parent's execution is suspended until the child terminates. The `wait` function expects one argument, the address of an `int` variable that stores the exit code of the process being waited on. In this example, the argument of `NULL` is used to keep things simple, but a parent process in general might implement different logic depending on the status code of a terminated child. There is also a `waitpid` function of three arguments, which allows for more granular control. The `waitpid` function is used in a forthcoming example.

The *pipeUN* program adopts another safeguard. The child calls library function `_exit` rather than `exit`: the former fast-tracks parent notification and so speeds up the reaping of a zombie entry. The parent process, by contrast, calls the regular `exit` function.

There are different ways to safeguard against zombies. The *pipeUN* program uses the `wait` approach to illustrate how independently executing processes still can be coordinated. A simpler approach, used in the *basicFork* program, is to make this call to `signal` at the start of the program:

```
signal(SIGCHLD, SIG_IGN); /* ignore signal about a child's
                             termination */
```

The effect of this call is to automate the reaping of a zombie. Were this approach taken in the current example, the parent's call to `wait` would not be needed to safeguard against a zombie.

7.3. The exec Family of Functions

In the forking of a child process, the multiprocessing is obvious in that the parent process, which calls `fork`, continues to execute as well; indeed, the parent and the child execute the *same* code unless program logic explicitly controls which process executes which code. The typical approach, illustrated in the code examples so far, is to use an *if*-test to separate the code intended for the parent from the code intended for the child.

The functions in the *exec* family, mentioned several times already but not yet analyzed, work differently. All of the functions in the family do essentially the same thing, but their argument formats differ. For example, the `execv` function has an argument vector, implemented as a `NULL`-terminated array of strings. Other members of the family such as `execle` use an *environment variable* to pass information to the executing program. The next code example goes into the details.

WHAT'S A PROCESS IMAGE?

Recall that a process is a program in execution, something dynamic. The executable program is stored somewhere, typically as a file on a local disk. To execute the program, the operating system first must load the file into memory. This in-memory representation of the process, read-only during process execution, is the process image.

Listing 7-3. The exec family of functions

```
#include <sys/types.h>  /* for safety: maybe there's no
                        unistd.h */
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
  pid_t pid = fork();      /* try to create a child process */
  if (-1 == pid) {         /* did the fork() work? */
    perror("fork()");      /* if not, error message and exit */
    exit(-1);
  }

  if (!pid) {              /* fork() returns 0 to the child */
    char* const args[ ] =
      {"./cline", "foo", "bar", "123", NULL}; /* some cmd-line
                                                 args: NULL to
                                                 terminate */
    int ret = execv("./cline", args);   /* "v" for "vector" */
    if (-1 == ret) {                    /* check for failure */
      perror("execv(...)");
      exit(-1);
```

```
    }
    else
      printf("This should not print!\n");     /* never
                                                executes */
  }
  return 0;
}
```

The *execing* program (see Listing 7-3) forks a child process, which then calls `execv` to execute the *cline* program (recall Listing 1-7). Each function in the *exec* family does the following:

- Replaces the image of the process that calls an *exec* function with a new process image. This is described as *overlaying* one process image with another.
- The new process, in this case *cline*, runs with the same *pid* as the original process, in this case *execing*.

The *cline* program expects command-line arguments, which are supplied in a `NULL`-terminated array of strings; the *cline* program simply prints the arguments to the standard output and then exits.

In the *execing* program, the call to `fork` follows the usual pattern except that parent process has nothing left to do if the `fork` succeeds; the parent terminates by returning from `main`. By contrast, the child process invokes `execv` with two arguments:

- The first argument is the path to the executable as a string, in this case `".\cline"`.
- The second argument is an array of strings, including (by tradition) the name of the executable as the first element in this array. A `NULL` marks the end of the string array.

The execv function returns -1 to signal an error—and otherwise does *not* return. Instead, the overlayed process image is used to execute the overlay program, in this case *cline*. Accordingly, the last printf statement in the *execing* program

```
printf("This should not print!\n");
```

does *not* execute. Only the newly executed *cline* program runs to completion: the process image for the forked child indeed has been overlaid.

There is a short experiment that can confirm the overlay in the *execing* program:

- Immediately after the successful fork of the child process, print the child's *pid* value, which can be obtained with a call within the if block to the getpid function.
- Amend the *cline* program to print its own *pid*, again using the library function getpid.

The two printed *pid* values should be the same, thereby confirming that the *execed* program *cline* is executing under the forked child's *pid*. The code available on GitHub includes this experiment.

7.3.1. Process Id and Exit Status

The next program reviews the forking API, in particular the *pid* and *ppid* values for a child process, but also focuses on the information available about how a child process terminates. The *exit status* of a forked process is available, with convenient macros for extracting this status information. These macros belong to C's *waiting* API, whose principal functions are wait (one argument for ease of use) and waitpid (three or four arguments for fine-grained control). The example introduces the waitpid function.

In production-grade multiprocessing programs, logic likely depends on the state of the constituent processes, including information about how a given process terminates. For example, a multiprocessing web server such as Nginx needs to track whether the *master process* and the *worker processes* (request handlers) are still alive and, if not, the exit status of a terminated process. The multiprocessing examples so far have ignored the exit status of a child process. The forthcoming *exiting* example focuses on the child's exit status and how the parent can get this status.

Listing 7-4. Exit status

```
#include <unistd.h>     /* symbolic constants */
#include <stdio.h>      /* printf, etc. */
#include <sys/wait.h>   /* waiting on process termination */
#include <stdlib.h>     /* utilities */

void main() {
  int status;         /* parent captures child's status here */
  int cret = 0xaa11bb22;  /* child returns this value */

  pid_t cpid = fork();    /* spawn the child process */

  if (0 == cpid) {        /* fork() returns 0 to the child */
    printf("Child's pid and ppid: %i  %i\n", getpid(),
      getppid()); /* 2614 2613 */
    printf("Child returns %x explicitly.\n", cret);
    _exit(cret);               /* return an arbitrary value */
  }
  else { /* fork() returns new pid to the parent process */
    printf("Parent's pid: %i\n", getpid());   /* 2613 */
    printf("Waiting for child to exit\n");
```

```
    if (-1 != waitpid(cpid, &status, 0)) { /* wait for child
                                            to exit, store its
                                            status */
      if (WIFEXITED(status))
      printf("Normal exit with %x\n", WEXITSTATUS(status));
      /** 22 **/
      else if (WIFSIGNALED(status))
        printf("Signaled with %x\n", WTERMSIG(status));
      else if (WIFSTOPPED(status))
        printf("Stopped with %x\n", WSTOPSIG(status));
        /* stop pauses the process */
      else
      puts("peculiar...");
    }
    exit(0); /* parent exits with normal termination */
  }
}
```

In the *exiting* program (see Listing 7-4), one process forks another in the by-now-familiar way. The parent waits for the child with a call to `waitpid`, which expects three arguments:

- The first argument is the *pid* of the process on which to wait, in this case the child.
- The second argument points to an `int` variable where the child's exit or comparable status is stored.
- The last argument consists of additional options, for instance, `WNOHANG` for *return at once if no child has exited*.

The `wait(NULL)` call used earlier is shorthand for

```
waitpid(-1, NULL, NULL);
```

The first argument to `waitpid` (-1) means, in effect, *any child of mine*; the second argument is `NULL` instead of a pointer to an `int` variable to store the child's exit status; and the third argument is `NULL` for *no flags*.

For the child process, there are various possibilities that a *waiter* such as the parent needs to consider. Three of these possibilities are considered in the *exiting* program:

- The child exits normally, with a nonnegative return value.
- The child receives a signal such as `SIGKILL` (terminate immediately), which cannot be ignored, or `SIGTERM` (please terminate immediately), which *can* be ignored.
- The child receives a `SIGSTOP` (stop executing: pause) signal, which cannot be ignored.

In this example, the child exits normally with a call to `_exit`. The `WEXITSTATUS` macro returns the low-order 8 bits of the child's 32-bit explicitly returned value, `0xaa11bb22` in hex. The macro thus extracts `22`.

The *exiting* program also confirms that a child's *ppid* is the same as its parent's *pid*. In a sample run, this value was 2613, and the child's *pid* was 2614. These values are not guaranteed to be consecutive, but it is a common pattern: the child's *pid* is one greater than the parent's.

7.4. Interprocess Communication Through Shared Memory

Although every process has its own address space, which ensures that processes do not share memory locations by default, processes can share memory. A standard library provides the appropriate functions. Shared memory is, like pipes, a mechanism for interprocess communication. A code example with two processes explores the details.

There are two separate libraries and APIs for shared memory: the legacy System V library and API, and the more recent POSIX pair. These APIs should never be mixed in a single application, however. The POSIX pair is still in development and dependent upon the version of the operating system kernel, which impacts code portability. By default, the POSIX API implements shared memory as a memory-mapped file: for a shared memory segment, the system maintains a backing file with corresponding contents. Shared memory under POSIX can be configured without a backing file, but this may impact portability. My example uses the POSIX API with a backing file, which combines the benefits of memory access (speed) and file storage (persistence).

The shared memory example has two programs, named *memwriter* and *memreader*, and uses a semaphore to coordinate their access to the shared memory. Whenever shared memory comes into the picture with a writer, so does the risk of a memory-based race condition with indeterminate results; hence, the semaphore is used to coordinate (synchronize) access to the shared memory so that the writer and the reader operations do not overlap.

The *memwriter* program, which creates the shared memory segment, should be started first in its own terminal. The *memreader* program then can be started (within a dozen seconds) in its own terminal. The output from the memreader is

```
This is the way the world ends...
```

Here is a review of how semaphores work as a synchronization mechanism. A general semaphore also is called a counting semaphore, as it has a value (typically initialized to zero) that can be incremented. Consider a shop that rents bicycles, with a hundred of them in stock, with a program that clerks use to do the rentals. Every time a bike is rented, the semaphore is incremented by one; when a bike is returned, the semaphore is decremented by one. Rentals can continue until the value hits 100 but then must halt until at least one bike is returned, thereby decrementing the semaphore to 99.

A binary semaphore is a special case requiring only two values, which are traditionally 0 and 1. In this situation, a semaphore acts as a mutex: a mutual exclusion construct. The shared memory example uses a semaphore as a mutex. When the semaphore's value is 0, the *memwriter* alone can access the shared memory. After writing, this process increments the semaphore's value, thereby allowing the *memreader* to read the shared memory.

Listing 7-5. The `memwriter` program

```
/** Compilation: gcc -o memwriter memwriter.c -lrt
-lpthread **/
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <semaphore.h>
#include <string.h>
#include "shmem.h"

void report_and_exit(const char* msg) {
  perror(msg);
  exit(-1);
}

int main() {
  int fd = shm_open(BackingFile,      /* name from smem.h */
                    O_RDWR | O_CREAT, /* read/write, create if
                                         needed */
                    AccessPerms);     /* access permissions
                                         (0644) */
```

```
if (fd < 0) report_and_exit("Can't open shared mem segment...");

ftruncate(fd, ByteSize); /* get the bytes */

caddr_t memptr = mmap(NULL,       /* let system pick where to
                                  put segment */
                      ByteSize,   /* how many bytes */
                      PROT_READ | PROT_WRITE, /* access
                                              protections */
                      MAP_SHARED, /* mapping visible to other
                                  processes */
                      fd,         /* file descriptor */
                      0);         /* offset: start at
                                  1st byte */
if ((caddr_t) -1  == memptr) report_and_exit("Can't get
segment...");

fprintf(stderr, "shared mem address: %p [0..%d]\n", memptr,
ByteSize - 1);
fprintf(stderr, "backing file:       /dev/shm%s\n",
BackingFile );

/* semaphore code to lock the shared mem */
sem_t* semptr = sem_open(SemaphoreName, /* name */
                         O_CREAT,       /* create the
                                        semaphore */
                         AccessPerms,   /* protection
                                        perms */
                         0);            /* initial value */
if (semptr == (void*) -1) report_and_exit("sem_open");

strcpy(memptr, MemContents); /* copy some ASCII bytes to the
                             segment */
```

```
  /* increment the semaphore so that memreader can read */
  if (sem_post(semptr) < 0) report_and_exit("sem_post");

  sleep(12); /* give reader a chance */

  /* clean up */
  munmap(memptr, ByteSize); /* unmap the storage */
  close(fd);
  sem_close(semptr);
  shm_unlink(BackingFile); /* unlink from the backing file */
  return 0;
}
```

The *memwriter* and *memreader* programs communicate through shared memory as follows. The *memwriter* program (see Listing 7-5) calls the `shm_open` library function to get a file descriptor for the backing file that the system coordinates with the shared memory. At this point, no memory has been allocated. The subsequent call to the misleadingly named function `ftruncate`

```
ftruncate(fd, ByteSize); /* get the bytes */
```

allocates `ByteSize` bytes, in this case, a modest 512 bytes. The *memwriter* and *memreader* programs access the shared memory only, not the backing file. The system is responsible for synchronizing the shared memory and the backing file.

The *memwriter* then calls the `mmap` library function

```
caddr_t memptr = mmap(NULL,       /* let system pick where to
                                     put segment */
                      ByteSize,   /* how many bytes */
                      PROT_READ | PROT_WRITE, /* access
                                                 protections */
```

```
                    MAP_SHARED, /* mapping visible to other
                                processes */
                    fd,         /* file descriptor */
                    0);         /* offset: start at
                                1st byte */
```

to get a pointer to the shared memory. (The *memreader* makes a similar call.) The pointer type `caddr_t` starts with a `c` for `calloc`, which initializes dynamically allocated storage to zeros. The *memwriter* uses the `memptr` for the later write operation, which uses the library `strcpy` function. At this point, the *memwriter* is ready for writing, but it first creates a semaphore to ensure exclusive access to the shared memory.

If the call to `sem_open` for the semaphore's creation succeeds

```
sem_t* semptr = sem_open(SemaphoreName, /* name */
                         O_CREAT,       /* create the
                                        semaphore */
                         AccessPerms,   /* protection perms */
                         0);            /* initial value */
```

then the writing can proceed. The `SemaphoreName` (any unique nonempty name will do) identifies the semaphore in both the *memwriter* and the *memreader*. The initial value of zero gives the semaphore's creator (in this case, the *memwriter*) the right to proceed (in this case, to the write operation).

After writing, the memwriter increments the semaphore value to 1:

```
if (sem_post(semptr) < 0)
```

with a call to the `sem_post` library function. Incrementing the semaphore releases the mutex lock and enables the *memreader* to perform its read operation. For good measure, the *memwriter* also unmaps the shared memory from the *memwriter* address space:

```
munmap(memptr, ByteSize); /* unmap the storage *
```

This bars the *memwriter* from further access to the shared memory.

