21 The Standard Library

Every program is a part of some other program and rarely fits.

In previous chapters we've looked at the C library piecemeal; this chapter focuses on the library as a whole. Section 21.1 lists general guidelines for using the library. It also describes a trick found in some library headers: using a macro to "hide" a function. Section 21.2 gives an overview of each header in the C89 library; Section 21.3 does the same for the new headers in the C99 library.

Later chapters cover the library's headers in depth, with related headers grouped together into chapters. The `<stddef.h>` and `<stdbool.h>` headers are very brief, so I've chosen to discuss them in this chapter (in Sections 21.4 and 21.5, respectively).

21.1 Using the Library

C99

The C89 standard library is divided into 15 parts, with each part described by a header. C99 has an additional nine headers, for a total of 24 (see Table 21.1).

Table 21.1
Standard Library Headers

`<assert.h>`	`<inttypes.h>`†	`<signal.h>`	`<stdlib.h>`
`<complex.h>`†	`<iso646.h>`†	`<stdarg.h>`	`<string.h>`
`<ctype.h>`	`<limits.h>`	`<stdbool.h>`†	`<tgmath.h>`†
`<errno.h>`	`<locale.h>`	`<stddef.h>`	`<time.h>`
`<fenv.h>`†	`<math.h>`	`<stdint.h>`†	`<wchar.h>`†
`<float.h>`	`<setjmp.h>`	`<stdio.h>`	`<wctype.h>`†

†C99 only

Most compilers come with a more extensive library that invariably has many headers that don't appear in Table 21.1. The extra headers aren't standard, of

course, so we can't count on them to be available with other compilers. These headers often provide functions that are specific on a particular computer or operating system (which explains why they're not standard). They may provide functions that allow more control over the screen and keyboard. Headers that support graphics or a window-based user interface are also common.

The standard headers consist primarily of function prototypes, type definitions, and macro definitions. If one of our files contains a call of a function declared in a header or uses one of the types or macros defined there, we'll need to include the header at the beginning of the file. When a file includes several standard headers, the order of `#include` directives doesn't matter. It's also legal to include a standard header more than once.

Restrictions on Names Used in the Library

Any file that includes a standard header must obey a couple of rules. First, it can't use the names of macros defined in that header for any other purpose. If a file includes `<stdio.h>`, for example, it can't reuse `NULL`, since a macro by that name is already defined in `<stdio.h>`. Second, library names with file scope (`typedef` names, in particular) can't be redefined at the file level. Thus, if a file includes `<stdio.h>`, it can't define `size_t` as a identifier with file scope, since `<stdio.h>` defines `size_t` to be a `typedef` name.

Although these restrictions are pretty obvious, C has other restrictions that you might not expect:

- ***Identifiers that begin with an underscore followed by an upper-case letter or a second underscore*** are reserved for use within the library; programs should never use names of this form for any purpose.
- ***Identifiers that begin with an underscore*** are reserved for use as identifiers and tags with file scope. You should never use such a name for your own purposes unless it's declared inside a function.
- ***Every identifier with external linkage in the standard library*** is reserved for use as an identifier with external linkage. In particular, the names of all standard library functions are reserved. Thus, even if a file *doesn't* include `<stdio.h>`, it shouldn't define an external function named `printf`, since there's already a function with this name in the library.

These rules apply to *every* file in a program, regardless of which headers the file includes. Although these rules aren't always enforced, failing to obey them can lead to a program that's not portable.

The rules listed above apply not just to names that are currently used in the library, but also to names that are set aside for future use. The complete description of which names are reserved is rather lengthy; you'll find it in the C standard under "future library directions." As an example, C reserves identifiers that begin with `str` followed by a lower-case letter, so that functions with such names can be added to the `<string.h>` header.

Functions Hidden by Macros

Q&A

It's common for C programmers to replace small functions by parameterized macros. This practice occurs even in the standard library. The C standard allows headers to define macros that have the same names as library functions, but protects the programmer by requiring that a true function be available as well. As a result, it's not unusual for a library header to declare a function *and* define a macro with the same name.

We've already seen an example of a macro duplicating a library function. `getchar` is a library function declared in the `<stdio.h>` header. It has the following prototype:

```
int getchar(void);
```

`<stdio.h>` usually defines `getchar` as a macro as well:

```
#define getchar() getc(stdin)
```

By default, a call of `getchar` will be treated as a macro invocation (since macro names are replaced during preprocessing).

Most of the time, we're happy using a macro instead of a true function, because it will probably make our program run faster. Occasionally, though, we want a genuine function, perhaps to minimize the size of the executable code.

`#undef` directive ➤ 14.3

If the need arises, we can remove a macro definition (thus gaining access to the true function) by using the `#undef` directive. For example, we could undefine the `getchar` macro after including `<stdio.h>`:

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

If `getchar` *isn't* a macro, no harm has been done; `#undef` has no effect when given a name that's not defined as a macro.

As an alternative, we can disable individual uses of a macro by putting parentheses around its name:

```
ch = (getchar)();    /* instead of ch = getchar(); */
```

The preprocessor can't spot a parameterized macro unless its name is followed by a left parenthesis. The compiler isn't so easily fooled, however; it can still recognize `getchar` as a function.

21.2 C89 Library Overview

We'll now take a quick look at the headers in the C89 standard library. This section can serve as a "road map" to help you determine which part of the library you need. Each header is described in detail later in this chapter or in a subsequent chapter.

<assert.h> Diagnostics

<assert.h> header ➤24.1

Contains only the `assert` macro, which allows us to insert self-checks into a program. If any check fails, the program terminates.

<ctype.h> Character Handling

<ctype.h> header ➤23.5

Provides functions for classifying characters and for converting letters from lower to upper case or vice versa.

<errno.h> Errors

<errno.h> header ➤24.2

Provides `errno` ("error number"), an lvalue that can be tested after a call of certain library functions to see if an error occurred during the call.

<float.h> Characteristics of Floating Types

<float.h> header ➤23.1

Provides macros that describe the characteristics of floating types, including their range and accuracy.

<limits.h> Sizes of Integer Types

<limits.h> header ➤23.2

Provides macros that describe the characteristics of integer types (including character types), including their maximum and minimum values.

<locale.h> Localization

<locale.h> header ➤25.1

Provides functions to help a program adapt its behavior to a country or other geographic region. Locale-specific behavior includes the way numbers are printed (such as the character used as the decimal point), the format of monetary values (the currency symbol, for example), the character set, and the appearance of the date and time.

<math.h> Mathematics

<math.h> header ➤23.3

Provides common mathematical functions, including trigonometric, hyperbolic, exponential, logarithmic, power, nearest integer, absolute value, and remainder functions.

<setjmp.h> Nonlocal Jumps

<setjmp.h> header ➤24.4

Provides the `setjmp` and `longjmp` functions. `setjmp` "marks" a place in a program; `longjmp` can then be used to return to that place later. These functions

make it possible to jump from one function into another, still-active function, bypassing the normal function-return mechanism. `setjmp` and `longjmp` are used primarily for handling serious problems that arise during program execution.