Listing 7-6. The memreader program

```
/** Compilation: gcc -o memreader memreader.c -lrt
-lpthread **/
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <semaphore.h>
#include <string.h>
#include "shmem.h"

void report_and_exit(const char* msg) {
  perror(msg);
  exit(-1);
}

int main() {
  int fd = shm_open(BackingFile, O_RDWR, AccessPerms);
  /* empty to begin */
  if (fd < 0) report_and_exit("Can't get file descriptor...");

  /* get a pointer to memory */
  caddr_t memptr = mmap(NULL,       /* let system pick where to
                                    put segment */
                        ByteSize,   /* how many bytes */
                        PROT_READ | PROT_WRITE, /* access
                                                protections */
                        MAP_SHARED, /* mapping visible to other
                                    processes */
```

```
                          fd,          /* file descriptor */
                          0);          /* offset: start at
                                       1st byte */
  if ((caddr_t) -1 == memptr) report_and_exit("Can't access
  segment...");

  /* create a semaphore for mutual exclusion */
  sem_t* semptr = sem_open(SemaphoreName, /* name */
                           O_CREAT,       /* create the
                                          semaphore */
                           AccessPerms,   /* protection
                                          perms */
                           0);            /* initial value */
  if (semptr == (void*) -1) report_and_exit("sem_open");

  /* use semaphore as a mutex (lock) by waiting for writer to
  increment it */
  if (!sem_wait(semptr)) { /* wait until semaphore != 0 */
    int i;
    for (i = 0; i < strlen(MemContents); i++)
      write(STDOUT_FILENO, memptr + i, 1); /* one byte at
                                           a time */
    sem_post(semptr);
  }

  /* cleanup */
  munmap(memptr, ByteSize);
  close(fd);
  sem_close(semptr);
  unlink(BackingFile);
  return 0;
}
```

In both the *memwriter* and *memreader* (see Listing 7-6) programs, the shared memory functions of primary interest are `shm_open` and `mmap`: on success, the first call returns a file descriptor for the backing file, which the second call then uses to get a pointer to the shared memory segment. The calls to `shm_open` are similar in the two programs except that the *memwriter* program creates the shared memory, whereas the *memreader* only accesses this already allocated memory:

```
int fd = shm_open(BackingFile, O_RDWR | O_CREAT, AccessPerms);
/* memwriter */
int fd = shm_open(BackingFile, O_RDWR,
AccessPerms);            /* memreader */
```

With a file descriptor in hand, the calls to `mmap` are the same:

```
caddr_t memptr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_
SHARED, fd, 0);
```

The first argument to `mmap` is `NULL`, which means that the system determines where to allocate the memory in virtual address space. It is possible (but tricky) to specify an address instead. The `MAP_SHARED` flag indicates that the allocated memory is shareable among processes, and the last argument (in this case, zero) means that the offset for the shared memory should be the first byte. The `size` argument specifies the number of bytes to be allocated (in this case, 512), and the protection argument indicates that the shared memory can be written and read.

When the *memwriter* program executes successfully, the system creates and maintains the backing file; on my system, the file is */dev/shm/shMemEx*, with *shMemEx* as my name (given in the header file *shmem.h*) for the shared storage. In the current version of the *memwriter* and *memreader* programs, the statement

```
shm_unlink(BackingFile); /* removes backing file */
```

removes the backing file. If the unlink statement is omitted, then the backing file persists after the program terminates.

The *memreader*, like the *memwriter*, accesses the semaphore through its name in a call to `sem_open`. But the *memreader* then goes into a wait state until the *memwriter* increments the semaphore, whose initial value is 0:

```
if (!sem_wait(semptr)) { /* wait until semaphore != 0 */
```

Once the wait is over, the *memreader* reads the ASCII bytes from the shared memory, cleans up, and terminates.

The shared memory API includes operations explicitly to synchronize the shared memory segment and the backing file. These operations have been omitted from the example to reduce clutter and keep the focus on the memory-sharing and semaphore code.

The *memwriter* and *memreader* programs are likely to execute without inducing a race condition even if the semaphore code is removed: the *memwriter* creates the shared memory segment and writes immediately to it; the *memreader* cannot even access the shared memory until this has been created. However, best practice requires that shared memory access is synchronized whenever a write operation is in the mix, and the semaphore API is important enough to be highlighted in a code example.

7.5. Interprocess Communication Through File Locking

Programmers are all too familiar with file access, including the many pitfalls (nonexistent files, bad file permissions, and so on) that beset the use of files in programs. Nonetheless, shared files may be the most basic mechanism for interprocess communication. Consider the relatively

simple case in which one process (*producer*) creates and writes to a file and another process (*consumer*) reads from this same file:

```
         writes  +-----------+  reads
producer-------->| disk file |<-------consumer
                 +-----------+
```

The obvious challenge in using a shared file is that a race condition might arise: the *producer* and the *consumer* might access the file at exactly the same time, thereby making the outcome indeterminate. To avoid a race condition, the file must be locked in a way that prevents a conflict between a write operation and any another operation, whether a read or a write. The locking API in the standard system library can be summarized as follows:

- A producer should gain an exclusive lock on the file before writing to the file. An exclusive lock can be held by one process at most, which rules out a race condition because no other process can access the file until the lock is released. (It is possible to lock only part of a file.)
- A consumer should gain at least a shared lock on the file before reading from the file. Multiple readers can hold a shared lock at the same time, but no writer can access a file when even a single reader holds a shared lock. A shared lock promotes efficiency. If one process is just reading a file and not changing its contents, there is no reason to prevent other processes from doing the same. Writing, however, clearly demands exclusive access to a file, as a whole or just in part.

The standard I/O library includes a utility function named `fcntl` that can be used to inspect and manipulate both exclusive and shared locks on a file. The function works through the by-now-familiar file descriptor, a nonnegative integer value that, within a process, identifies a file. (Recall that different file descriptors in different processes may identify the same physical file.) For file locking, Linux provides the library function `flock`, which is a thin wrapper around `fcntl`. The code examples use the `fcntl` function to expose API details.

Listing 7-7. The producer program

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define FileName "data.dat"
#define DataString "Now is the winter of our discontent\nMade
glorious summer by this sun of York\n"

void report_and_exit(const char* msg) {
  perror(msg);
  exit(-1); /* EXIT_FAILURE */
}

int main() {
  struct flock lock;
  lock.l_type = F_WRLCK;    /* read/write (exclusive versus
                               shared) lock */
  lock.l_whence = SEEK_SET; /* base for seek offsets */
  lock.l_start = 0;         /* 1st byte in file */
  lock.l_len = 0;           /* 0 here means 'until EOF' */
  lock.l_pid = getpid();    /* process id */
```

```
  int fd; /* file descriptor to identify a file within a
          process */
  if ((fd = open(FileName, O_RDWR | O_CREAT, 0666)) < 0)  /* -1
  signals an error */
    report_and_exit("open failed...");

  if (fcntl(fd, F_SETLK, &lock) < 0) /** F_SETLK doesn't block,
                                      F_SETLKW does **/
    report_and_exit("fcntl failed to get lock...");
  else {
    write(fd, DataString, strlen(DataString)); /* populate
                                                  data file */
    fprintf(stderr, "Process %d has written to data file...\n",
    lock.l_pid);
  }

  /* Now release the lock explicitly. */
  lock.l_type = F_UNLCK;
  if (fcntl(fd, F_SETLK, &lock) < 0)
    report_and_exit("explicit unlocking failed...");

  close(fd); /* close the file: would unlock if needed */
  return 0;  /* terminating the process would unlock as well */
}
```

The main steps in the *producer* program (see Listing 7-7) can be summarized as follows. The program declares a variable of type `struct flock`, which represents a lock, and initializes the structure's five fields. The first initialization

```
lock.l_type = F_WRLCK; /* exclusive lock */
```

makes the lock an exclusive (read-write) rather than a shared (read-only) lock. If the *producer* gains the lock, then no other process will be able to write or read the file until the *producer* releases the lock, either explicitly with the appropriate call to `fcntl` or implicitly by closing the file. (When the process terminates, any opened files would be closed automatically, thereby releasing the lock.) The program then initializes the remaining fields. The chief effect is that the entire file is to be locked. However, the locking API allows only designated bytes to be locked. For example, if the file contains multiple text records, then a single record (or even part of a record) could be locked and the rest left unlocked.

The first call to `fcntl`

```
if (fcntl(fd, F_SETLK, &lock) < 0)
```

tries to lock the file exclusively, checking whether the call succeeded. In general, the `fcntl` function returns -1 (hence, less than zero) to indicate failure. The second argument `F_SETLK` means that the call to `fcntl` does not block: the function returns immediately, either granting the lock or indicating failure. If the flag `F_SETLKW` (the `W` at the end is for *wait*) were used instead, the call to `fcntl` would block until gaining the lock was possible. In the calls to `fcntl`, the first argument `fd` is the file descriptor, the second argument specifies the action to be taken (in this case, `F_SETLK` for setting the lock), and the third argument is the address of the lock structure (in this case, `&lock`).

If the *producer* gains the lock, the program writes two text records to the file. After writing to the file, the *producer* changes the lock structure's `l_type` field to the unlock value:

```
lock.l_type = F_UNLCK;
```

and calls `fcntl` to perform the unlocking operation. The program finishes up by closing the file and exiting.

Listing 7-8. The consumer program

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

#define FileName "data.dat"

void report_and_exit(const char* msg) {
  perror(msg);
  exit(-1); /* EXIT_FAILURE */
}

int main() {
  struct flock lock;
  lock.l_type = F_WRLCK;    /* read/write (exclusive) lock */
  lock.l_whence = SEEK_SET; /* base for seek offsets */
  lock.l_start = 0;         /* 1st byte in file */
  lock.l_len = 0;           /* 0 here means 'until EOF' */
  lock.l_pid = getpid();    /* process id */

  int fd; /* file descriptor to identify a file within a
  process */
  if ((fd = open(FileName, O_RDONLY)) < 0)  /* -1 signals an
                                               error */
    report_and_exit("open to read failed...");

  /* If the file is write-locked, we can't continue. */
  fcntl(fd, F_GETLK, &lock); /* sets lock.l_type to F_UNLCK if
                                no write lock */
  if (lock.l_type != F_UNLCK)
    report_and_exit("file is still write locked...");
```

```
  lock.l_type = F_RDLCK; /* prevents any writing during the
                            reading */
  if (fcntl(fd, F_SETLK, &lock) < 0)
    report_and_exit("can't get a read-only lock...");

  /* Read the bytes (they happen to be ASCII codes) one at a
  time. */
  int c; /* buffer for read bytes */
  while (read(fd, &c, 1) > 0)    /* 0 signals EOF */
    write(STDOUT_FILENO, &c, 1); /* write one byte to the
                                    standard output */

  /* Release the lock explicitly. */
  lock.l_type = F_UNLCK;
  if (fcntl(fd, F_SETLK, &lock) < 0)
    report_and_exit("explicit unlocking failed...");

  close(fd);
  return 0;
}
```

The *consumer* program (see Listing 7-8) is more complicated than necessary to highlight features of the locking API. In particular, the *consumer* program first checks whether the file is exclusively locked and only then tries to gain a shared lock. The relevant code is

```
lock.l_type = F_WRLCK;
...
fcntl(fd, F_GETLK, &lock); /* sets lock.l_type to F_UNLCK if no
                              write lock */
if (lock.l_type != F_UNLCK)
  report_and_exit("file is still write locked...");
```

The F_GETLK operation specified in the fcntl call checks for a lock, in this case, an exclusive lock given as F_WRLCK in the first statement earlier. If the specified lock does not exist, then the fcntl call automatically changes the lock type field to F_UNLCK to indicate this fact. If the file is exclusively locked, the *consumer* terminates. (A more robust version of the program might have the *consumer* sleep a bit and try again several times.)

If the file is not currently locked, then the *consumer* tries to gain a shared (read-only) lock (F_RDLCK). To shorten the program, the F_GETLK call to fcntl could be dropped because the F_RDLCK call would fail if a read-write lock already were held by some other process. Recall that a read-only lock does prevent any other process from writing to the file but allows other processes to read from the file. In short, a shared lock can be held by multiple processes. After gaining a shared lock, the *consumer* program reads the bytes one at a time from the file, prints the bytes to the standard output, releases the lock, closes the file, and terminates.

Here is the output from the two programs launched from the same terminal:

```
% ./producer
Process 29255 has written to data file...

% ./consumer
Now is the winter of our discontent
Made glorious summer by this sun of York
```

The data shared through this interprocess communication is text: two lines from Shakespeare's play Richard III. Yet the shared file's contents could be voluminous, arbitrary bytes (e.g., a digitized movie), which makes file sharing an impressively flexible mechanism. The downside is that file access is relatively slow, whether the access involves reading or writing. As always, programming comes with trade-offs.

7.6. Interprocess Communication Through Message Queues

Earlier code examples highlighted pipes, both named and unnamed. Pipes of either type have strict FIFO behavior: the first byte written is the first byte read, the second byte written is the second byte read, and so forth. Message queues can behave in the same way but are flexible enough that byte chunks can be retrieved out of FIFO order.

As the name suggests, a message queue is a sequence of messages, each of which has two parts:

- The payload, which is an array of bytes (`char`).
- A type, given as a positive integer value; types categorize messages for flexible retrieval.

Consider the following depiction of a message queue, with each message labeled with an integer type:

```
          +-+    +-+    +-+    +-+
sender--->|3|--->|2|--->|2|--->|1|--->receiver
          +-+    +-+    +-+    +-+
```

Of the four messages shown, the one labeled 1 is at the front, that is, closest to the receiver. Next come two messages with label 2, and finally, a message labeled 3 at the back. If strict FIFO behavior were in play, then the messages would be received in the order 1-2-2-3. However, the message queue allows other retrieval orders. For example, the messages could be retrieved by the receiver in the order 3-2-1-2.

The mqueue example consists of two programs: the *sender* that writes to the message queue and the *receiver* that reads from this queue. Both programs include the header file *queue.h* shown in Listing 7-9.

Listing 7-9. The header file queue.h

```
#define ProjectId 123
#define PathName  "queue.h" /* any existing, accessible file
                            would do */
#define MsgLen    4
#define MsgCount  6

typedef struct {
  long type;                 /* must be of type long */
  char payload[MsgLen + 1];  /* bytes in the message */
} queuedMessage;
```

The header file defines a structure type named `queuedMessage`, with payload (byte array) and type (integer) fields. This file also defines symbolic constants (the `#define` directives), the first two of which are used to generate a key that, in turn, is used to get a message queue ID. The `ProjectId` can be any positive integer value, and the `PathName` must be of an existing, accessible file—in this case, the file *queue.h*. The setup statements in both the *sender* and the *receiver* programs are

```
key_t key = ftok(PathName, ProjectId);   /* generate key */
int qid = msgget(key, 0666 | IPC_CREAT); /* use key to get
queue id */
```

The ID `qid` is, in effect, the counterpart of a file descriptor for message queues.

Listing 7-10. The message sender program

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdlib.h>
```

```
#include <string.h>
#include "queue.h"

void report_and_exit(const char* msg) {
  perror(msg);
  exit(-1); /* EXIT_FAILURE */
}

int main() {
  key_t key = ftok(PathName, ProjectId);
  if (key < 0) report_and_exit("couldn't get key...");

  int qid = msgget(key, 0666 | IPC_CREAT);
  if (qid < 0) report_and_exit("couldn't get queue id...");

  char* payloads[] = {"msg1", "msg2", "msg3", "msg4", "msg5",
  "msg6"};
  int types[] = {1, 1, 2, 2, 3, 3}; /* each must be > 0 */
  int i;
  for (i = 0; i < MsgCount; i++) {
    /* build the message */
    queuedMessage msg;
    msg.type = types[i];
    strcpy(msg.payload, payloads[i]);

    /* send the message */
    msgsnd(qid, &msg, MsgLen + 1, IPC_NOWAIT); /* don't
                                                  block */
    printf("%s sent as type %i\n", msg.payload, (int)
    msg.type);
  }
  return 0;
}
```

The preceding *sender* program sends out six messages, two each of a specified type: the first messages are of type 1, the next two of type 2, and the last two of type 3. The sending statement

```
msgsnd(qid, &msg, MsgLen + 1, IPC_NOWAIT);
```

is configured to be nonblocking (the flag `IPC_NOWAIT`) because the messages are so small. The only danger is that a full queue, unlikely in this example, would result in a sending failure. The following *receiver* program also receives messages using the `IPC_NOWAIT` flag.

Listing 7-11. The message receiver program

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdlib.h>
#include "queue.h"

void report_and_exit(const char* msg) {
  perror(msg);
  exit(-1); /* EXIT_FAILURE */
}

int main() {
  key_t key= ftok(PathName, ProjectId); /* key to identify the
                                           queue */
  if (key < 0) report_and_exit("key not gotten...");

  int qid = msgget(key, 0666 | IPC_CREAT); /* access if created
                                              already */
  if (qid < 0) report_and_exit("no access to queue...");

  int types[] = {3, 1, 2, 1, 3, 2}; /* different than in
                                       sender */
```

```
  int i;
  for (i = 0; i < MsgCount; i++) {
    queuedMessage msg; /* defined in queue.h */
    if (msgrcv(qid, &msg, MsgLen + 1, types[i], MSG_NOERROR |
    IPC_NOWAIT) < 0)
      puts("msgrcv trouble...");
    printf("%s received as type %i\n", msg.payload, (int)
    msg.type);
  }

  /** remove the queue **/
  if (msgctl(qid, IPC_RMID, NULL) < 0)  /* NULL = 'no flags' */
    report_and_exit("trouble removing queue...");

  return 0;
}
```

The *receiver* program does not create the message queue, although the API suggests as much. In the *receiver*, the call

```
int qid = msgget(key, 0666 | IPC_CREAT);
```

is misleading because of the `IPC_CREAT` flag, but this flag really means create if needed, otherwise access. The *sender* program calls `msgsnd` to send messages, whereas the *receiver* calls `msgrcv` to retrieve them. In this example, the *sender* sends the messages in the order 1-1-2-2-3-3, but the *receiver* then retrieves them in the order 3-1-2-1-3-2, showing that message queues are not bound to strict FIFO behavior:

```
% ./sender
msg1 sent as type 1
msg2 sent as type 1
msg3 sent as type 2
msg4 sent as type 2
```

```
msg5 sent as type 3
msg6 sent as type 3

% ./receiver
msg5 received as type 3
msg1 received as type 1
msg3 received as type 2
msg2 received as type 1
msg6 received as type 3
msg4 received as type 2
```

The preceding output shows that the *sender* and the *receiver* can be launched from the same terminal. The output also shows that the message queue persists even after the *sender* process creates the queue, writes to it, and exits. The queue goes away only after the *receiver* process explicitly removes the queue with the call to `msgctl`:

```
if (msgctl(qid, IPC_RMID, NULL) < 0) /* remove queue */
```

7.7. Multithreading

Recall that a *multithreaded* process has multiple threads (sequences) of executable instructions, which can be executed concurrently and, on a multiprocessor machine, in parallel. Multithreading, like multiprocessing, is a way to multitask. Multithreading has the upside of efficiency because thread-level context switches are quite fast but the downside of challenging the programmer with the twin perils of race conditions and deadlock. Code examples go into detail. To begin, an example of *pthread* (the standard thread library) basics should be helpful.

Listing 7-12. A first multithreaded example

```
/* compilation: gcc -o greet greet.c -lpthread */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define ThreadCount 4

void* greet(void* my_id) { /* void* is 8 bytes on a 64-bit
machine */
  unsigned i, n = ThreadCount;
  for (i = 0; i < n; i++) {
    printf("from thread %ld...\n", (unsigned long) my_id);
    sleep(rand() % 3);
  }
  return 0;
} /* implicit call to pthread_exit(NULL) */

void main() {
  pthread_t threads[ThreadCount];
  unsigned long i;
  for (i = 0; i < ThreadCount; i++) {
    /* four args: pointer to pthread_t instance, attributes,
    start function,
       and argument passed to start function */
    int flag = pthread_create(threads + i,  /* 0 on success */
                              NULL,
                              greet,
                              (void*) i + 1);
    if (flag < 0) {
      perror(NULL);
```

```
      exit(-1);
    }
  }
  puts("main exiting...");
  pthread_exit(NULL); /* allows other threads to continue
                         execution */
}
```

The *multiT* program (see Listing 7-12) has five threads in all: the *main thread*, which executes the body of `main`, and four additional threads that `main` creates through calls to the library function `pthread_create`. The `pthread_create` function takes four arguments:

- The first argument is a pointer to a `pthread_t` instance, in this case an element in the `threads` array.
- The second argument specifies thread attributes. A value of `NULL` indicates that the default attributes should be used.
- The third argument is a pointer to the thread's *start function*, which the thread executes once the operating system starts the thread. A created thread automatically terminates when it returns from its start function. The start function can call other functions and do whatever else comes naturally to functions.
- The fourth and last argument specifies what should be passed, as an argument, to the start function. In this case, the argument passed to the `greet` start function will be one of the values 1, 2, 3, and 4, which identify each of the created threads. The argument passed to the start function is always of the generic type `void*`, and `NULL` for *no argument* can be used.

All four of the created threads execute the same code, the body of the `greet` function, but no race condition arises. Arguments passed to a function, and local (`auto` or `register`) variables within the function, are thereby *thread-safe* because each thread gets its own copies. If a variable is neither `extern` nor `static`, then it represents a thread-safe memory location.

The `pthread_create` function returns -1 to signal an error and 0 to signal success. A successfully created thread is ready to be scheduled for execution on a processor.

At the end of `main`, the *multiT* program calls the library function `pthread_exit` with an argument of `NULL`. The address of an `int` exit-status variable also could be used as the argument. This call from `main` allows other threads to continue executing. On a sample run, for instance, the output began:

```
from thread 2...
from thread 4...
main exiting...
from thread 3...
...
```

The *order* of thread execution is indeterminate. Once the threads are created, the operating system takes over the scheduling, using whatever algorithm the host system employs. A `pthread` instance is a *native thread* under operating system control. By contrast, a *green thread* is under the control of a virtual machine. For example, early implementations of Java (before JDK 1.4) were required to support only green threads. If the *multiT* program is run several times, the output is likely to differ each time.

WHAT'S POSIX?

The Portable Operating System Interface is a family of standards from the IEEE Computer Society meant to encourage compatibility among operating systems. The multithreading examples use *pthreads*, where the *p* stands for POSIX.

7.7.1. A Thread-Based Race Condition

The next code example illustrates a race condition in a multithreaded program. The program later introduces a mechanism for coordinating thread execution, thereby preventing this race condition. A short depiction of a race condition follows.

Suppose that there is a `static` variable named `n`, which is initialized to 1 and updated as follows:

```
n += rand();  /* add a pseudo random value to n */
```

The assignment operator += makes it clear that *two* operations are involved: an addition followed by an assignment. Suppose that this same statement belongs to two separate threads of execution, *T1* and *T2*, each of which accesses the same variable `n`. For emphasis, assume that each thread executes literally at the same time on a multiprocessor machine. Here is one possible scenario, where each of the numbered items represents one tick of the system clock:

1. Thread *T1* gets 123 from its call to `rand()` and performs the addition. Assume that the sum of the two numbers 123 + 1 = 124 is stored on the stack. Call this storage location *temp1*, which now holds 124.

2. Thread *T2* gets 987 from its call to `rand()` and performs the addition. The sum 988 is stored in *temp2*, also on the stack.
3. Thread *T2* performs the assignment, using the value from *temp2*: the value of `n` is updated to 988.
4. Thread *T1* performs the assignment, using the value from *temp1*: the value of `n` changes to 124.

It is clear that improper interleaving of machine-level instructions has taken place. Thread *T2* does its addition and assignment *without interruption*, which is the correct way to perform the two operations. By contrast, thread *T1* does its addition, is delayed two ticks of the clock, and then finishes up with an assignment. By coming in last, thread *T1 wins* the race: the final value of variable `n`, 124, reflects only what thread *T1* did, and what thread *T2* did is effectively lost.

The two operations, the addition and then the assignment, make up a *critical section*, a sequence of operations that must be executed in a single-threaded, uninterrupted manner: if one thread starts its addition, no other thread should access variable `n` until this first thread completes its work with an assignment. The code segment at present does not enforce single-threaded or *thread-safe* execution of the

```
n += rand(); /* addition then assignment */
```

critical section. The outcome is, therefore, indeterminate and unpredictable.

7.7.2. The Miser/Spendthrift Race Condition

The forthcoming *miserSpend* program encourages a race condition by having two threads concurrently update a shared memory location, in this case the single `static` variable named `account`, which represents a

shared bank account: both threads access the same account. A memory-based race condition requires contention for a *shared* memory location. Of course, the `account` variable could be `extern` rather than `static` without changing the program's behavior.

The miser (saver) and the spendthrift (spender) are implemented as two separate threads, each with uncoordinated access to the `account`. To highlight the race condition, the miser and the spendthrift update the balance the *same* number of times, given as a command-line argument. Here is a depiction of what goes on in the *miserSpend* program:

```
      increment  +---------+  decrement
miser----------->| account |<-----------spendthrift  ## updates are done many times
                 +---------+
```

On a multiprocessor machine, the miser and the spendthrift can execute in a truly parallel fashion. Because access to the `account` is uncoordinated, a race condition ensues, and the final value of `account` is indeterminate. Indeed, if the two threads increment and decrement a sufficient number of times (e.g., ten million apiece), it becomes highly unlikely that the `account` will have zero as its value at the end, or that the `account` will have a repeated value over multiple runs.

As in the earlier multithreading example, the *main thread* starts the other threads, but the *main thread* now must wait for the miser and the spendthrift threads to terminate. For the program to illustrate the race condition, the *main thread* must be the last thread standing. The reason is that the *main thread* prints the final value of the `account` and must not do so prematurely, that is, before all of the updates have completed. Otherwise, the *main thread* might print the value of `account` when this value just happens to be zero. The *pthread* library has a function to enable the required waiting.

Listing 7-13. Creating, starting, and waiting on the miser and spendthrift threads

```
void report_and_die(const char* msg) {
  fprintf(stderr, "%s\n", msg);
   exit(-1);
}

void main(int argc, char* argv[]) {
  if (argc < 2) report_and_die("Usage: saveSpend <number of
  operations apiece>\n");
  int n = atoi(argv[1]); /** command-line argument conversion
                         to integer **/

  pthread_t miser, spendt;
  if (pthread_create(&miser, NULL, deposit, &n) < 0)
    report_and_die("pthread_create: miser");

  if (pthread_create(&spendt, NULL, withdraw, &n) < 0)
    report_and_die("pthread_create: spendt");

  pthread_join(miser,  NULL);  /* main thread waits on miser:
                               NULL for exit status */
  pthread_join(spendt, NULL);  /* main thread waits on spendt:
                               NULL for exit status */
  printf("The final account balance is: %10i\n", account);
}
```

The code for the *saveSpend* program is divided into two parts for readability. The first part (see Listing 7-13) has the *main thread* create and then start two other threads: the miser and the spendthrift threads. Each created thread is of type `pthread_t`, and the `pthread_create` function can be reviewed as follows:

- The first argument is the address of a `pthread_t` instance, in this case, of either the `miser` or the `spendt` variable.
- The second argument, `NULL`, indicates that default thread properties are to be used.
- The third argument is the address of the *start function*, either `deposit` (miser) or `withdraw` (spendthrift). Recall that each created thread terminates automatically when exiting its start function.
- The fourth argument is the address of the argument passed to the start function, in this case the address of integer variable `n`, which is the number of times that each started thread should update the `account`.

The *saveSpend* program introduces only one new function from the *pthread* API, `pthread_join`. The caller of the function, in this case `main`, thereby goes into a wait state until the thread identified in the first argument has exited. For review, the `main` function calls the `pthread_join` function twice:

```
pthread_join(miser,  NULL);  /* main thread waits on miser */
pthread_join(spendt, NULL);  /* main thread waits on spendt */
```

If the miser already has exited, the first call to `pthread_join` returns immediately; if not, the call returns when the miser does exit. The second argument to `pthread_join` can be used to get the exit status of the thread given as the second argument; in this case, the status is ignored with `NULL` as the second argument. The two calls to `pthread_join` ensure that the *main thread* prints the final balance—the balance *after* the other two threads have terminated.

Listing 7-14. The miser/spendthrift start functions

```
/** To compile: gcc -o saveSpend saveSpend.c -lpthread **/
#include<stdio.h>
#include<pthread.h>
#include<stdlib.h>

static int account = 0;  /** shared storage across the
                         threads **/

void update(int n) {
  account += n; /** critical section **/
}

void* deposit(void* n) {  /** miser code **/
  int limit = *(int*) n, i;
  for (i = 0; i < limit; i++) update(+1); /* add 1 to
                                          account */
  return NULL;
} /** thread terminates when exiting deposit **/

void* withdraw(void* n) { /** spendt code **/
  int limit = *(int*) n, i;
  for (i = 0; i < limit; i++) update(-1); /* subtract 1 from
                                          account */
  return NULL;
} /** thread terminates when exiting withdraw **/
```

The second part of the *saveSpend* program (see Listing 7-14) has the two start functions for the created threads: `deposit` (miser) and `withdraw` (spendthrift). Each of these functions takes, as its single argument, the number of times to perform an account update, implemented as the `update` function: the `deposit` function calls `update` with 1 as the argument, whereas the `withdraw` function calls `update` with -1 as the argument.

A command-line argument determines the number of deposits and withdrawals, and this number is the *same* for the miser and the spendthrift. The command-line argument should be sufficiently large to be interesting, that is, to confirm the race condition. If the number is too small (e.g., 100), then the miser might do its 100 deposits before the spendthrift does any withdrawals. The goal is to have each thread run long enough that there is improper interleaving of the arithmetic and assignment operations in the critical section, the body of the `update` function. With a command-line argument of 10M (million), the output from two consecutive runs was

```
The final account balance is:   203692
The final account balance is: -1800416
```

With a command-line argument of 10M, a result of zero is highly unlikely.

In the *saveSpend* program, the `account` is changed in only one place: the function `update`, which takes a single `int` argument and updates the `account` by this amount. For the *saveSpend* program to behave properly, the body of `update` function must execute in a single-threaded fashion. There are different ways to enforce this policy, and using a mutex to lock access to the `account` is one way. (Recall the earlier example of the *memwriter/memreader* in which a semaphore is used as a mutex.) In the current example, the mutex from the *pthread* library ensures single-threaded execution of a critical section—the body of the `update` function in which the `account` is either incremented or decremented.