`<signal.h>` *Signal Handling*

`<signal.h>` header ➤*24.3*

Provides functions that deal with exceptional conditions (signals), including interrupts and run-time errors. The `signal` function installs a function to be called if a given signal should occur later. The `raise` function causes a signal to occur.

`<stdarg.h>` *Variable Arguments*

`<stdarg.h>` header ➤*26.1*

Provides tools for writing functions that, like `printf` and `scanf`, can have a variable number of arguments.

`<stddef.h>` *Common Definitions*

`<stddef.h>` header ➤*21.4*

Provides definitions of frequently used types and macros.

`<stdio.h>` *Input/Output*

`<stdio.h>` header ➤*22.1–22.8*

Provides a large assortment of input/output functions, including operations on both sequential and random-access files.

`<stdlib.h>` *General Utilities*

`<stdlib.h>` header ➤*26.2*

A "catchall" header for functions that don't fit into any of the other headers. The functions in this header can convert strings to numbers, generate pseudo-random numbers, perform memory management tasks, communicate with the operating system, do searching and sorting, and perform conversions between multibyte characters and wide characters.

`<string.h>` *String Handling*

`<string.h>` header ➤*23.6*

Provides functions that perform string operations, including copying, concatenation, comparison, and searching, as well as functions that operate on arbitrary blocks of memory.

`<time.h>` *Date and Time*

`<time.h>` header ➤*26.3*

Provides functions for determining the time (and date), manipulating times, and formatting times for display.

21.3 C99 Library Changes

Some of the biggest changes in C99 affect the standard library. These changes fall into three groups:

- ***Additional headers.*** The C99 standard library has nine headers that don't exist in C89. Three of these (`<iso646.h>`, `<wchar.h>`, and `<wctype.h>`) were actually added to C in 1995 when the C89 standard was amended. The other six (`<complex.h>`, `<fenv.h>`, `<inttypes.h>`, `<stdbool.h>`, `<stdint.h>`, and `<tgmath.h>`) are new in C99.
- ***Additional macros and functions.*** The C99 standard adds macros and functions to several existing headers, primarily `<float.h>`, `<math.h>`, and `<stdio.h>`. The additions to the `<math.h>` header are so extensive that they're covered in a separate section (Section 23.4).
- ***Enhanced versions of existing functions.*** Some existing functions, including `printf` and `scanf`, have additional capabilities in C99.

We'll now take a quick look at the nine additional headers in the C99 standard library, just as we did in Section 21.2 for the headers in the C89 library.

`<complex.h>` *Complex Arithmetic*

`<complex.h>` header ➤27.4

Defines the `complex` and `I` macros, which are useful when working with complex numbers. Also provides functions for performing mathematical operations on complex numbers.

`<fenv.h>` *Floating-Point Environment*

`<fenv.h>` header ➤27.6

Provides access to floating-point status flags and control modes. For example, a program might test a flag to see if overflow occurred during a floating-point operation or set a control mode to specify how rounding should be done.

`<inttypes.h>` *Format Conversion of Integer Types*

`<inttypes.h>` header ➤27.2

Defines macros that can be used in format strings for input/output of the integer types declared in `<stdint.h>`. Also provides functions for working with greatest-width integers.

`<iso646.h>` *Alternative Spellings*

`<iso646.h>` header ➤25.3

Defines macros that represent certain operators (the ones containing the characters `&`, `|`, `~`, `!`, and `^`). These macros are useful for writing programs in an environment where these characters might not be part of the local character set.

<stdbool.h> Boolean Type and Values

<stdbool.h> header ➤21.5

Defines the bool, true, and false macros, as well as a macro that can be used to test whether these macros have been defined.

<stdint.h> Integer Types

<stdint.h> header ➤27.1

Declares integer types with specified widths and defines related macros (such as macros that specify the maximum and minimum values of each type). Also defines parameterized macros that construct integer constants with specific types.

<tgmath.h> Type-Generic Math

<tgmath.h> header ➤27.5

In C99, there are multiple versions of many math functions in the <math.h> and <complex.h> headers. The "type-generic" macros in <tgmath.h> can detect the types of the arguments passed to them and substitute a call of the appropriate <math.h> or <complex.h> function.

<wchar.h> Extended Multibyte and Wide-Character Utilities

<wchar.h> header ➤25.5

Provides functions for wide-character input/output and wide string manipulation.

<wctype.h> Wide-Character Classification and Mapping Utilities

<wctype.h> header ➤25.6

The wide-character version of <ctype.h>. Provides functions for classifying and changing the case of wide characters.

21.4 The <stddef.h> Header: Common Definitions

The <stddef.h> header provides definitions of frequently used types and macros; it doesn't declare any functions. The types are:

- ptrdiff_t. The type of the result when two pointers are subtracted.
- size_t. The type returned by the sizeof operator.
- wchar_t. A type large enough to represent all possible characters in all supported locales.

All three are names for integer types; ptrdiff_t must be a signed type, while size_t must be an unsigned type. For more information about wchar_t, see Section 25.2.

The <stddef.h> header also defines two macros. One of them is NULL, which represents the null pointer. The other macro, offsetof, requires two arguments: *type* (a structure type) and *member-designator* (a member of the structure).

`offsetof` computes the number of bytes between the beginning of the structure and the specified member.

Consider the following structure:

```
struct s {
  char a;
  int b[2];
  float c;
};
```

The value of `offsetof(struct s, a)` must be 0; C guarantees that the first member of a structure has the same address as the structure itself. We can't say for sure what the offsets of `b` and `c` are. One possibility is that `offsetof(struct s, b)` is 1 (since `a` is one byte long), and `offsetof(struct s, c)` is 9 (assuming 32-bit integers). However, some compilers leave "holes"—unused bytes—in structures (see the Q&A section at the end of Chapter 16), which can affect the value produced by `offsetof`. If a compiler should leave a three-byte hole after `a`, for example, then the offsets of `b` and `c` would be 4 and 12, respectively. But that's the beauty of `offsetof`: it produces the correct offsets for any compiler, enabling us to write portable programs.

There are various uses for `offsetof`. For example, suppose that we want to save the first two members of an `s` structure in a file, ignoring the `c` member. Instead of having the `fwrite` function write `sizeof(struct s)` bytes, which would save the entire structure, we'll tell it to write only `offsetof(struct s, c)` bytes.

`fwrite` function ➤22.6

A final remark: Some of the types and macros defined in `<stddef.h>` appear in other headers as well. (The `NULL` macro, for example, is also defined in `<locale.h>`, `<stdio.h>`, `<stdlib.h>`, `<string.h>`, and `<time.h>`, as well as in the C99 header `<wchar.h>`.) As a result, few programs need to include `<stddef.h>`.