Listing 7-15. Fixing the saveSpend program

```
static int account = 0;  /** shared storage across the
                         threads **/
static pthread_mutex_t lock; /* named lock for clarity */
```

```
void update(int n) {
  if (0 == pthread_mutex_lock(&lock)) {
    account += n;                    /** critical section **/
    pthread_mutex_unlock(&lock);
  }
}
```

The *saveSpend* program requires only a few changes to fix (see Listing 7-15):

- A `pthread_mutex_lock` variable named `lock` is added. There should be a *single* lock to ensure that the miser and the spendthrift contend for the *same* lock. The lock is `static` but could be `extern` as well.
- The lock is used in the `update` function. To update the `account`, a thread first must grab the lock, expressed here as the condition of the `if` clause. The `pthread_mutex_lock` function returns 0 to signal that the lock has been grabbed.
- Once a thread completes its update, the thread releases the lock so that another thread can try to grab it.

With these changes in place, the *saveSpend* program always prints 0 as the value of the `account` when the miser and spendthrift threads have terminated.

One more change is recommended in fixing the *saveSpend* program. After the miser and spendthrift threads terminate, the lock is no longer needed; hence, it should be destroyed. The function `main` could be changed as follows:

```
...
pthread_join(spendt, NULL);
pthread_mutex_destroy(&lock); /** added **/
```

A high-level summary of the `pthread_mutex` seems in order:

- To execute a locked critical section, a thread first must grab the lock. After finishing the execution of the critical section, a thread should release the lock to enable some other thread to grab the lock and thus to safeguard against deadlock.
- If multiple threads are contending for the lock, the implementation ensures that exactly one thread grabs it.
- In general, a mutex such as `pthread_mutex` does not guarantee fairness. For example, if two threads are contending for the lock, the mutex implementation does *not* guarantee that each thread will be successful half the time. However, the *saveSpend* program has other logic to ensure that the *miser* and the *spendthrift* threads execute the same number of times.

If the fixed *saveSpend* program is run with a sufficiently large loop count (e.g., 10,000,000) as the command-line argument, there will be noticeable slowdown compared to the original version of the program. There is a performance cost to mutual exclusion, which enforces single-threaded execution of a critical section; in this code example, the cost ensures that the *saveSpend* program runs correctly.

7.8. Deadlock in Multithreading

Deadlock can occur in either a multiprocessing or multithreading. In the multithreading context, deadlock can occur with just two threads: *T1* and *T2*. To access a shared resource *R*, either *T1* or *T2* must hold two locks (*L1* and *L2*) at the same time. Suppose the two threads try to access *R*, with *T1* managing to grab lock *L1* and *T2* managing to grab lock *L2*. Each thread

now waits indefinitely for the other to release its held lock—and deadlock results. Deadlock is usually inadvertent, of course, but the next code example tries to cause deadlock.

Listing 7-16. Deadlocking with threads

```
/** To compile: gcc -o deadlock deadlock.c -lpthread **/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

static pthread_mutex_t lock1, lock2; /** two locks protect the
                                     resource **/
static int resource = 0;             /** the resource **/

void grab_locks(const char* tname,
                const char* lock_name,
                const char* other_lock_name,
                pthread_mutex_t* lock,
                pthread_mutex_t* other_lock) {
  printf("%s trying to grab %s...\n", tname, lock_name);
  pthread_mutex_lock(lock);
  printf("%s grabbed %s\n", tname, lock_name);

  if (0 == strcmp(tname, "thread1")) usleep(100); /** fix
                                                  is in! **/
  printf("%s trying to grab %s...\n", tname, other_lock_name);
  pthread_mutex_lock(other_lock);
  printf("%s grabbed %s\n", tname, other_lock_name);
```

```
  resource = (0 == strcmp(tname, "thread1")) ? -9999 : 1111;
  pthread_mutex_unlock(other_lock);
  pthread_mutex_unlock(lock);
}

void* thread1() {
  grab_locks("thread1", "lock1", "lock2", &lock1, &lock2);
  /* lock1...lock2 */
  return NULL;
}

void* thread2() {
  grab_locks("thread2", "lock2", "lock1", &lock2, &lock1);
  /* lock2...lock1 */
  return NULL;
}

void main(){
    pthread_t t1, t2;
    pthread_create(&t1, NULL, thread1, NULL);     /* start
                                                  thread 1 */
    pthread_create(&t2, NULL, thread2, NULL);     /* start
                                                  thread 2 */
    pthread_join(t1, NULL);           /* wait for thread 1 */
    pthread_join(t2, NULL);           /* wait for thread 2 */
    printf("Number: %i (Unlikely to print...)\n", resource);
}
```

The *deadlock* program (see Listing 7-16) is likely but not certain to deadlock. Although deadlock is intended, the code still might execute in such a way that deadlock does *not* occur. On a sample run, however, the *deadlock* program produced this output:

```
thread1 trying to grab lock1...
thread1 grabbed lock1

thread2 trying to grab lock2...
thread2 grabbed lock2

thread2 trying to grab lock1...
thread1 trying to grab lock2...
```

A code analysis shows what happened.

The *main thread* creates two threads: `t1` and `t2`. Thread `t1` is created first, and the output confirms that `t1` starts executing first—although the order of execution is indeterminate. There are two locks, `lock1` and `lock2`, which protect `resource`, an `int` variable: thread `t1` tries to set this variable to -9999, whereas thread `t2` tries to set the variable to 1111. To set the variable, a thread must grab *both* locks.

Thread `t1` has `thread1` as its start function, and `t2` has `thread2` as its start function. In turn, these functions immediately call the `grab_locks` function, but with arguments in a different order. Recall that, in multithreading, each thread has its own copies of arguments and local variables.

Given the output shown previously, the concurrent execution of `grab_locks` can be summarized as follows:

1. Thread `t1` succeeds in grabbing `lock1` but fails in the attempt to grab `lock2`. After grabbing `lock1`, thread `t1` sleeps for 100 microseconds—time enough, as it turns out, for thread `t2` to grab `lock2`.
2. After grabbing `lock2`, thread `t2` tries to grab `lock1`, which thread `t1` already holds. At this point, `t1` holds `lock1` and `t2` holds `lock2`.

3. The last two statements in the `grab_locks` function release the locks. However, neither thread can proceed to the release code without first grabbing a lock that the other thread already holds—deadlock.

Why is deadlock not certain in the *deadlock* program? On my desktop machine, no deadlock results if the `usleep` call is removed from the `grab_locks` function. No deadlock results if the argument passed to `usleep` is sufficiently small. Even with the current `usleep` value of 100, it is possible that thread `t1` might grab both locks before thread `t2` even begins executing. It is also possible, on a multiprocessor machine, that thread `t2` is scheduled on a faster processor than is `t1`; as a result, `t2` grabs both locks before `t1` even begins executing. A thread that holds both locks can proceed to the release code: no deadlock occurs. The *deadlock* program tries to cause deadlock, but even this requires some experimentation by setting the amount of time that thread `t1` sleeps after grabbing the first lock.

The *deadlock* program tries to cause deadlock, but the real-world challenge is a concurrent program that, although designed not to deadlock, does so anyway. Modern database systems typically include at least a deadlock-detection module. In general, however, software systems neither detect, nor prevent, nor recover from deadlock. The burden thus falls on the programmer to write code that avoids deadlock.

7.9. SIMD Parallelism

The acronym SIMD was introduced in the mid-1960s as part of Flynn's taxonomy for parallel computing. SIMD stands for *single instruction, multiple data stream*. Flynn's taxonomy introduces other acronyms (e.g., MIMD for *multiple instruction, multiple data stream*) to describe additional approaches to parallel computation. This section focuses on SIMD parallelism.

Imagine integer values collected in array and a code segment that doubles the value of each element. A conventional approach would be to loop over the array and, one element at a time, double each value. In a SIMD architecture, a single instruction would execute on each element in parallel. The serial or iterative computation gives way to a one-step parallel computation, with a boost in performance that is both intuitive and compelling.

The concurrent programs examined so far become truly parallel programs without any programmer intervention. If a multiprocessing or multithreading program happens to execute on a multiprocessor machine (now the norm), then the operating system transforms the concurrent program into a parallel one by scheduling processes/threads onto different processors. SIMD parallelism differs in that *parallel instructions* come into play. SIMD is thus a type of *instruction-level parallelism*, which requires underlying architectural support.

The appeal of SIMD parallelism is obvious. Even everyday applications regularly iterate over arrays, performing the same operation on each element. For an array of size *N*, this iterative approach requires that *N* instructions be executed in sequence. Assume, for simplicity, that each instruction requires one tick of the system clock. In this scenario, doubling the array elements takes *N* ticks. If the doubling can be done in a single SIMD instruction, the time required drops from *N* ticks to roughly one tick, although there is nontrivial overhead to set up the parallel addition.

For some time, computers have had devices tailored for SIMD. A graphics processing unit (GPU) is a case in point; indeed, the acronym GP_GPU describes a GPU designed for *general purpose* rather than just graphics-specific processing. There are various C libraries and entire frameworks devoted to putting such devices to use in SIMD processing. This section goes another way, focusing instead on how the standard C compilers are now able to use native SIMD instructions, in particular on modern Intel and AMD machines. (ARM Neon machines likewise support SIMD.)

In the late 1990s, Intel released the P5 (*P* for *Pentium*) line of microprocessors, which support the MMX instruction set, a first step toward SIMD parallelism. The MM registers associated with this instruction set, and the instruction set itself, soon gave way to SSE (Streaming SIMD Extensions) in different versions (e.g., SSE2 and SSE4). The XMM registers of SSE are 128 bits in size and small in number—only eight to begin but later sixteen. The SIMD architecture and instruction set have continued to evolve. For example, the XMM registers (128 bits) now have siblings: YMM registers (256 bits) and ZMM registers (512 bits).

Listing 7-17. A SIMD program in C

```
#include <stdio.h>

#define Length 8
typedef double doubleV8 __attribute__ ((vector_size (Length *
sizeof(double)))); /** critical **/

void main() {
  doubleV8 dataV1 = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8};
  /* no square brackets on dataV1 */
  doubleV8 dataV2 = {4.4, 6.6, 1.1, 3.3, 5.5, 2.2, 3.3, 5.5};
  /* no square brackets on dataV2 */

  doubleV8 add = dataV1 + dataV2;
  doubleV8 mul = dataV1 * dataV2;
  doubleV8 div = dataV1 / dataV2;

  int i;
  for (i = 0; i < Length; i++)
    printf("%f ", add[i]); /* 5.500000 8.800000 4.400000
    7.700000 11.000000 8.800000 11.000000 14.300000 */
```

```
  putchar('\n');
  for (i = 0; i < Length; i++)
    printf("%f ", mul[i]); /* 4.840000 14.520000 3.630000
                              14.520000 30.250000 14.520000
                              25.410000 48.400000 */

  putchar('\n');
  for (i = 0; i < Length; i++)
    printf("%f ", div[i]); /* 0.250000 0.333333 3.000000
                              1.333333 1.000000 3.000000 2.333333
                              1.600000 */

  putchar('\n');
}
```

The *simd* program (see Listing 7-17) has a `typedef` that triggers the C compiler to use native SIMD instructions and the supporting architectural components, in particular SIMD registers. The `typedef` makes `doubleV8` an alias for a `double` vector by using a special attribute:

```
__attribute__ ((vector_size (Length * sizeof(double))))
```

The `attribute` specifier has two underscores in front and in back. The specified attribute is `vector_size`, whose value is `Length` (defined as 8) multiplied by `sizeof(double)`, which is typically 8 bytes. A `doubleV8` instance is thereby defined as a vector of eight 8-byte floating-point values, which requires 512 bits in all.

With this `typedef` in place, the arithmetic operations in the remaining code are easy to read—and highly efficient. To begin, each of the two `doubleV8` variables, `dataV1` and `dataV2`, is initialized. Notice that the square brackets usually associated with arrays are absent. Here, for review, is the initialization of vector `dataV1`:

```
doubleV8 dataV1 = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8};
```

The vectors `dataV1` and `dataV2` can be used with indexes, as the three loops near the end of the code illustrate:

```
printf("%f ", div[i]); /* print ith value */
```

However, the arithmetic operations to add, multiply, and divide the vectors are one statement apiece in the source code. Here, for review, is the multiplication of the two vectors:

```
doubleV8 mul = dataV1 * dataV2; /* no looping! */
```

The standard compilers now make SIMD programming straightforward in C itself, without any additional libraries or tools. Of course, the underlying architecture must support machine-level SIMD instructions. It is reasonable to expect that SIMD architectures will continue to improve and that the C compilers will continue to generate code that takes advantage of the evolving SIMD instruction sets and architectures.

7.10. What's Next?

The next chapter covers miscellaneous topics to provide a better sense of the libraries available in C, both standard and third party. There is also a section on building software libraries from scratch. As usual, the code examples highlight the power and flexibility of C.

The forthcoming code examples cover regular expressions for pattern matching and data validation; assertions for enforcing conditions in code modules; locale management for internationalization; the compilation of C code into WebAssembly for high-performance web modules; signals for interprocess communication; and the building, deployment, and use (by both C and Python clients) of software libraries.

CHAPTER 8

Miscellaneous Topics

8.1. Overview

This chapter introduces libraries and topics not seen so far, but it also extends and refines the coverage of earlier material. For example, the flexible library function `system`, for quick multiprocessing, is introduced; the input function `scanf` is examined more closely.

The chapter begins with *regular expressions*, a language designed for pattern matching, which makes the language well suited for verifying input. Indeed, professional data validation relies on regular expressions as a base level. The chapter then moves to assertions, which allow the programmer to express and enforce constraints in a program. A section on locales and internationalization follows. Short code examples and full programs get into the details.

WebAssembly is a language designed for high-performance web modules, for example, ones that do serious number crunching. C is among the earliest languages (the others are C++ and Rust) to compile into WebAssembly. This section goes into detail with an full code example.

A *signal* is a low-level but still powerful way for one process to communicate with another, and C has an API for generating and handling signals. The section on signals is code oriented as usual.

M. Kalin, *Modern C Up and Running*, https://doi.org/10.1007/978-1-4842-8676-0_8

The chapter ends with a section on building static and dynamic libraries in C. It is no surprise that a client written in C can consume a library written in the same language, but almost every modern language can interoperate with C. This section underscores the point by having a Python client consume a C library built from scratch.

8.2. Regular Expressions

The *regular expression* language, or *regex* for short, is used to match strings against patterns and even for editing strings. Users of command-line utilities such as *grep* (short for *grab regular expression*) or *rename* already have experience with regex. In web and other applications, regex verification of user input is best practice; modern programming languages typically support regex. The first code example prompts a user for an employee ID and then checks whether the entered string matches a pattern that validates IDs.

Listing 8-1. A regex to check an employee ID

```
#include <stdio.h>
#include <regex.h>
#define MaxBuffer 64

void main() {
  char input[MaxBuffer];
  char error[MaxBuffer + 1]; /* null terminator */
  printf("Employee Id: ");
  scanf("%7s", input); /* read only 7 chars */

  const char* regex = "^[A-Z]{2}[1-9]{3}[a-k]{2}$"; /* regex as
                                                       a string */
  regex_t regex_comp;
```

```
  int flag;
  if ((flag = regcomp(&regex_comp, regex, REG_EXTENDED)) < 0) {
  /* compile regex */
    regerror(flag, &regex_comp, error, MaxBuffer);
    fprintf(stderr, "Error compiling '%s': %s\n", regex, error);
    return;
  }

  if (REG_NOMATCH == regexec(&regex_comp, input, 0, NULL, 0))
  /* match? */
    fprintf(stderr, "\n%s is an invalid employee ID.\n", input);
  else
    fprintf(stderr, "\n%s is a valid employee ID.\n", input);
  regfree(&regex_comp); /* good idea to clean up */
}
```

The *empId* program (see Listing 8-1) prompts the user for an employee ID and then reads the entered ID using `scanf`:

```
scanf("%7s", input); /* read only 7 chars */
```

The 7 in the format string `%7s` ensures that no more than seven characters are scanned into the buffer named `input`, which has room for 64 in any case.

The program then compiles a regex pattern given as a string. This pattern is the most complicated part of the program and so deserves careful analysis. The pattern consists of three parts, and each part consists of a *set* and a *count*. For now, ignore the start character ^ and the end character $; these are covered shortly.

The first set/count pair is

```
[A-Z]{2}
```

The square brackets represent a set, a collection of nonduplicate items in which order does not matter. For example, the set

```
[1234]
```

is the same as the set

```
[2143]
```

In the *empId* program, the members of the first set are the uppercase letters *A*,*B*,...,*Z*. These letters could be enumerated in the square brackets and in any order—a tedious undertaking. The regex language thus has a shortcut: `[A-Z]` means *the uppercase letters A through Z*.

Immediately after the set `[A-Z]` comes the *count* (*quantifier*) of how many characters from the set are required. The count occurs in braces:

```
[A-Z]{2} /* exactly 2 letters from the set A-Z */
```

The count can be flexible. For example, the count in

```
[A-Z]{2,4} /* 2 to 4 letters from the set A-Z */
```

allows two to four letters from the set.

The second part of the pattern requires exactly three decimal digits from the set `[1-9]`:

```
[1-9]{3} /* 3 digits, 1 through 9 */
```

The third part of the pattern requires two lowercase letters, but in the range of *a* through *k*:

```
[a-k]{2} /* 2 letters, a through k */
```

Here is a summary of other quantifier options:

```
[A-Z]?   /* zero or one from the set */
[A-Z]*   /* zero or more from the set */
[A-Z]+   /* one or more from the set */
```

The employee ID is supposed to begin with an uppercase letter and end with a lowercase letter. There should not be any other characters, including whitespace, flanking the employee ID on either side. To express this requirement, the regex expression uses *anchors*: the hat character ^ is the *left anchor*, and the dollar-sign character $ is the *right anchor*. Without these anchors, an employee ID such as

```
foobarAB123bb9876
```

would pass muster because the substring *AB123bb* matches the pattern without the anchors. The anchored expression requires that the ID start with an uppercase letter and end with a lowercase one.

The employee ID pattern as a string is compiled using the library function `regcomp`, which creates a `regex_t` instance if successful. The compiled pattern is used in matches. The last argument to `regcomp` is `REG_EXTENDED`, which enables various POSIX extensions to the original regex library. There is also a C library that supports Perl syntax and features (see www.pcre.org/), which has become the de facto standard for regex syntax.

Once the pattern is compiled, it can be used in a call to `regexec`, which matches the pattern against an input string. The call takes five arguments:

```
if (REG_NOMATCH == regexec(&pattern_comp, /* pattern */
                           input,         /* input string */
                           0,        /* zero capture groups */
                           NULL,     /* no capture array */
                           0))       /* no special flags */
```

The first two arguments are the address of the compiled pattern and the string to test against the pattern, which in this case is the user input. The next two arguments, `0` and `NULL`, are for *capture groups*: parts of the string to be tested can be *captured* for later reference. In this example, the capture option is not needed; hence, the number of capture groups is 0, and then there is `NULL` instead of an array in which to save the captures. A later example illustrates captures. The last argument consists of optional

integer flags, for example, a flag to ignore case when matching letters. In this example, there are no flags, which 0 represents.

The *empId* program works as advertised. For example, it accepts *AQ431af* as an employee ID but rejects *AQ431mf* (*m* is not between *a* and *k*, inclusive) and *AQ444kk7* (ends with a digit, not a letter).

A first experience with regex syntax may seem daunting, but a rhetorical question puts the challenge into perspective: Would it be easier to learn regex, or to write a program from scratch that does what the *empId* example requires? Regular expressions are not always intuitive, but they make up for this shortcoming with their power and flexibility.

Listing 8-2. A revised version of the empId program

```
#include <stdio.h>
#include <unistd.h>
#include <regex.h>
#define MaxBuffer 128
#define GroupCount  4 /* entire expression counts as one group
by default */

void main() {
  char error[MaxBuffer + 1];
  char* inputs[ ] = {"AABC123dd95", "Az4321jb81", "QQ987ii4",
                     "QQ98ii4", "YTE987ef4", "ARNQ999kk6", NULL};

  const char* regex = "^([A-Z]{2,4})([1-9]{3})([a-k]{2})
  [0-9]+$";
  regex_t regex_comp;
  int flag;
  if ((flag = regcomp(&regex_comp, regex, REG_EXTENDED)) < 0) {
    regerror(flag, &regex_comp, error, MaxBuffer);
    printf("Regex error compiling '%s': %s\n", regex, error);
    return;
  }
```

```
  unsigned i = 0, j;
  while (inputs[i]) { /* iterate over the inputs */
    regmatch_t groups[GroupCount]; /* for extracting
    substrings */
    if (REG_NOMATCH == regexec(&regex_comp, inputs[i],
    GroupCount, groups, 0))
      fprintf(stderr, "\t%s is not a valid employee ID.\n",
      inputs[i]);
    else {
      fprintf(stdout, "\nValid employee ID. %i parts
      follow:\n", GroupCount);
      for (j = 0; j < GroupCount; j++) {
        if (groups[j].rm_so < 0) break;
        write(1, inputs[i] + groups[j].rm_so, groups[j].rm_eo -
        groups[j].rm_so);
        write(1, "\n", 1);
      }
      printf("-----");
    }
    i++; /* loop counter */
  }
  regfree(&regex_comp); /* good idea to clean up */
}
```

The *empId2* program (see Listing 8-2) adds features to the original *empId* program. The new features can be summarized as follows:

- An employee ID may start out with between two and four letters. In the fictitious company for which the employees work, the number of starting letters is a security code: two letters is low-security, three is middle-security, and four is high-security clearance.

- An employee ID must end with one or more decimal digits.
- The *empId2* program introduces *groups*, the three parenthesized expressions, in order to parse the employee ID.

The revised regex expression is

```
^([A-Z]{2,4})([1-9]{3})([a-k]{2})[0-9]+$ ## [0-9]+ means 1 or
more decimal digits
```

The anchors remain, but the end requirement for *one or more* decimal digits is new. The other major change is the use of parenthesized subexpressions, each of which represents a *group* that is captured for later analysis.

The major change in the rest of the code has to do with *group captures*. The code declares an array:

```
regmatch_t groups[GroupCount]; /* for extracting substrings */
```

The value of `GroupCount` is four, one more than the number of parenthesized subexpressions (in this case, three) in the regex. The reason is that the *entire* string to be matched counts as one group, in fact the first. The `regmatch_t` type is

```
typedef struct {
   regoff_t rm_so; /* start offset */
   regoff_t rm_eo; /* end offset */
} regmatch_t;
```

The two offsets indicate where, in the string to be matched, the different groups begin and end. The `groups` array, in the current example, has four elements of this type. For the first string to be matched, *AABC123dd95*, the start index (`rm_so` in the structure) for the first subexpression is 0, and the end index (`rm_eo`) is 4, immediately beyond the last character *C* in the first subexpression.

Given the `regmatch_t`, it is straightforward to print the captured groups in valid employee IDs. Indeed, the easy way is to use the low-level I/O API. Here is the relevant statement:

```
write(1,                                    /* stdout */
      inputs[i] + groups[j].rm_so,          /* start */
      groups[j].rm_eo - groups[j].rm_so); /* length */
```

The first argument to `write` is, of course, the standard output. The second argument takes the base address of a test string (for instance, `inputs[0]` is the string *AABC123dd95*) and adds the start offset (`rm_so`, which is 0, 4, or 7). The third argument to `write` is the captured part's length: the end index (one beyond the end of the part) minus the start index. The output for parsing the first two candidate IDs is

```
Valid employee ID. 4 parts follow:
AABC123dd95
AABC
123
dd
        Az4321jb81 is not a valid employee ID.
```

The standard C library for regex covers the basics but does not include newer features such as *lookaheads*. These features make it easier or more efficient to do pattern matching that still can be done without them. The previously mentioned PCRE (Perl Compatible Regular Expressions) library is an option for such newer features.

8.3. Assertions

An *assertion* checks whether a program satisfies a condition at a specified point in its execution. There are three traditional types of assertion that can be used to check a program module such as a C block:

- An assertion expressing a *precondition*, which must hold at the start of a block
- An assertion expressing a *postcondition*, which must hold at the end of a block
- An assertion expressing an *invariant*, which must hold throughout a block

C implements assertions with the `assert` macro, which takes an arbitrary boolean expression as its argument. If the `assert` evaluates to *true* (nonzero), the program continues execution; otherwise, the program aborts with an explanatory error message.

Listing 8-3. Using assertions to track login attempts

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

#define MaxBuffer 64
#define MaxTries 3

unsigned check_id(const char* id, regex_t* regex) {
  return REG_NOMATCH != regexec(regex, id, 0, NULL, 0);
}

void main() {
  const char* regex_s = "^[A-Z]{2,4}[1-9]{3}[a-k]{2}[0-1]?$";
  regex_t regex_c;
```

```
  if (regcomp(&regex_c, regex_s, REG_EXTENDED) < 0) {
    fprintf(stderr, "Bad regex. Exiting.\n");
    return;
  }

  char id[MaxBuffer];
  unsigned tries = 0, flag = 0;
  assert(0 == tries);            /* precondition */

  do {
    assert(tries < MaxTries);    /* invariant */
    printf("Employee Id: ");
    scanf("%10s", id);
    if (check_id(id, &regex_c)) {
      flag = 1;
      break;
    }
    tries++;
  } while (tries < MaxTries);

  assert(tries <= MaxTries);        /* postcondition */
  regfree(&regex_c); /* clean up */
  if (flag) printf("%s verified.\n", id);
  else printf("%s not verified.\n", id);
}
```

The *verifyEmp* program (see Listing 8-3) builds on the earlier *empId* program, in particular by using a regex to verify an employee's ID. The regex itself has changed a little in order to show more aspects of the language:

```
^[A-Z]{2,4}[1-9]{3}[a-k]{2}[0-1]?$ /* new part is: [0-1]? */
```

This pattern allows the starting uppercase letters to be between two and four in number and makes a single ending digit (either 0 or 1) optional. The function `check_id` takes two arguments, the ID to verify and the compiled regex; the function returns either *true*, if the candidate ID matches the regex, or *false* otherwise.

The program uses a `do while` loop to prompt the user for an employee ID. Of interest now is that the employee is to get no more than `MaxTries` chances to enter the ID. Similar approaches are used for login/password combinations, of course. The loop condition is

```
while (tries < MaxTries)
```

where `tries` is updated on each attempt and `MaxTries` is a macro defined as 3. If this condition were changed to

```
while (tries < MaxTries + 1)
```

and the user failed to provide a valid ID, the program would abort, and the error message from the failed assertion would be

```
empId3: empId3.c:24: main: Assertion 'tries < 3' failed.
```

The 24 represents line 24 in the source code, the assertion immediately after the `do`:

```
assert(tries < MaxTries); /* invariant */
```

The *verifyEmp* program has three assertions, each with a different test:

- The *precondition* occurs immediately before the loop starts. It checks that, at this point, the value of `tries` is zero. If `tries` were not initialized at all, then—as a stack-based variable—its value would be random and possibly greater than `MaxTries` already. The *precondition* is evaluated exactly once, as it occurs before the loop.

- The *postcondition* occurs immediately after the loop ends. It checks that, at this point, `tries` is less than *or equal to* the value of `MaxTries`. There are two possibilities:
    - Suppose that the candidate ID is verified in any one of the three allowed attempts. Even if success comes at the third and final attempt, the value of `tries` is only 2 and so still less than `MaxTries`, which is 3.
    - Suppose that the candidate ID fails three times. Control then exits the loop because of the loop test that the value of `tries` be strictly less than the value of `MaxTries`: both `tries` and `MaxTries` now have a value of 3. The loop test has done its job, and so the program should continue to run normally. The *postcondition* thus must allow `tries` to be less than *or equal to* the value of constant `MaxTries`.
- The *invariant* occurs immediately inside the loop, which is the only place that `tries` changes after its initialization to zero. On each iteration, `tries` is incremented by 1. If the candidate ID is verified, then the `break` statement, rather than the loop test, is what moves control beyond the loop. If `tries` is incremented to 3, then the loop condition, not the `break` statement, should cause control to exit the loop. Accordingly, the *invariant* checks that `tries` is always *less than* `MaxTries`.

The syntax of assertions is easy in C, but the reasoning behind assertion tests and assertion placement can be complicated. Even a program as relatively simple as *verifyEmp* confirms the point. The complication arises because assertions articulate reasoning about program correctness—and determining what makes a program correct is notoriously hard.

C has a convenient way to turn assertions off without commenting out the `assert` statements or deleting them from the source code. In a file with assertions, simply define the macro `NDEBUG`:

```
#define NDEBUG /* turns off assertions */
```

As code development moves from testing to production, it is common to turn assertions off.

8.4. Locales and i18n

Date, currency, and other information should be formatted in a locale-aware way as part of i18n programming, where *i18n* abbreviates *internationalization*. (The skeptic should count the letters between the *i* and the *n*.) Consider, for example, this large number formatted in a way familiar to North Americans:

```
1,234,567,891.234
```

In Germany, Italy, or Norway, the expected format would be

```
1 234 567.891,234
```

Locale information is available as part of the *environment* of a local system. When a C program begins execution, the program inherits environment variables about the locale and other features, but this locale inheritance does *not* extend to library functions that the program may

call. Accordingly, a locale-aware program needs to do some initialization. Before looking at this initialization in code, it will be useful to consider how a C program can get environment information in general.

Listing 8-4. How to get information about the program environment

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

extern char** environ; /* declaration */

void main () {
  int i = 0;
  while (environ[i]) printf("%s\n", environ[i++]);

  printf("Locale: %s\n", getenv("LANG")); /* en_US.UTF-8 */

  char cmd[32];
  strcpy(cmd, "locale -a");
  int status = system(cmd);
  printf("\n%s exited with %i\n", cmd, status);
}
```

The *environ* program (see Listing 8-4) shows two ways to access environment information. The first way uses the `extern` variable `environ`, an array of strings each with a *key=value* format. Here, for example, are two entries from my desktop system: the first key/value pair provides information about the terminal and the second about the *shell* language.

```
TERM=xterm
SHELL=/bin/bash
```

The library function `getenv` takes a single argument, a key such as `TERM` or `SHELL` as a string. The `printf` call illustrates with the key `LANG`, which gives a standard abbreviation (`en_US` for *English in the United States*) together with the *character encoding* scheme, in this case `UTF-8` (Unicode Transformation Format-8). UTF-8 formats multibyte Unicode character encodings as a sequence of 8-bit bytes.