21.5 The `<stdbool.h>` Header (C99): Boolean Type and Values

The `<stdbool.h>` header defines four macros:

- `bool` (defined to be `_Bool`)
- `true` (defined to be `1`)
- `false` (defined to be `0`)
- `__bool_true_false_are_defined` (defined to be `1`)

We've seen many examples of how `bool`, `true`, and `false` are used. Potential uses of the `__bool_true_false_are_defined` macro are more limited. A program could use a preprocessing directive (such as `#if` or `#ifdef`) to test this macro before attempting to define its own version of `bool`, `true`, or `false`.

Q & A

Q: I notice that you use the term "standard header" rather than "standard header file." Is there any reason for not using the word "file"?

A: Yes. According to the C standard, a "standard header" need not be a file. Although most compilers do indeed store standard headers as files, the headers could in fact be built into the compiler itself.

Q: Section 14.3 described some disadvantages of using parameterized macros in place of functions. In light of these problems, isn't it dangerous to provide a macro substitute for a standard library function? [p. 531]

A: According to the C standard, a parameterized macro that substitutes for a library function must be "fully protected" by parentheses and must evaluate its arguments exactly once. These rules avoid most of the problems mentioned in Section 14.3.

Exercises

Section 21.1

1. Locate where header files are kept on your system. Find the nonstandard headers and determine the purpose of each.

2. Having located the header files on your system (see Exercise 1), find a standard header in which a macro hides a function.

3. When a macro hides a function, which must come first in the header file: the macro definition or the function prototype? Justify your answer.

4. Make a list of all reserved identifiers in the "future library directions" section of the C99 standard. Distinguish between identifiers that are reserved for use only when a specific header is included versus identifiers that are reserved for use as external names.

*5. The `islower` function, which belongs to `<ctype.h>`, tests whether a character is a lower-case letter. Why would the following macro version of `islower` not be legal, according to the C standard? (You may assume that the character set is ASCII.)

```
#define islower(c) ((c) >= 'a' && (c) <= 'z')
```

6. The `<ctype.h>` header usually defines most of its functions as macros as well. These macros rely on a static array that's declared in `<ctype.h>` but defined in a separate file. A portion of a typical `<ctype.h>` header appears below. Use this sample to answer the following questions.

   (a) Why do the names of the "bit" macros (such as `_UPPER`) and the `_ctype` array begin with an underscore?

   (b) Explain what the `_ctype` array will contain. Assuming that the character set is ASCII, show the values of the array elements at positions 9 (the horizontal tab character), 32 (the space character), 65 (the letter `A`), and 94 (the `^` character). See Section 23.5 for a description of what each macro should return.

(c) What's the advantage of using an array to implement these macros?

```
#define _UPPER   0x01   /* upper-case letter */
#define _LOWER   0x02   /* lower-case letter */
#define _DIGIT   0x04   /* decimal digit */
#define _CONTROL 0x08   /* control character */
#define _PUNCT   0x10   /* punctuation character */
#define _SPACE   0x20   /* white-space character */
#define _HEX     0x40   /* hexadecimal digit */
#define _BLANK   0x80   /* space character */

#define isalnum(c)  (_ctype[c] & (_UPPER|_LOWER|_DIGIT))
#define isalpha(c)  (_ctype[c] & (_UPPER|_LOWER))
#define iscntrl(c)  (_ctype[c] & _CONTROL)
#define isdigit(c)  (_ctype[c] & _DIGIT)
#define isgraph(c)  (_ctype[c] &
                      (_PUNCT|_UPPER|_LOWER|_DIGIT))
#define islower(c)  (_ctype[c] & _LOWER)
#define isprint(c)  (_ctype[c] &
                      (_BLANK|_PUNCT|_UPPER|_LOWER|_DIGIT))
#define ispunct(c)  (_ctype[c] & _PUNCT)
#define isspace(c)  (_ctype[c] & _SPACE)
#define isupper(c)  (_ctype[c] & _UPPER)
#define isxdigit(c) (_ctype[c] & (_DIGIT|_HEX))
```

Section 21.2 W 7. In which standard header would you expect to find each of the following?

(a) A function that determines the current day of the week
(b) A function that tests whether a character is a digit
(c) A macro that gives the largest `unsigned int` value
(d) A function that rounds a floating-point number to the next higher integer
(e) A macro that specifies the number of bits in a character
(f) A macro that specifies the number of significant digits in a `double` value
(g) A function that searches a string for a particular character
(h) A function that opens a file for reading

Programming Projects

1. Write a program that declares the `s` structure (see Section 21.4) and prints the sizes and offsets of the `a`, `b`, and `c` members. (Use `sizeof` to find sizes; use `offsetof` to find offsets.) Have the program print the size of the entire structure as well. From this information, determine whether or not the structure has any holes. If it does, describe the location and size of each.

22 Input/Output

In man-machine symbiosis, it is man who must adjust: The machines can't.

C's input/output library is the biggest and most important part of the standard library. As befits its lofty status, we'll devote an entire chapter (the longest in the book) to the `<stdio.h>` header, the primary repository of input/output functions.

We've been using `<stdio.h>` since Chapter 2, and we have experience with the `printf`, `scanf`, `putchar`, `getchar`, `puts`, and `gets` functions. This chapter provides more information about these six functions, as well as introducing a host of new functions, most of which deal with files. Fortunately, many of the new functions are closely related to functions with which we're already acquainted. `fprintf`, for instance, is the "file version" of the `printf` function.

We'll start the chapter with a discussion of some basic issues: the stream concept, the `FILE` type, input and output redirection, and the difference between text files and binary files (Section 22.1). We'll then turn to functions that are designed specifically for use with files, including functions that open and close files (Section 22.2). After covering `printf`, `scanf`, and related functions for "formatted" input/output (Section 22.3), we'll look at functions that read and write unformatted data:

- `getc`, `putc`, and related functions, which read and write one *character* at a time (Section 22.4).
- `gets`, `puts`, and related functions, which read and write one *line* at a time (Section 22.5).
- `fread` and `fwrite`, which read and write *blocks* of data (Section 22.6).

Section 22.7 then shows how to perform random access operations on files. Finally, Section 22.8 describes the `sprintf`, `snprintf`, and `sscanf` functions, variants of `printf` and `scanf` that write to a string or read from a string.

This chapter covers all but eight of the functions in `<stdio.h>`. One of these eight, the `perror` function, is closely related to the `<errno.h>` header, so

I'll postpone it until Section 24.2, which discusses that header. Section 26.1 covers the remaining functions (`vfprintf`, `vprintf`, `vsprintf`, `vsnprintf`, `vfscanf`, `vscanf`, and `vsscanf`). These functions rely on the `va_list` type, which is introduced in that section.

C99

`<wchar.h>` header ➤25.5

In C89, all standard input/output functions belong to `<stdio.h>`, but such is not the case in C99, where some I/O functions are declared in the `<wchar.h>` header. The `<wchar.h>` functions deal with wide characters rather than ordinary characters; the good news is that most of these functions closely resemble those of `<stdio.h>`. Functions in `<stdio.h>` that read or write data are known as ***byte input/output functions;*** similar functions in `<wchar.h>` are called ***wide-character input/output functions.***

22.1 Streams

In C, the term ***stream*** means any source of input or any destination for output. Many small programs, like the ones in previous chapters, obtain all their input from one stream (usually associated with the keyboard) and write all their output to another stream (usually associated with the screen).

Larger programs may need additional streams. These streams often represent files stored on various media (such as hard drives, CDs, DVDs, and flash memory), but they could just as easily be associated with devices that don't store files: network ports, printers, and the like. We'll concentrate on files, since they're common and easy to understand. (I may even occasionally use the term *file* when I should say *stream*.) Keep in mind, however, that many of the functions in `<stdio.h>` work equally well with all streams, not just the ones that represent files.

File Pointers

Accessing a stream in a C program is done through a ***file pointer***, which has type `FILE *` (the `FILE` type is declared in `<stdio.h>`). Certain streams are represented by file pointers with standard names; we can declare additional file pointers as needed. For example, if a program needs two streams in addition to the standard ones, it might contain the following declaration:

```
FILE *fp1, *fp2;
```

A program may declare any number of `FILE *` variables, although operating systems usually limit the number of streams that can be open at one time.

Standard Streams and Redirection

`<stdio.h>` provides three standard streams (Table 22.1). These streams are ready to use—we don't declare them, and we don't open or close them.

Table 22.1
Standard Streams

File Pointer	*Stream*	*Default Meaning*
`stdin`	Standard input	Keyboard
`stdout`	Standard output	Screen
`stderr`	Standard error	Screen

The functions that we've used in previous chapters—`printf`, `scanf`, `putchar`, `getchar`, `puts`, and `gets`—obtain input from `stdin` and send output to `stdout`. By default, `stdin` represents the keyboard; `stdout` and `stderr` represent the screen. However, many operating systems allow these default meanings to be changed via a mechanism known as ***redirection***.

Q&A

Typically, we can force a program to obtain its input from a file instead of from the keyboard by putting the name of the file on the command line, preceded by the < character:

```
demo <in.dat
```

This technique, known as ***input redirection***, essentially makes the `stdin` stream represent a file (`in.dat`, in this case) instead of the keyboard. The beauty of redirection is that the `demo` program doesn't realize that it's reading from `in.dat`; as far as it knows, any data it obtains from `stdin` is being entered at the keyboard.

Output redirection is similar. Redirecting the `stdout` stream is usually done by putting a file name on the command line, preceded by the > character:

```
demo >out.dat
```

All data written to `stdout` will now go into the `out.dat` file instead of appearing on the screen. Incidentally, we can combine output redirection with input redirection:

```
demo <in.dat >out.dat
```

The < and > characters don't have to be adjacent to file names, and the order in which the redirected files are listed doesn't matter, so the following examples would work just as well:

```
demo < in.dat > out.dat
demo >out.dat <in.dat
```

One problem with output redirection is that *everything* written to `stdout` is put into a file. If the program goes off the rails and begins writing error messages, we won't see them until we look at the file. This is where `stderr` comes in. By writing error messages to `stderr` instead of `stdout`, we can guarantee that those messages will appear on the screen even when `stdout` has been redirected. (Operating systems often allow `stderr` itself to be redirected, though.)

Text Files versus Binary Files

`<stdio.h>` supports two kinds of files: text and binary. The bytes in a ***text file*** represent characters, making it possible for a human to examine the file or edit it.

The source code for a C program is stored in a text file, for example. In a ***binary file***, on the other hand, bytes don't necessarily represent characters; groups of bytes might represent other types of data, such as integers and floating-point numbers. An executable C program is stored in a binary file, as you'll quickly realize if you try to look at the contents of one.

Text files have two characteristics that binary files don't possess:

- ***Text files are divided into lines.*** Each line in a text file normally ends with one or two special characters; the choice of characters depends on the operating system. In Windows, the end-of-line marker is a carriage-return character (`'\x0d'`) followed immediately by a line-feed character (`'\x0a'`). In UNIX and newer versions of the Macintosh operating system (Mac OS), the end-of-line marker is a single line-feed character. Older versions of Mac OS use a single carriage-return character.

Q&A

- ***Text files may contain a special "end-of-file" marker.*** Some operating systems allow a special byte to be used as a marker at the end of a text file. In Windows, the marker is `'\x1a'` (Ctrl-Z). There's no requirement that Ctrl-Z be present, but if it is, it marks the end of the file; any bytes after Ctrl-Z are to be ignored. The Ctrl-Z convention is a holdover from DOS, which in turn inherited it from CP/M, an early operating system for personal computers. Most other operating systems, including UNIX, have no special end-of-file character.

Binary files aren't divided into lines. In a binary file, there are no end-of-line or end-of-file markers; all bytes are treated equally.

Q&A

When we write data to a file, we'll need to consider whether to store it in text form or in binary form. To see the difference, consider how we might store the number 32767 in a file. One option would be to write the number in text form as the characters 3, 2, 7, 6, and 7. If the character set is ASCII, we'd have the following five bytes:

00110011	00110010	00110111	00110110	00110111
'3'	'2'	'7'	'6'	'7'

The other option is to store the number in binary, which would take as few as two bytes:

01111111	11111111

little-endian order ➤20.3

(The bytes will be reversed on systems that store data in little-endian order.) As this example shows, storing numbers in binary can often save quite a bit of space.

When we're writing a program that reads from a file or writes to a file, we need to take into account whether it's a text file or a binary file. A program that displays the contents of a file on the screen will probably assume it's a text file. A file-

copying program, on the other hand, can't assume that the file to be copied is a text file. If it does, binary files containing an end-of-file character won't be copied completely. When we can't say for sure whether a file is text or binary, it's safer to assume that it's binary.

22.2 File Operations

Simplicity is one of the attractions of input and output redirection; there's no need to open a file, close a file, or perform any other explicit file operations. Unfortunately, redirection is too limited for many applications. When a program relies on redirection, it has no control over its files; it doesn't even know their names. Worse still, redirection doesn't help if the program needs to read from two files or write to two files at the same time.

When redirection isn't enough, we'll end up using the file operations that `<stdio.h>` provides. In this section, we'll explore these operations, which include opening a file, closing a file, changing the way a file is buffered, deleting a file, and renaming a file.