The last part of the *environ* program introduces the versatile `system` function. This function takes a single string argument, which represents a *shell command*, that is, a command that can be given at the command line. The `system` function starts another process and then blocks until the started process terminates. The `int` value returned to the `system` function is the *exit status* of the process in question. In this example, the command is *locale -a*, a utility that (with the *-a* flag) lists all of the locales available on the system. (The *locale* utility is available on Unix-like systems and on Windows through Cygwin.)

A given system supports some locales, but not others. The system administrator is responsible for installing and otherwise managing locale information. At the command line, or through the *environ* program shown previously, a listing of locales would look something like this:

```
C
C.UTF-8
en_AG.utf8
en_AU.utf8
...
```

The string `en_AG.utf-8` represents English in Antigua, whereas `en_AU.utf8` represents English in Australia. The first two entries, `C` and `C.UTF-8`, represent the default locale. In the `setlocale` function, investigated shortly, entries such as `C.UTF-8` can be used as an argument.

Here is the declaration for the `setlocale` function:

```
char* setlocale(int category, const char* locale);
```

If the second argument is NULL, the function acts as a *getter* or query: the function returns a string that represents the current locale. If the second argument is not NULL, the function acts as a *setter* by setting the locale represented by the second argument, a string. (The empty string as the second argument also represents the default locale C.) Furthermore, the string returned from setlocale is *opaque* and typically prints as (null). This string is useful only as a second argument to setlocale. A typical use of the string would be as follows:

1. Retrieve the current locale, and save it as a string.
2. Set the locale to something new, and perform whatever application logic is appropriate.
3. Restore the saved locale by using the string from step 1 as the second argument to setlocale.

The next code example illustrates.

Listing 8-5. Introducing the setlocale function

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

void main () {
  setlocale(LC_ALL, ""); /* set current locale for library
                            functions */
  char* prev_locale = setlocale(LC_ALL, NULL);
                          /* with NULL, a getter, not a setter */
  char* saved_locale = strdup(prev_locale);
                                        /* get a separate copy */
```

```
  if (NULL == saved_locale) {          /* verify the copying */
    perror(NULL); /* out of memory */
    return;
  }

  const struct lconv* loc = localeconv(); /* get ptr to current
                                             locale struct */
  printf("Currency symbol: %s\n", loc->currency_symbol);
  setlocale(LC_ALL, "en_GB.utf8"); /* english in Great
                                      Britain */
  loc = localeconv();
  printf("Currency symbol: %s\n", loc->currency_symbol);

  setlocale(LC_ALL, saved_locale); /* restored saved locale */
  /*...*/
}
```

The *localeBasics* program (see Listing 8-5) opens with two calls to library function `setlocale`, but the calls are quite different. The first call has the empty string, hence non-`NULL`, as its second argument:

```
setlocale(LC_ALL, ""); /* set current locale for library
                          functions */
```

The integer macro `LC_ALL` represents all of the locale categories, and the empty string represents the default locale. Because the second argument is a string, even though empty, this call to `setlocale` acts as a *setter* rather than a *getter* of information.

The immediately following call to the `setlocale` function acts as a *getter*:

```
char* prev_locale = setlocale(LC_ALL, NULL);
                            /* with NULL as 2nd arg, a getter */
```

The program then uses the strdup function (*string duplicate*) to make an altogether separate copy of this string just in case there are further calls to setlocale. Note that setlocale returns a *pointer* to a string, not a copy of this string.

The program ends by resetting the locale to the saved_locale. The save/restore pattern is common in locale-aware programs.

In the middle, the *localeBasics* program calls the library function localeconv to get a pointer to a structure that contains information in all of the locale categories. This structure is displayed shortly. For now, the pointer loc is used to access the currency symbol, first for the United States and then for Great Britain. The output is

```
Currency symbol: $ /* default locale, en_US */
Currency symbol: £ /* en_GB */
```

At the end, the program resets the locale to the original one:

```
setlocale(LC_ALL, saved_locale); /* restored saved locale */
```

Recall that saved_locale is a string copy of the original locale and so not NULL. This call to setlocale is therefore a *setter*, which restores the locale back to the original setting.

Listing 8-6. The lconv structure with locale information

```
typedef struct {
   char *decimal_point;
   char *thousands_sep;
   char *grouping;
   char *int_curr_symbol;
   char *currency_symbol;
   char *mon_decimal_point;
   char *mon_thousands_sep;
   char *mon_grouping;
```

```
    char *positive_sign;
    char *negative_sign;
    char int_frac_digits;
    char frac_digits;
    char p_cs_precedes;
    char p_sep_by_space;
    char n_cs_precedes;
    char n_sep_by_space;
    char p_sign_posn;
    char n_sign_posn;
} lconv;
```

Locale information is stored in a structure of type `lconv` (see Listing 8-6), and the library function `localeconv` returns a pointer to a typically `static` instance of this structure. The 18 fields contain locale-specific information. In Canada, for example, the `decimal_point` is the period symbol, whereas in Germany, the `decimal_point` is the comma symbol.

Table 8-1. *Argument categories for* `setlocale`

Category	Meaning
`LC_ALL`	All of the below
`LC_COLLATE`	regex string settings
`LC_CTYPE`	regex, character conversion, etc.
`LC_MESSAGES`	Localizable natural-language messages
`LC_MONETARY`	Currency formatting
`LC_NUMERIC`	Number formatting
`LC_TIME`	Time and date formatting

The fields in the lconv structure are numerous, and there are connections among many of them. The connections may not be evident. Accordingly, these fields are divided into seven categories, with macros to define each category (see Table 8-1). The categories make it easier to set related pieces of locale information.

A typical call to function setlocale uses the LC_ALL category as the first argument:

```
setlocale(LC_ALL, ""); /* set all categories to default
locale */
```

For fine-tuning, however, a specific category could be used instead as the first argument:

```
setlocale(LC_MONETARY, "en_GB.utf-8"); /* monetary category for
Great Britain */
```

The next code example puts the LC_MONETARY category to use. The program first sets all locale categories (LC_ALL) to local settings. The program then resets LC_MONETARY only to get locale-specific currency information from six English-speaking regions around the world.

Listing 8-7. Using the category LC_MONENTARY

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

void main () {
  setlocale(LC_ALL, ""); /* set all categories to default
  locale */
  char* regions[ ] = {"en_AU.utf-8", "en_CA.utf-8",
  "en_GB.utf-8", "en_US.utf-8", "en_NZ.utf-8",
  "en_ZM.utf-8", NULL};
```

```
  int i = 0;
  while (regions[i]) {
    setlocale(LC_MONETARY, regions[i]); /* change the locale */
    const struct lconv* loc = localeconv();
    printf("Region: %s Currency symbol: %s International
    currency symbol: %s\n",
           regions[i], loc->currency_symbol, loc->int_curr_
            symbol);
    i++;
  }
}
```

The *locMonetary* program (see Listing 8-7) initializes the array `regions` to standard codes for six English-speaking regions around the world. For each of these regions, the `LC_MONETARY` category is set before the `currency_symbol` and the `int_curr_symbol` (*international currency symbol*) are printed in a `while` loop. The `localeconv` library function is called to get a pointer to the `lconv` structure that stores the desired information.

Listing 8-8. Output from the locMonetary program

```
Region: en_AU.utf-8  Currency symbol: $  International currency
symbol: AUD
Region: en_CA.utf-8  Currency symbol: $  International currency
symbol: CAD
Region: en_GB.utf-8  Currency symbol: £  International currency
symbol: GBP
Region: en_US.utf-8  Currency symbol: $  International currency
symbol: USD
Region: en_NZ.utf-8 Currency symbol: $ International currency
symbol: NZD
Region: en_ZM.utf-8 Currency symbol: K International currency
symbol: ZMK
```

The output from the *locMonetary* program (see Listing 8-8) shows the region, currency symbol, and international currency acronym for the six regions.

8.5. C and WebAssembly

WebAssembly is a language well-suited for compute-bound tasks (e.g., number crunching) executed on a browser. All rumors to the contrary, the WebAssembly language is not meant to replace JavaScript, but rather to supplement JavaScript by providing better performance on CPU-intensive tasks that JavaScript otherwise might perform. JavaScript remains the glue that ties together HTML pages and WebAssembly modules:

```
HTML pages<--->JavaScript<--->WebAssembly modules
```

WebAssembly has an advantage over other web artifacts when it comes to downloading. For example, a browser fetches HTML pages, CSS stylesheets, and JavaScript code as text, an inefficiency that WebAssembly addresses: a WebAssembly module has a compact binary format, which speeds up downloading.

After a WebAssembly program is downloaded to a browser, the just-in-time (JIT) compiler in the browser's virtual machine translates the binary WebAssembly code into fast, platform-specific machine code. Here is a summary depiction:

```
             download  +-------+ translate
wasm module---------->|browser|---------->fast machine code
                      +-------+
```

JavaScript code embedded in an HTML page can call functions delivered in WebAssembly modules.

WebAssembly has a development language known as the *text format* language, which has a Lisp-like syntax for writing programs on a virtual

stack-based machine. However, code from higher-level programming languages (including C) can be translated in WebAssembly. Although the list of languages that can be translated into WebAssembly is growing, the original ones were C, C++, and Rust—three languages suited for systems programming and high-performance applications programming. These three languages share two features that promote fast execution: explicit data typing and no garbage collector.

When it comes to high-performance web code, WebAssembly is not the only game in town. For example, asm.js is a JavaScript dialect designed, like WebAssembly, to approach native speed. The asm.js dialect invites optimization because the code mimics the explicit data types in the three aforementioned languages. Here is an example with C and then asm.js. The sample function in C is

```
int f(int n) {  /** C **/
  return n + 1;
}
```

Both the parameter `n` and the returned value are explicitly typed as `int`. The equivalent function is asm.js would be

```
function f(n) { /** asm.js **/
  n = n | 0;
  return (n + 1) | 0;
}
```

JavaScript, in general, does not have explicit data types, but a bitwise-OR operation in JavaScript yields an integer value. This explains the otherwise pointless bitwise-OR operation:

```
n = n | 0; /* bitwise-OR of n and zero */
```

The bitwise-OR of n and zero evaluates to n, but the purpose here is to signal that n holds an integer value. The return statement repeats this optimizing trick. Among the JavaScript dialects, TypeScript stands out for adopting explicit data types, which makes this language attractive for compilation into WebAssembly.

Almost any discussion of the WebAssembly language covers *near-native speed* as one of the language's major design goals. The native speed is that of the compiled systems languages C, C++, and Rust; hence, these three languages were also the originally designated candidates for compilation into WebAssembly.

8.5.1. A C into WebAssembly Example

A production-grade example would have WebAssembly code perform a heavy compute-bound task such as generating large cryptographic key pairs or using such pairs for encryption and decryption. A simpler example fits the bill as a stand-in that is easy to follow. There is number crunching, but of the routine sort.

Consider the function *hstone* (for *hailstone*), which takes a positive integer as an argument. The function is defined as follows:

```
                3N + 1 if N is odd
hstone(N) =
                N/2 if N is even
```

For example, hstone(12) returns 6, whereas hstone(11) returns 34. If N is odd, then 3N+1 is even; but if N is even, then N/2 could be either even (e.g., 4/2 = 2) or odd (e.g., 6/2 = 3).

The *hstone* function can be used iteratively by passing the returned value as the next argument. The result is a hailstone sequence such as this one, which starts with 24 as the original argument, the returned value 12 as the next argument, and so on:

```
24,12,6,3,10,5,16,8,4,2,1,4,2,1,...
```

It takes ten calls for the sequence to converge to 1, at which point the sequence of 4,2,1 repeats indefinitely: (3x1)+1 is 4, which is halved to yield 2, which is halved to yield 1, and so on. The Wikipedia page (https://en.wikipedia.org/wiki/Collatz_conjecture) goes into technical detail on the hailstone function, including a clarification of the name *hailstone*.

Note that powers of two (2^N) converge quickly to 1, requiring just N divisions by two to reach 1. For example, 32 (2^5) has a convergence length of five, and 64 (2^6) has a convergence length of six. A hailstone sequence converges to 1 if and only if the sequence generates a power of two. At issue, therefore, is whether a hailstone sequence inevitably generates a power of two.

The Collatz conjecture is that a hailstone sequence converges to 1 no matter what the initial argument N > 0 happens to be. No one has found a counterexample to the Collatz conjecture, nor has anyone come up with a proof to elevate the conjecture to a theorem. The conjecture, simple as it is to test with a program, remains a profoundly challenging problem in mathematics. My *hstone* example generates hailstone sequences and counts the number of steps required for a sequence to hit the first 1.

8.5.2. The Emscripten Toolchain

The systems languages, including C, require specialized toolchains to translate source code into WebAssembly. Emscripten is a pioneering and excellent option, one built upon the well-known LLVM (Low-Level Virtual Machine) compiler infrastructure. Emscription can be installed following the instructions at https://emscripten.org/docs/getting_started/downloads.html.

To begin, consider this version of a C *hstone* program (see Listing 8-9) with two functions, the familiar entry point `main` and `hstone`, which `main` invokes repeatedly.

Listing 8-9. The hstoneCL program with `main`

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

int hstone(int n) {
  int len = 0;
  while (1) {
    if (1 == n) break; /* halt on 1 */
    if (0 == (n & 1)) n = n / 2; /* if n is even */
    else n = (3 * n) + 1; /* if n is odd */
    len++; /* increment counter */
  }
  return len;
}

#define HowMany 8
int main() {
  srand(time(NULL)); /* seed random number generator */
  int i;
  puts(" Num Steps to 1");

  for (i = 0; i < HowMany; i++) {
    int num = rand() % 100 + 1; /* + 1 to avoid zero */
    printf("%4i %7i\n", num, hstone(num));
  }
  return 0;
}
```

On a sample run, the *hstoneCL* program (with *CL* for *command line*) had this output:

```
Num   Steps to 1
64        6
```

```
40          8
86         30
16          4
30         18
47        104
12          9
60         19
```

The *hstoneCL* program can be webified—with no changes whatsoever to the source code—by using the Emscription toolchain, which can do the following:

- Compile the C source into a WebAssembly module.
- Generate a test HTML page with calls to ams.js code that, in turn, invokes the `hstone` function through a call to `main`.

However, the WebAssembly module does not require the `main` function because JavaScript could invoke the `hstone` function directly. The *hstone* program can be simplified by dropping the `main` function in the *hstoneCL* version.

The *hstoneWA* revision (see Listing 8-10) drops `main` and adds the directive `EMSCRIPTEN_KEEPALIVE` to the `hstone` function. This directive informs the compiler that the C function named `hstone` should be exposed, under the same name, as a WebAssembly function.

Listing 8-10. The revised hstone code

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <emscripten/emscripten.h>

int EMSCRIPTEN_KEEPALIVE hstone(int n) {
```

```
  int len = 0;
  while (1) {
    if (1 == n) break; /* halt on 1 */
    if (0 == (n & 1)) n = n / 2; /* if n is even */
    else n = (3 * n) + 1; /* if n is odd */
    len++; /* increment counter */
  }
  return len;
}
```

As noted earlier, the Emscripten toolchain can be used not only to compile C code into WebAssembly but also to generate an appropriate HTML page together with JavaScript glue that links the WebAssembly module with the HTML page. To understand the details, however, it is useful to generate only the WebAssembly module and to craft, by hand, the HTML page and some JavaScript test calls.