Opening a File

```
FILE *fopen(const char * restrict filename,
            const char * restrict mode);
```

fopen

Opening a file for use as a stream requires a call of the `fopen` function. `fopen`'s first argument is a string containing the name of the file to be opened. (A "file name" may include information about the file's location, such as a drive specifier or path.) The second argument is a "mode string" that specifies what operations we intend to perform on the file. The string `"r"`, for instance, indicates that data will be read from the file, but none will be written to it.

restrict keyword ➤17.8

C99

Note that `restrict` appears twice in the prototype for the `fopen` function. `restrict`, which is a C99 keyword, indicates that `filename` and `mode` should point to strings that don't share memory locations. The C89 prototype for `fopen` doesn't contain `restrict` but is otherwise identical. `restrict` has no effect on the behavior of `fopen`, so it can usually just be ignored. In this and subsequent chapters, I'll italicize `restrict` as a reminder that it's a C99 feature.

escape sequences ➤7.3

Windows programmers: Be careful when the file name in a call of `fopen` includes the \ character, since C treats \ as the beginning of an escape sequence. The call

```
fopen("c:\project\test1.dat", "r")
```

will fail, because the compiler treats `\t` as a character escape. (`\p` isn't a valid character escape, but it looks like one. The C standard states that its meaning is

undefined.) There are two ways to avoid the problem. One is to use \\ instead of \:

```
fopen("c:\\project\\test1.dat", "r")
```

The other technique is even easier—just use the / character instead of \:

```
fopen("c:/project/test1.dat", "r")
```

Windows will happily accept / instead of \ as the directory separator.

`fopen` returns a file pointer that the program can (and usually will) save in a variable and use later whenever it needs to perform an operation on the file. Here's a typical call of `fopen`, where `fp` is a variable of type `FILE *`:

```
fp = fopen("in.dat", "r");   /* opens in.dat for reading */
```

When the program calls an input function to read from `in.dat` later, it will supply `fp` as an argument.

When it can't open a file, `fopen` returns a null pointer. Perhaps the file doesn't exist, or it's in the wrong place, or we don't have permission to open it.

⚠ Never assume that a file can be opened; always test the return value of `fopen` to make sure it's not a null pointer.

Modes

Which mode string we'll pass to `fopen` depends not only on what operations we plan to perform on the file later but also on whether the file contains text or binary data. To open a text file, we'd use one of the mode strings in Table 22.2.

Table 22.2 Mode Strings for Text Files

String	*Meaning*
"r"	Open for reading
"w"	Open for writing (file need not exist)
"a"	Open for appending (file need not exist)
"r+"	Open for reading and writing, starting at beginning
"w+"	Open for reading and writing (truncate if file exists)
"a+"	Open for reading and writing (append if file exists)

Q&A When we use `fopen` to open a binary file, we'll need to include the letter `b` in the mode string. Table 22.3 lists mode strings for binary files.

From Tables 22.2 and 22.3, we see that `<stdio.h>` distinguishes between *writing* data and *appending* data. When data is written to a file, it normally overwrites what was previously there. When a file is opened for appending, however, data written to the file is added at the end, thus preserving the file's original contents.

By the way, special rules apply when a file is opened for both reading and writing (the mode string contains the + character). We can't switch from reading to writ-

Table 22.3
Mode Strings for Binary Files

String	*Meaning*
`"rb"`	Open for reading
`"wb"`	Open for writing (file need not exist)
`"ab"`	Open for appending (file need not exist)
`"r+b"` or `"rb+"`	Open for reading and writing, starting at beginning
`"w+b"` or `"wb+"`	Open for reading and writing (truncate if file exists)
`"a+b"` or `"ab+"`	Open for reading and writing (append if file exists)

file-positioning functions ➤22.7

ing without first calling a file-positioning function unless the reading operation encountered the end of the file. Also, we can't switch from writing to reading without either calling `fflush` (covered later in this section) or calling a file-positioning function.

Closing a File

```
int fclose(FILE *stream);
```

fclose

The `fclose` function allows a program to close a file that it's no longer using. The argument to `fclose` must be a file pointer obtained from a call of `fopen` or `freopen` (discussed later in this section). `fclose` returns zero if the file was closed successfully; otherwise, it returns the error code `EOF` (a macro defined in `<stdio.h>`).

Q&A

To show how `fopen` and `fclose` are used in practice, here's the outline of a program that opens the file `example.dat` for reading, checks that it was opened successfully, then closes it before terminating:

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

#define FILE_NAME "example.dat"

int main(void)
{
  FILE *fp;

  fp = fopen(FILE_NAME, "r");
  if (fp == NULL) {
    printf("Can't open %s\n", FILE_NAME);
    exit(EXIT_FAILURE);
  }
  ...
  fclose(fp);
  return 0;
}
```

Of course, C programmers being the way they are, it's not unusual to see the call of `fopen` combined with the declaration of `fp`:

```
FILE *fp = fopen(FILE_NAME, "r");
```

or the test against NULL:

```
if ((fp = fopen(FILE_NAME, "r")) == NULL) ...
```

Attaching a File to an Open Stream

```
FILE *freopen(const char * restrict filename,
              const char * restrict mode,
              FILE * restrict stream);
```

freopen

`freopen` attaches a different file to a stream that's already open. The most common use of `freopen` is to associate a file with one of the standard streams (`stdin`, `stdout`, or `stderr`). To cause a program to begin writing to the file `foo`, for instance, we could use the following call of `freopen`:

```
if (freopen("foo", "w", stdout) == NULL) {
  /* error; foo can't be opened */
}
```

After closing any file previously associated with `stdout` (by command-line redirection or a previous call of `freopen`), `freopen` will open `foo` and associate it with `stdout`.

`freopen`'s normal return value is its third argument (a file pointer). If it can't open the new file, `freopen` returns a null pointer. (`freopen` ignores the error if the old file can't be closed.)

C99

C99 adds a new twist. If `filename` is a null pointer, `freopen` attempts to change the stream's mode to that specified by the `mode` parameter. Implementations aren't required to support this feature, however; if they do, they may place restrictions on which mode changes are permitted.

Obtaining File Names from the Command Line

When we're writing a program that will need to open a file, one problem soon becomes apparent: how do we supply the file name to the program? Building file names into the program itself doesn't provide much flexibility, and prompting the user to enter file names can be awkward. Often, the best solution is to have the program obtain file names from the command line. When we execute a program named `demo`, for example, we might supply it with file names by putting them on the command line:

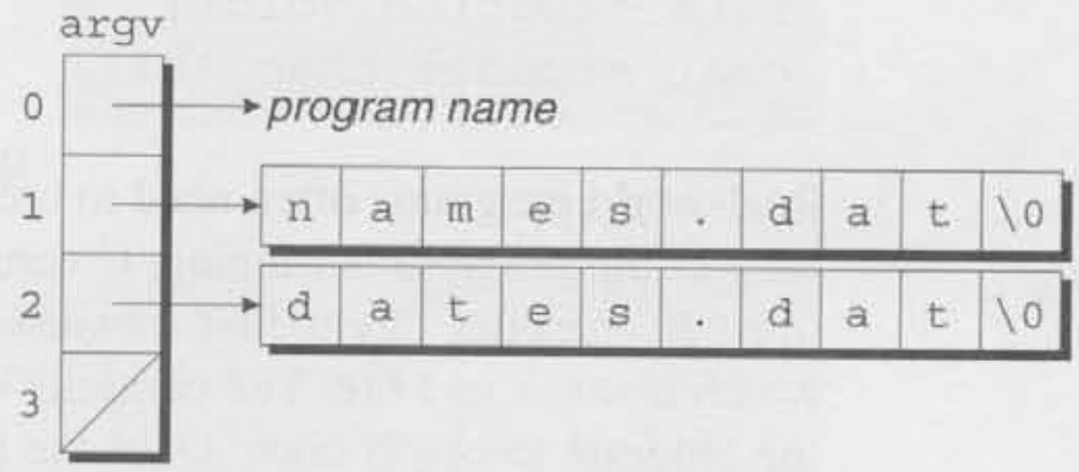
Q&A