With the Emscripten toolchain installed, the C function `hstone` in the file *hstoneWA.c* can be compiled into WebAssembly from the command line as follows:

```
% emcc hstoneWA.c --no-entry -o hstone.wasm
```

The flag `--no-entry` indicates that the file *hstoneWA.c* does not contain the function `main`, and the `-o` flag stands for *output*: the resulting WebAssembly file is named *hstone.wasm*. On my desktop machine, this file is a trim 662 bytes in size.

For testing, the next requirement is an HTML page that, when downloaded to a browser, fetches the WebAssembly module. A production-grade version of the HTML page would include embedded JavaScript calls to appropriate WebAssembly functions. A handcrafted version of the HTML page reveals details that otherwise remain hidden. Here is an HTML page that downloads and prepares the WebAssembly module stored in the *hstone.wasm* file:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8"/>
    <script>
      fetch('hstone.wasm').then(response =>     <!-- Line 1 -->
      response.arrayBuffer()                    <!-- Line 2 -->
      ).then(bytes =>                           <!-- Line 3 -->
      WebAssembly.instantiate(bytes, {imports: {}})
      <!-- Line 4 -->
      ).then(results => {                       <!-- Line 5 -->
      window.hstone = results.instance.exports.hstone;
      <!-- Line 6 -->
});
    </script>
  </head>
  <body/>
</html>
```

The `script` element in the preceding HTML page can be clarified line by line. The `fetch` call in Line 1 uses the web Fetch module to get the WebAssembly module from the web server that hosts this HTML page. When the HTTP response arrives, the WebAssembly module does so as a sequence of bytes, which are stored in the `arrayBuffer` of the script's Line 2. These bytes make up the WebAssembly module, the contents of the file *hstone.wasm*. This module has no imports from other WebAssembly modules, as indicated at the end of Line 4.

At the start of Line 4, the WebAssembly module is instantiated. A WebAssembly module is akin to a nonstatic class with nonstatic members in an object-oriented language such as Java. The module contains variables, functions, and various support artifacts; but the module must be instantiated to be called from JavaScript.

The script's Line 6 exports the original C function `hstone` under the same name. This WebAssembly function is available now to any JavaScript code, as a session in the browser's JavaScript console confirms. Here is part of my test session in Chrome's JavaScript console:

```
> hstone(27)      ## invoke hstone by name
< 111             ## output
> hstone(7)       ## again
< 16              ## output
```

The outputs are the steps required to reach 1 from the input (e.g., `hstone(27)` requires 111 steps to reach 1).

WebAssembly now has a more concise API for fetching and instantiating a module; the new API reduces the preceding script to only the fetch and instantiate operations. The longer version shown previously has the benefit of exhibiting details, in particular the representation of a WebAssembly module as a byte array that gets instantiated as an object with exported functions.

Emscripten comes with a test server, which can be invoked as follows to host the handcrafted HTML file *hstone.html* and the WebAssembly file *hstone.wasm*:

```
% emrun --no_browser --port 7777 .
```

The flag `--no_browser` means that a user manually opens a browser such as Firefox or Chrome. The request URL from the browser is then *localhost:7777/hstone.html*. If all goes well, the browser's JavaScript console can be used to confirm, as shown previously, that the WebAssembly module is available for use.

8.5.3. WebAssembly and Code Reuse

The `EMSCRIPTEN_KEEPALIVE` directive is the straightforward way to have the Emscripten compiler produce a WebAssembly module that exports any

C function of interest to the JavaScript glue embedded in an HTML page. A customized HTML document, with whatever handcrafted JavaScript is appropriate, can call the functions exported from the WebAssembly module. Hats off to Emscripten for this clean approach.

Web programmers are unlikely to write WebAssembly in its own *text format* language, as compiling from some high-level language, such as C or Rust, is far too attractive an option. Compiler writers, by contrast, might find it productive to work at the fine-grained level that the *text format* language provides.

Much has been made of WebAssembly's goal of achieving near-native speed. But as the JIT compilers for JavaScript continue to improve, and as dialects well-suited for optimization (e.g., TypeScript) emerge and evolve, it may be that JavaScript also achieves near-native speed. Would this imply that WebAssembly is wasted effort? I think not.

WebAssembly addresses another traditional goal in computing: code reuse. As even the short *hstone* example illustrates, code in a suitable language, such as C, translates readily into a WebAssembly module, which plays well with JavaScript code—the glue that connects a range of technologies used on the Web. WebAssembly is thus an inviting way to reuse legacy code and to broaden the use of new code. For example, a high-performance program for image processing, written originally as a desktop application, might also be useful in a web application. WebAssembly then becomes an attractive path to reuse. (For new web modules that are compute bound, WebAssembly is a sound choice.) My hunch is that WebAssembly will thrive as much for reuse as for performance.

8.6. Signals

A *signal* interrupts an executing program (*process*) to notify it of some exceptional event:

```
                                interrupt +---------+
signal from outside the program---------->| process |
             /                            +---------+
  e.g., Control-C from the keyboard
```

Signals have integer values as identifiers, with symbolic constants such as `SIGKILL` for ease of reference. When interrupted through a signal, a process may be able to ignore the interruption or else handle it in some program-appropriate way. However, some signals cannot be ignored, in particular `SIGKILL` (terminate) and `SIGSTOP` (pause).

Operating system routines regularly use signals to notify a process of an exceptional condition. For example, if a process runs out of memory, an operating system routine uses a signal as notification. Programs designed to handle signals typically do so in one of two ways:

- The program requests that the signal be ignored. Recall the *basicFork* program (see Listing 7-1), which included this call to the `signal` function:

  ```
  signal(SIGCHLD, SIG_IGN); /** prevent child
  from becoming a permanent zombie **/
  ```

  The call requests that the `SIGCHLD` signal, which the system sends to a parent process when a child terminates, be ignored. The motive is to prevent the child from becoming a permanent *zombie* process, if the parent should happen to terminate before the child.

- The program provides a *signal handler* as a callback function automatically invoked when a specified signal occurs. For example, the `SIGINT` (*interrupt*) signal can be sent to a process by hitting Control-C in the terminal window from which the program is launched. Perhaps a user hits Control-C by accident: the program might handle the signal by asking the user to confirm that the running program should be stopped.

At the core of the signal library is the legacy `signal` function, but best practice now favors the newer `sigaction` function. The `signal` function may behave differently across platforms and even operating system versions. The forthcoming code example uses the better-behaved `sigaction` function, introduced as a POSIX replacement for `signal`.

Listing 8-11. A signal-handling program

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

#define MaxLoops 500

void cntrlC_handler(int signum) { /** callback function: int
arg, void return **/
  fprintf(stderr, "\n\tHandling signal %i\n", signum);
  int ans = 1;
  printf("Sure you want to exit (1 = yes, 0 = no)? ");
  scanf("%i", &ans);
  if (1 == ans) exit(EXIT_SUCCESS);
}
```

```
void main() {
  /** Set up a signal handler. **/
  struct sigaction current;
  sigemptyset(&current.sa_mask);       /* clear the
                                          signal set */
  current.sa_flags = 0;                /* enables setting
                                          sa_handler, not sa_
                                          action */
  current.sa_handler = cntrlC_handler; /* specify a handler */
  sigaction(SIGINT, &current, NULL);   /* control-C is a
                                          SIGINT */

  int i;
  for (i = 0; i < MaxLoops; i++) {
    printf("Counting sheep %i...\n", i + 1);
    sleep(1);
  }
}
```

The *signals* program (see Listing 8-11) introduces the basic signal API. Here is an overview of how the program handles `SIGINT` and why the program does so:

- The `main` function has a tiresome loop that prints integer values 1 through `MaxLoops`, currently set at 500. After printing each value, the program sleeps for a second. A user will be inclined to terminate this program from the command line with a Control-C.
- At the start of `main`, a *signal handler* is registered for `SIGINT`, which a Control-C from the keyboard can generate. A program's default response to a `SIGINT` is termination.

- The signal handler `cntrlC_handler` can have any name but should return `void` and take a single `int` argument, which is the signal number. (The integer value for `SIGINT` happens to be 2.) This signal handler prompts the user for confirmation: if the user confirms, the program exits; otherwise, the program continues as before.

To record a signal handler using the `sigaction` function, a program first uses an instance of the `struct sigaction` type to set relevant information. In this example, the *signal set* for the process first is emptied; the relevant field is `sa_mask`, whose address is passed to the library function `sigemptyset`. In general, a child process may inherit signal behavior from a parent, and so clearing the signal set may be done to wipe out the inheritance. In this case, the call to `sigemptyset` is simply to illustrate details of the API.

Two different callback types can be registered with the `sigaction` function: one takes a single argument (the signal number), and the other takes three arguments (the signal number and pointers to two different structures that contain pertinent information about the current process state with respect to signals). The initialization

```
current.sa_flags = 0; /* current is a struct sigaction
instance */
```

is a setup for using the simpler of the two callbacks:

```
current.sa_handler = cntrlC_handler; /* cntrlC_handler is the
1-argument callback */
```

If the `sa_action` field were used instead, then the `sa_flags` field would indicate which pieces of signal information were of interest.

The `sigaction` function, which sets the desired signal-handling action, takes three arguments:

```
sigaction(SIGINT, &current, NULL);
```

The first argument is the signal number, in this case SIGINT. The second argument is a pointer to the *new* signal-handling action, and the last argument is a pointer to the *previous* action, which can be saved with a non-NULL pointer for later retrieval. In this example, the old action is not saved: the third argument is NULL. Each *action* is specified by setting a field in an instance of the struct sigaction type.

Listing 8-12. A sample run of the signals program

```
% ./signals                                  ## on Windows, drop ./
Counting sheep 1...
Counting sheep 2...
^C                                            ## 1st Control-C
        Handling signal 2
Sure you want to exit (1 = yes, 0 = no)? 0 ## resume execution
Counting sheep 3...
Counting sheep 4...
^C                                            ## 2nd Control-C
        Handling signal 2
Sure you want to exit (1 = yes, 0 = no)? 1 ## terminate
%
```

A sample run (see Listing 8-12) of the *signals* program confirms that the signal handling works as expected. As the loop starts, there is a Control-C from the user, and then a user response of 0, which means *continue*. The looping thus goes on. After a second Control-C and a user response of 1, which means *terminate*, the program ends.

Signals are a powerful, widely used mechanism not only for user/program interaction but also for interprocess communication. For example, the kill function

```
int kill(pid_t pid, int signum)
```

can be used by one process to terminate another process or group of processes. If the first argument to function `kill` is greater than zero, this argument is treated as the *pid* of the targeted process; if the argument is zero, the argument identifies the *group* of processes to which the signal sender belongs. The graceful shutdown of a multiprocessing application such as a web server could be accomplished by killing a group of processes. The second argument to `kill` is either a standard signal number (e.g., `SIGTERM` terminates a process but can be blocked, whereas `SIGKILL` terminates a process and cannot be blocked) or `0`, which makes the call to `signal` a query about whether the *pid* in the first argument is indeed valid.

The older `signal` function is still used widely and dominates in legacy code. It is worth repeating that the `sigaction` replacement is the preferred way forward.

8.7. Software Libraries

Software libraries are a long-standing, easy, and sensible way to reuse code and to extend C by providing new functionalities. This section explains how to build such libraries from scratch and to make them easily available to clients. Although the two sample libraries target Linux, the steps for creating, publishing, and using these libraries apply in essentials to other Unix-like systems.

There are two sample clients (one in C, the other in Python) to access the libraries. It is no surprise that a C client can access a library written in C, but the Python client underscores that a library written in C can serve clients from other languages.

Computer systems in general and Linux in particular have two types of library:

- A static library (*library archive*) is baked into a statically compiled client (e.g., one in C or Rust) during the

link phase of the compilation process. In effect, each client gets its own copy of the library. A significant downside of a static library comes to the fore if the library needs revision, for example, to fix a bug—each library client now must be linked to the revised static library. A dynamic library, described next, avoids this shortcoming.

- A dynamic (*shared*) library is flagged during the link phase of a statically compiled client program, but the client program and the library code remain otherwise unconnected until runtime—the library code is not baked into the client. At runtime, the system's dynamic loader connects a shared library with an executing client, regardless of whether the client comes from a statically compiled language such as C or a dynamically compiled language such as Python. As a result, a dynamic library can be updated without thereby inconveniencing clients. Finally, multiple clients can share a single copy of a dynamic library.

In general, dynamic libraries are preferred over static ones, although there is a cost in complexity and performance. Here is a first look at how a library of either type is created and published:

1. The source code for the library is compiled into one or more object modules, which can be packaged as a library and linked to executable clients.
2. The object modules are packaged into a single file. For a static library, the standard extension is *.a* for "archive." For a dynamic library, the extension is *.so* for "shared object." The two sample libraries, which have the same functionality, are published

as the files *libprimes.a* (static) and *libshprimes.so* (dynamic). The prefix *lib* is standard for both types of library.

3. The library file is copied to a standard directory so that client programs, without fuss, can access the library. A typical location for the library, whether static or dynamic, is */usr/lib* or */usr/local/lib*; other locations are possible.

Detailed steps for building and publishing each type of library are coming shortly. First, however, the C functions in the two libraries should be introduced.

8.7.1. The Library Functions

The two sample libraries are built from the same five C functions, four of which are `extern` and, therefore, accessible to client programs. The fifth function, which is a utility for one of the other four, is `static` and thus accessible only to the four `extern` functions defined in the same file. The library functions are elementary and deal, in various ways, with prime numbers. All of the functions expect unsigned (nonnegative) integer values as arguments:

- The `is_prime` function tests whether its single argument is a prime.
- The `are_coprimes` function checks whether its two arguments have a greatest common divisor (gcd) of 1, which defines co-primes.
- The `prime_factors` function lists the prime factors of its argument.

- The `goldbach` function expects an even integer value of 4 or more, listing whichever two primes sum to this argument; there may be multiple summing pairs. The function is named after the 18th-century mathematician Christian Goldbach, whose conjecture that every even integer greater than two is the sum of two primes remains one of the oldest unsolved problems in number theory.

The `static` utility function `gcd`, which the `are_coprimes` function invokes, resides in the deployed library files, but this function is not accessible outside of its containing file; hence, a library client cannot directly invoke the `gcd` function.

8.7.2. Library Source Code and Header File

The header file *primes.h* provides declarations for the four `extern` functions in each library. Such a header file also serves as input for utilities (e.g., the Rust *bindgen* utility) that enable clients in other languages to access a C library. Here is the *primes.h* header file:

```
/** header file primes.h: function declarations **/
extern unsigned is_prime(unsigned);
extern void prime_factors(unsigned);
extern unsigned are_coprimes(unsigned, unsigned);
extern void goldbach(unsigned);
```

As usual, these declarations serve as an interface by specifying the invocation syntax for each function. For client convenience, the text file *primes.h* could be stored in a directory on the C compiler's search path. Typical locations are */usr/include* and */usr/local/include*.

Listing 8-13. The library functions

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

extern unsigned is_prime(unsigned n) {
  if (n <= 3) return n > 1;            /* 2 and 3 are prime */
  if (0 == (n % 2) || 0 == (n % 3)) return 0; /* multiples of 2
  or 3 aren't */

  /* check that n is not a multiple of other values < n */
  unsigned i;
  for (i = 5; (i * i) <= n; i += 6)
    if (0 == (n % i) || 0 == (n % (i + 2))) return 0; /* not
    prime */

  return 1; /* a prime other than 2 or 3 */
}

extern void prime_factors(unsigned n) {
  /* list 2s in n's prime factorization */
  while (0 == (n % 2)) {
    printf("%i ", 2);
    n /= 2;
  }

  /* 2s are done, the divisor is now odd */
  unsigned i;
  for (i = 3; i <= sqrt(n); i += 2) {
    while (0 == (n % i)) {
      printf("%i ", i);
      n /= i;
    }
  }
```

```
  /* one more prime factor? */
  if (n > 2) printf("%i", n);
}

/* utility function: greatest common divisor */
static unsigned gcd(unsigned n1, unsigned n2) {
  while (n1 != 0) {
    unsigned n3 = n1;
    n1 = n2 % n1;
    n2 = n3;
  }
  return n2;
}

extern unsigned are_coprimes(unsigned n1, unsigned n2) {
  return 1 == gcd(n1, n2);
}

extern void goldbach(unsigned n) {
  /* input errors */
  if ((n <= 2) || ((n & 0x01) > 0)) {
    printf("Number must be > 2 and even: %i is not.\n", n);
    return;
  }

  /* two simple cases: 4 and 6 */
  if ((4 == n) || (6 == n)) {
    printf("%i = %i + %i\n", n, n / 2, n / 2);
    return;
  }

  /* for n >= 8: multiple possibilities for many */
  unsigned i;
  for (i = 3; i < (n / 2); i++) {
```

```
    if (is_prime(i) && is_prime(n - i)) {
      printf("%i = %i + %i\n", n, i, n - i);
      /* if one pair is enough, replace this with break */
    }
  }
}
```

The five functions (see Listing 8-13) serve as grist for the library mill. The two libraries derive from exactly the same source code, and the header file *primes.h* is the C interface for both libraries.

8.7.3. Steps for Building the Libraries

The steps for building and then publishing a static and a dynamic library differ in a few details. Only three steps are required for the static library and just two more for the dynamic library. The additional steps in building the dynamic library reflect the added flexibility of the dynamic approach.

The library source file *primes.c* is compiled into an object module. Here is the command, with the percent sign again as the system prompt and with double sharp signs to introduce my comments:

```
% gcc -c primes.c ## step 1 static
```

This produces the binary file *primes.o*, the object module. The flag `-c` means compile only. The next step is to archive the object module(s) by using the Linux *ar* utility:

```
% ar -cvq libprimes.a primes.o ## step 2 static
```

The three flags `-cvq` are short for “create,” “verbose,” and “quick append” in case new files must be added to an archive. The prefix `lib` is standard, but the library name is arbitrary. Of course, the file name for a library must be unique to avoid conflicts.

The archive is ready to be published:

```
% sudo cp libprimes.a /usr/local/lib ## step 3 static
```

The static library is now accessible to clients, examples of which are forthcoming. (The `sudo` is included to ensure the correct access rights for copying a file into */usr/local/lib*.)

The dynamic library also requires one or more object modules for packaging:

```
% gcc primes.c -c -fpic ## step 1 dynamic
```

The added flag `-fpic` directs the compiler to generate position-independent code, which is a binary module that need not be loaded into a fixed memory location. Such flexibility is critical in a system of multiple dynamic libraries. The resulting object module is slightly larger than the one generated for the static library.

Here is the command to create the single library file from the object module(s):

```
% gcc -shared -Wl,-soname,libshprimes.so -o libshprimes.so.1
primes.o ## step 2 dynamic
```

The flag `-shared` indicates that the library is shared (dynamic) rather than static. The `-Wl` flag introduces a list of compiler options, the first of which sets the dynamic library's `soname`, which is required. The `soname` first specifies the library's logical name (*libshprimes.so*) and then, following the `-o` flag, the library's physical file name (*libshprimes.so.1*). The goal is to keep the logical name constant while allowing the physical file name to change with new versions. In this example, the 1 at the end of the physical file name *libshprimes.so.1* represents the first version of the library. The logical and physical file names could be the same, but best practice is to have separate names. A client accesses the library through its logical name (in this case, *libshprimes.so*), as clarified shortly.

The next step is to make the shared library easily accessible to clients by copying it to the appropriate directory, for example, */usr/local/lib* again:

```
% sudo cp libshprimes.so.1 /usr/local/lib ## step 3 dynamic
```

A symbolic link is now set up between the shared library's logical name (*libshprimes.so*) and its full physical file name (*/usr/local/lib/libshprimes.so.1*). Here is the command with */usr/local/lib* as the working directory:

```
% sudo ln --symbolic libshprimes.so.1 libshprimes.so ## step
4 dynamic
```

The logical name *libshprimes.so* should not change, but the target of the symbolic link (*libshprimes.so.1*) can be updated as needed for new library implementations that fix bugs, boost performance, and so on.

The final step (a precautionary one) is to invoke the *ldconfig* utility, which configures the system's dynamic loader. This configuration ensures that the loader will find the newly published library:

```
% sudo ldconfig ## step 5 dynamic
```

The dynamic library is now ready for clients, including the two sample ones that follow.

8.7.4. A Sample C Client

The sample C client is the program *tester*, whose source code begins with two `#include` directives:

```
#include <stdio.h>   /* standard input/output functions */
#include <primes.h>  /* my library functions */
```

Both header files are to be found on the compiler's search path (in the case of *primes.h*, the directory */usr/local/include*). Without this `#include`, the compiler would complain as usual about missing declarations for

functions such as `is_prime` and `prime_factors`. By the way, the source code for the *tester* program need not change at all to test each of the two libraries.

By contrast, the source file for the library (*primes.c*) opens with these `#include` directives:

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

The header file *math.h* is required because the library function `prime_factors` calls the mathematics function `sqrt` from the standard library *libm.so*.

For reference, Listing 8-14 is the source code for the *tester* program.

Listing 8-14. A sample C client

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

int main() {
  /* is_prime */
  printf("\nis_prime\n");
  unsigned i, count = 0, n = 1000;
  for (i = 1; i <= n; i++) {
    if (is_prime(i)) {
      count++;
      if (1 == (i % 100)) printf("Sample prime ending in 1:
      %i\n", i);
    }
  }
  printf("%i primes in range of 1 to a thousand.\n", count);

  /* prime_factors */
  printf("\nprime_factors\n");
```

```
  printf("prime factors of 12: ");
  prime_factors(12);
  printf("\n");

  printf("prime factors of 13: ");
  prime_factors(13);
  printf("\n");

  printf("prime factors of 876,512,779: ");
  prime_factors(876512779);
  printf("\n");

  /* are_coprimes */
  printf("\nare_coprime\n");
  printf("Are %i and %i coprime? %s\n",
         21, 22, are_coprimes(21, 22) ? "yes" : "no");
  printf("Are %i and %i coprime? %s\n",
         21, 24, are_coprimes(21, 24) ? "yes" : "no");

  /* goldbach */
  printf("\ngoldbach\n");
  goldbach(11);  /* error */
  goldbach(4);   /* small one */
  goldbach(6);   /* another */
  for (i = 100; i <= 150; i += 2) goldbach(i);

  return 0;
}
```

In compiling *tester.c* into an executable, the tricky part is the order of the link flags. Recall that the two sample libraries begin with the prefix *lib* and each has the usual extension: *.a* for the static library *libprimes.a* and *.so* for the dynamic library *libshprimes.so*. In a links specification, the prefix *lib* and the extension fall away. A link flag begins with *-l* (lowercase L), and

a compilation command may contain arbitrarily many link flags. Here is the full compilation command for the tester program, using the dynamic library as the example:

```
% gcc -o tester tester.c -lshprimes -lm
```

The first link flag identifies the library *libshprimes.so*, and the second link flag identifies the standard mathematics library *libm.so*.

The linker is lazy, which means that the order of the link flags matters. For example, reversing the order of the link specifications generates a compile-time error:

```
% gcc -o tester tester.c -lm -lshprimes ## DANGER!
```

The flag that links to *libm.so* comes first, but no function from this library is invoked explicitly in the *tester* program; hence, the linker does not link to the *math.so* library. The call to the `sqrt` library function occurs only in the `prime_factors` function from the *libshprimes.so* library. The resulting error in compiling the *tester* program is

```
primes.c: undefined reference to 'sqrt'
```

Accordingly, the order of the link flags should notify the linker that the `sqrt` function is needed:

```
% gcc -o tester tester.c -lshprimes -lm ## -lshprimes 1st
```

The linker picks up the call to the library function `sqrt` in the *libshprimes.so* library and, therefore, does the appropriate link to the mathematics library *libm.so*. There is a more complicated option for linking that supports either link-flag order; in this case, however, it is just as easy to arrange the link flags appropriately.

Here is some output from a run of the *tester* client:

```
is_prime
Sample prime ending in 1: 101
```

```
Sample prime ending in 1: 401
...
168 primes in range of 1 to a thousand.

prime_factors
prime factors of 12: 2 2 3
prime factors of 13: 13
prime factors of 876,512,779: 211 4154089

are_coprime
Are 21 and 22 coprime? yes
Are 21 and 24 coprime? no

goldbach
Number must be > 2 and even: 11 is not.
4 = 2 + 2
6 = 3 + 3
...
32 = 3 + 29
32 = 13 + 19
...
100 = 3 + 97
100 = 11 + 89
...
```

For the `goldbach` function, even a relatively small even value (e.g., 18) may have multiple pairs of primes that sum to it (in this case, 5 + 13 and 7 + 11). Such multiple prime pairs are among the factors that complicate an attempted proof of Goldbach's conjecture.

8.7.5. A Sample Python Client

Python, unlike C, is not a statically compiled language, which means that the sample Python client must access the dynamic rather than the static version of the *primes* library. To do so, Python has various modules (standard and third party) that support a foreign function interface (FFI), which allows a program written in one language to invoke functions written in another. Python *ctypes* is a standard and relatively simple FFI that enables Python code to call C functions.

Any FFI has challenges because the interfacing languages are unlikely to have exactly the same data types. For example, the primes library uses the C type `unsigned int`, which Python does not have; the *ctypes* FFI maps a C `unsigned int` to a Python `int`. Of the four `extern` C functions published in the primes library, two behave better in Python with explicit *ctypes* configuration.

The C functions `prime_factors` and `goldbach` have `void` instead of a return type, but *ctypes* by default replaces the C `void` with the Python `int`. When called from Python code, the two C functions then return a random (hence, meaningless) integer value from the stack. However, *ctypes* can be configured to have the functions return `None` (Python's null type) instead. Here is the configuration for the `prime_factors` function:

```
primes.prime_factors.restype = None
```

A similar statement handles the `goldbach` function.

The following interactive session (in Python3) shows that the interface between a Python client and the *primes* library is straightforward:

```
>>> from ctypes import cdll

>>> primes = cdll.LoadLibrary("libshprimes.so") ## logical name

>>> primes.is_prime(13)
1
```

```
>>> primes.is_prime(12)
0

>>> primes.are_coprimes(8, 24)
0
>>> primes.are_coprimes(8, 25)
1

>>> primes.prime_factors.restype = None
>>> primes.goldbach.restype = None

>>> primes.prime_factors(72)
2 2 2 3 3

>>> primes.goldbach(32)
32 = 3 + 29
32 = 13 + 19
```

The functions in the primes library use only a simple data type, `unsigned int`. If this C library used complicated types such as structures, and if pointers to structures were passed to and returned from library functions, then an FFI more powerful than *ctypes* might be better for a smooth interface between Python and C. Nonetheless, the *ctypes* example shows that a Python client can use a library written in C. Indeed, the popular NumPy library for scientific computing is written in C and then exposed in a high-level Python API.

8.8. What's Next?

This is a small book about a big language—not big in size, but in its impact throughout computing. C is a very small language with easy access to an expanse of standard and third-party libraries. As the libraries get better, C gets better.

C has quirks and presents challenges. Perhaps the greatest challenge is memory leakage: heap storage that the program either allocates explicitly or obtains indirectly through library functions must be freed *explicitly*, and it is easy to allocate—and then forget to deallocate. Better APIs and tools such as *valgrind* (`https://valgrind.org`) address this challenge. The OpenSSL API illustrates best practices: the API includes a family of *free* functions that do whatever *nested* deallocation might be required. C brings the programmer close to the machine, an intimacy that requires particular discipline in code that uses dynamic storage.

Despite its age, C has the look and feel of a modern language with an emphatic separation of concerns: an interface describes, in particular the invocation syntax of functions; an implementation defines by providing the appropriate operational detail. Once published, an interface should remain unchanged, as it represents a contract with programmers; by contrast, an implementation can change to fix bugs, boost performance, and so on.

The standard C library functions are minimalist in design and, therefore, a guide for programmers. Recall the `write` function, which requires three arguments: where to write, what to write, and how many bytes to write. There are no formatting flags or data-type specifications. If these are needed, there are higher-level I/O functions at hand.

C can interact with virtually every other programming language. Is it nonetheless possible that C might lose its role as the *lingua franca* in programming? What would replace C? Its position as the dominant systems language, but one suited for applications as well, makes C the natural language to play this role. Are the standard system libraries, let alone the operating system kernel, to be rewritten in some other language? C combines two features that make it an ideal systems language: C has a high-level syntax that promotes the writing of clear, modular code; but C remains close to the metal, which promotes efficiency.

What, then, is next? The code examples are available from GitHub (https://github.com/mkalin/cbook.git). They are short enough to explore, to tweak, and to improve.

Index

A

B

M. Kalin, *Modern C Up and Running*, https://doi.org/10.1007/978-1-4842-8676-0

C

D

E

F

G

H

I

J, K

L

M

N

O

P

Q

R

S

T

U

V

W, X, Y

Z