```
demo names.dat dates.dat
```

In Section 13.7, we saw how to access command-line arguments by defining `main` as a function with two parameters:

```
int main(int argc, char *argv[])
{
  ...
}
```

`argc` is the number of command-line arguments; `argv` is an array of pointers to the argument strings. `argv[0]` points to the program name, `argv[1]` through `argv[argc-1]` point to the remaining arguments, and `argv[argc]` is a null pointer. In the example above, `argc` is 3, `argv[0]` points to a string containing the program name, `argv[1]` points to the string `"names.dat"`, and `argv[2]` points to the string `"dates.dat"`:



PROGRAM

Checking Whether a File Can Be Opened

The following program determines if a file exists and can be opened for reading. When the program is run, the user will give it a file name to check:

`canopen` *file*

The program will then print either *file* `can be opened` or *file* `can't be opened`. If the user enters the wrong number of arguments on the command line, the program will print the message `usage: canopen filename` to remind the user that `canopen` requires a single file name.

canopen.c

```
/* Checks whether a file can be opened for reading */

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

int main(int argc, char *argv[])
{
  FILE *fp;

  if (argc != 2) {
    printf("usage: canopen filename\n");
    exit(EXIT_FAILURE);
  }

  if ((fp = fopen(argv[1], "r")) == NULL) {
    printf("%s can't be opened\n", argv[1]);
    exit(EXIT_FAILURE);
  }

  printf("%s can be opened\n", argv[1]);
  fclose(fp);
  return 0;
}
```

Note that we can use redirection to discard the output of `canopen` and simply test the status value it returns.

Temporary Files

```
FILE *tmpfile(void);
char *tmpnam(char *s);
```

Real-world programs often need to create temporary files—files that exist only as long as the program is running. C compilers, for instance, often create temporary files. A compiler might first translate a C program to some intermediate form, which it stores in a file. The compiler would then read the file later as it translates the program to object code. Once the program is completely compiled, there's no need to preserve the file containing the program's intermediate form. `<stdio.h>` provides two functions, `tmpfile` and `tmpnam`, for working with temporary files.

tmpfile

`tmpfile` creates a temporary file (opened in `"wb+"` mode) that will exist until it's closed or the program ends. A call of `tmpfile` returns a file pointer that can be used to access the file later:

```
FILE *tempptr;
...
tempptr = tmpfile();    /* creates a temporary file */
```

If it fails to create a file, `tmpfile` returns a null pointer.

Although `tmpfile` is easy to use, it has a couple of drawbacks: (1) we don't know the name of the file that `tmpfile` creates, and (2) we can't decide later to make the file permanent. If these restrictions turn out to be a problem, the alternative is to create a temporary file using `fopen`. Of course, we don't want this file to have the same name as a previously existing file, so we need some way to generate new file names; that's where the `tmpnam` function comes in.

tmpnam

`tmpnam` generates a name for a temporary file. If its argument is a null pointer, `tmpnam` stores the file name in a static variable and returns a pointer to it:

```
char *filename;
...
filename = tmpnam(NULL);    /* creates a temporary file name */
```

Otherwise, `tmpnam` copies the file name into a character array provided by the programmer:

```
char filename[L_tmpnam];
...
tmpnam(filename);           /* creates a temporary file name */
```

In the latter case, `tmpnam` also returns a pointer to the first character of this array. `L_tmpnam` is a macro in `<stdio.h>` that specifies how long to make a character array that will hold a temporary file name.

Be sure that `tmpnam`'s argument points to an array of at least `L_tmpnam` characters. Also, be careful not to call `tmpnam` too often; the `TMP_MAX` macro (defined in `<stdio.h>`) specifies the maximum number of temporary file names that can potentially be generated by `tmpnam` during the execution of a program. If it fails to generate a file name, `tmpnam` returns a null pointer.

File Buffering

```
int fflush(FILE *stream);
void setbuf(FILE * restrict stream,
            char * restrict buf);
int setvbuf(FILE * restrict stream,
            char * restrict buf,
            int mode, size_t size);
```

Transferring data to or from a disk drive is a relatively slow operation. As a result, it isn't feasible for a program to access a disk file directly each time it wants to read or write a byte. The secret to achieving acceptable performance is ***buffering:*** data written to a stream is actually stored in a buffer area in memory; when it's full (or the stream is closed), the buffer is "flushed" (written to the actual output device). Input streams can be buffered in a similar way: the buffer contains data from the input device; input is read from this buffer instead of the device itself. Buffering can result in enormous gains in efficiency, since reading a byte from a buffer or storing a byte in a buffer takes hardly any time at all. Of course, it takes time to transfer the buffer contents to or from disk, but one large "block move" is much faster than many tiny byte moves.

The functions in `<stdio.h>` perform buffering automatically when it seems advantageous. The buffering takes place behind the scenes, and we usually don't worry about it. On rare occasions, though, we may need to take a more active role. If so, we can use the functions `fflush`, `setbuf`, and `setvbuf`.

fflush

When a program writes output to a file, the data normally goes into a buffer first. The buffer is flushed automatically when it's full or the file is closed. By calling `fflush`, however, a program can flush a file's buffer as often as it wishes. The call

Q&A

```
fflush(fp);      /* flushes buffer for fp */
```

flushes the buffer for the file associated with `fp`. The call

```
fflush(NULL);    /* flushes all buffers */
```

flushes *all* output streams. `fflush` returns zero if it's successful and `EOF` if an error occurs.

setvbuf

`setvbuf` allows us to change the way a stream is buffered and to control the size and location of the buffer. The function's third argument, which specifies the kind of buffering desired, should be one of the following macros:

- `_IOFBF` (full buffering). Data is read from the stream when the buffer is empty or written to the stream when it's full.
- `_IOLBF` (line buffering). Data is read from the stream or written to the stream one line at a time.
- `_IONBF` (no buffering). Data is read from the stream or written to the stream directly, without a buffer.

(All three macros are defined in `<stdio.h>`.) Full buffering is the default for streams that aren't connected to interactive devices.

`setvbuf`'s second argument (if it's not a null pointer) is the address of the desired buffer. The buffer might have static storage duration, automatic storage duration, or even be allocated dynamically. Making the buffer automatic allows its space to be reclaimed automatically at block exit; allocating it dynamically enables us to free the buffer when it's no longer needed. `setvbuf`'s last argument is the number of bytes in the buffer. A larger buffer may give better performance; a smaller buffer saves space.

For example, the following call of `setvbuf` changes the buffering of `stream` to full buffering, using the `N` bytes in the `buffer` array as the buffer:

```
char buffer[N];
...
setvbuf(stream, buffer, _IOFBF, N);
```

`setvbuf` must be called after `stream` is opened but before any other operations are performed on it.

It's also legal to call `setvbuf` with a null pointer as the second argument, which requests that `setvbuf` create a buffer with the specified size. `setvbuf` returns zero if it's successful. It returns a nonzero value if the `mode` argument is invalid or the request can't be honored.

setbuf

`setbuf` is an older function that assumes default values for the buffering mode and buffer size. If `buf` is a null pointer, the call `setbuf(stream, buf)` is equivalent to

```
(void) setvbuf(stream, NULL, _IONBF, 0);
```

Otherwise, it's equivalent to

```
(void) setvbuf(stream, buf, _IOFBF, BUFSIZ);
```

where `BUFSIZ` is a macro defined in `<stdio.h>`. The `setbuf` function is considered obsolete; it's not recommended for use in new programs.

When using `setvbuf` or `setbuf`, be sure to close the stream before its buffer is deallocated. In particular, if the buffer is local to a function and has automatic storage duration, be sure to close the stream before the function returns.

Miscellaneous File Operations

```
int remove(const char *filename);
int rename(const char *old, const char *new);
```

The functions `remove` and `rename` allow a program to perform basic file management operations. Unlike most other functions in this section, `remove` and `rename` work with file *names* instead of file *pointers*. Both functions return zero if they succeed and a nonzero value if they fail.

remove

`remove` deletes a file:

```
remove("foo");          /* deletes the file named "foo" */
```

If a program uses `fopen` (instead of `tmpfile`) to create a temporary file, it can use `remove` to delete the file before the program terminates. Be sure that the file to be removed has been closed; the effect of removing a file that's currently open is implementation-defined.

rename

`rename` changes the name of a file:

```
rename("foo", "bar");   /* renames "foo" to "bar" */
```

`rename` is handy for renaming a temporary file created using `fopen` if a program should decide to make it permanent. If a file with the new name already exists, the effect is implementation-defined.

If the file to be renamed is open, be sure to close it before calling `rename`; the function may fail if asked to rename an open file.

22.3 Formatted I/O

In this section, we'll examine library functions that use format strings to control reading and writing. These functions, which include our old friends `printf` and `scanf`, have the ability to convert data from character form to numeric form during input and from numeric form to character form during output. None of the other I/O functions can do such conversions.

The ...printf Functions

```
int fprintf(FILE * restrict stream,
            const char * restrict format, ...);
int printf(const char * restrict format, ...);
```

fprintf
printf
ellipsis ➤26.1

The `fprintf` and `printf` functions write a variable number of data items to an output stream, using a format string to control the appearance of the output. The prototypes for both functions end with the `...` symbol (an ***ellipsis***), which indicates a variable number of additional arguments. Both functions return the number of characters written; a negative return value indicates that an error occurred.

The only difference between `printf` and `fprintf` is that `printf` always writes to `stdout` (the standard output stream), whereas `fprintf` writes to the stream indicated by its first argument:

```
printf("Total: %d\n", total);          /* writes to stdout */
fprintf(fp, "Total: %d\n", total);     /* writes to fp */
```

A call of `printf` is equivalent to a call of `fprintf` with `stdout` as the first argument.

Don't think of `fprintf` as merely a function that writes data to disk files, though. Like many functions in `<stdio.h>`, `fprintf` works fine with any output stream. In fact, one of the most common uses of `fprintf`—writing error messages to `stderr`, the standard error stream—has nothing to do with disk files. Here's what such a call might look like:

```
fprintf(stderr, "Error: data file can't be opened.\n");
```

Writing the message to `stderr` guarantees that it will appear on the screen even if the user redirects `stdout`.

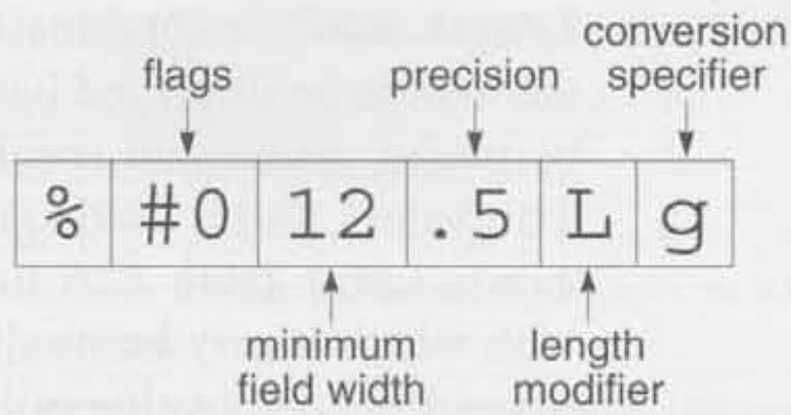
v...printf functions ➤26.1

There are two other functions in `<stdio.h>` that can write formatted output to a stream. These functions, named `vfprintf` and `vprintf`, are fairly obscure. Both rely on the `va_list` type, which is declared in `<stdarg.h>`, so they're discussed along with that header.

...printf Conversion Specifications

Both `printf` and `fprintf` require a format string containing ordinary characters and/or conversion specifications. Ordinary characters are printed as is; conversion specifications describe how the remaining arguments are to be converted to character form for display. Section 3.1 described conversion specifications briefly, and we added more details in later chapters. We'll now review what we know about conversion specifications and fill in the remaining gaps.

A ...`printf` conversion specification consists of the `%` character, followed by as many as five distinct items:



Here's a detailed description of these items, which must appear in the order shown:

- ***Flags*** (optional; more than one permitted). The - flag causes left justification within a field; the other flags affect the way numbers are displayed. Table 22.4 gives a complete list of flags.

Table 22.4 Flags for ...printf Functions

Flag	*Meaning*
-	Left-justify within field. (The default is right justification.)
+	Numbers produced by signed conversions always begin with + or -. (Normally, only negative numbers are preceded by a sign.)
space	Nonnegative numbers produced by signed conversions are preceded by a space. (The + flag overrides the *space* flag.)
#	Octal numbers begin with 0, nonzero hexadecimal numbers with 0x or 0X. Floating-point numbers always have a decimal point. Trailing zeros aren't removed from numbers printed with the g or G conversions.
0 *(zero)*	Numbers are padded with leading zeros up to the field width. The 0 flag is ignored if the conversion is d, i, o, u, x, or X and a precision is specified. (The - flag overrides the 0 flag.)

- ***Minimum field width*** (optional). An item that's too small to occupy this number of characters will be padded. (By default, spaces are added to the left of the item, thus right-justifying it within the field.) An item that's too large for the field width will still be displayed in its entirety. The field width is either an integer or the character *. If * is present, the field width is obtained from the next argument. If this argument is negative, it's treated as a positive number preceded by a - flag.
- ***Precision*** (optional). The meaning of the precision depends on the conversion:

  d, i, o, u, x, X: minimum number of digits
  (leading zeros are added if the number has fewer digits)
  a, A, e, E, f, F: number of digits after the decimal point
  g, G: number of significant digits
  s: maximum number of bytes

  The precision is a period (.) followed by an integer or the character *. If * is present, the precision is obtained from the next argument. (If this argument is negative, the effect is the same as not specifying a precision.) If only the period is present, the precision is zero.

- ***Length modifier*** (optional). The presence of a length modifier indicates that the item to be displayed has a type that's longer or shorter than is normal for a particular conversion specification. (For example, `%d` normally refers to an `int` value; `%hd` is used to display a `short int` and `%ld` is used to display a `long int`.) Table 22.5 lists each length modifier, the conversion specifiers with which it may be used, and the type indicated by the combination of the two. (Any combination of length modifier and conversion specifier not shown in the table causes undefined behavior.)

Table 22.5 Length Modifiers for ...`printf` Functions

Length Modifier	*Conversion Specifiers*	*Meaning*
hh†	d, i, o, u, x, X	signed char, unsigned char
	n	signed char *
h	d, i, o, u, x, X	short int, unsigned short int
	n	short int *
l (*ell*)	d, i, o, u, x, X	long int, unsigned long int
	n	long int *
	c	wint_t
	s	wchar_t *
	a, A, e, E, f, F, g, G	no effect
ll† (*ell-ell*)	d, i, o, u, x, X	long long int, unsigned long long int
	n	long long int *
j†	d, i, o, u, x, X	intmax_t, uintmax_t
	n	intmax_t *
z†	d, i, o, u, x, X	size_t
	n	size_t *
t†	d, i, o, u, x, X	ptrdiff_t
	n	ptrdiff_t *
L	a, A, e, E, f, F, g, G	long double

†C99 only

- ***Conversion specifier***. The conversion specifier must be one of the characters listed in Table 22.6. Notice that `f`, `F`, `e`, `E`, `g`, `G`, `a`, and `A` are all designed to write `double` values. However, they work fine with `float` values as well; thanks to the default argument promotions, `float` arguments are converted automatically to `double` when passed to a function with a variable number of arguments. Similarly, a character passed to ...`printf` is converted automatically to `int`, so the `c` conversion works properly.

default argument promotions ➤9.3

Be careful to follow the rules described here; the effect of using an invalid conversion specification is undefined.

Table 22.6
Conversion Specifiers for ...printf Functions

Conversion Specifier	*Meaning*
d, i	Converts an int value to decimal form.
o, u, x, X	Converts an unsigned int value to base 8 (o), base 10 (u), or base 16 (x, X). x displays the hexadecimal digits a–f in lower case; X displays them in upper case.
f, F†	Converts a double value to decimal form, putting the decimal point in the correct position. If no precision is specified, displays six digits after the decimal point.
e, E	Converts a double value to scientific notation. If no precision is specified, displays six digits after the decimal point. If e is chosen, the exponent is preceded by the letter e; if E is chosen, the exponent is preceded by E.
g, G	g converts a double value to either f form or e form. e form is selected if the number's exponent is less than –4 *or* greater than or equal to the precision. Trailing zeros are not displayed (unless the # flag is used); a decimal point appears only when followed by a digit. G chooses between F and E forms.
a†, A†	Converts a double value to hexadecimal scientific notation using the form [-]0x*h*.*hhhh*p±*d*, where [-] is an optional minus sign, the *h*'s represent hex digits, ± is either a plus or minus sign, and *d* is the exponent. *d* is a decimal number that represents a power of 2. If no precision is specified, enough digits are displayed after the decimal point to represent the exact value of the number (if possible). a displays the hex digits a–f in lower case; A displays them in upper case. The choice of a or A also affects the case of the letters x and p.
c	Displays an int value as an unsigned character.
s	Writes the characters pointed to by the argument. Stops writing when the number of bytes specified by the precision (if present) is reached or a null character is encountered.
p	Converts a void * value to printable form.
n	The corresponding argument must point to an object of type int. Stores in this object the number of characters written so far by this call of ...printf; produces no output.
%	Writes the character %.

†C99 only

C99 C99 Changes to ...printf Conversion Specifications

The conversion specifications for printf and fprintf have undergone a number of changes in C99:

- ***Additional length modifiers.*** C99 adds the hh, ll, j, z, and t length modifiers. hh and ll provide additional length options, j allows greatest-width integers to be written, and z and t make it easier to write values of type size_t and ptrdiff_t, respectively.

greatest-width integers ➤27.1

- ***Additional conversion specifiers.*** C99 adds the `F`, `a`, and `A` conversion specifiers. `F` is the same as `f` except for the way in which infinity and NaN (see below) are written. The `a` and `A` conversion specifications are rarely used. They're related to hexadecimal floating constants, which are discussed in the Q&A section at the end of Chapter 7.

IEEE floating-point standard ➤23.4

- ***Ability to write infinity and NaN.*** The IEEE 754 floating-point standard allows the result of a floating-point operation to be infinity, negative infinity, or NaN ("not a number"). For example, dividing 1.0 by 0.0 yields positive infinity, dividing –1.0 by 0.0 yields negative infinity, and dividing 0.0 by 0.0 yields NaN (because the result is mathematically undefined). In C99, the `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversion specifiers are capable of converting these special values to a form that can be displayed. `a`, `e`, `f`, and `g` convert positive infinity to `inf` or `infinity` (either one is legal), negative infinity to `-inf` or `-infinity`, and NaN to `nan` or `-nan` (possibly followed by a series of characters enclosed in parentheses). `A`, `E`, `F`, and `G` are equivalent to `a`, `e`, `f`, and `g`, except that upper-case letters are used (`INF`, `INFINITY`, `NAN`).

wide characters ➤25.2

- ***Support for wide characters.*** Another C99 feature is the ability of `fprintf` to write wide characters. The `%lc` conversion specification is used to write a single wide character; `%ls` is used for a string of wide characters.
- ***Previously undefined conversion specifications now allowed.*** In C89, the effect of using `%le`, `%lE`, `%lf`, `%lg`, and `%lG` is undefined. These conversion specifications are legal in C99 (the `l` length modifier is simply ignored).

Examples of ...`printf` Conversion Specifications

Whew! It's about time for a few examples. We've seen plenty of everyday conversion specifications in previous chapters, so we'll concentrate here on illustrating some of the more advanced ones. As in previous chapters, I'll use • to represent the space character.

Let's start off by examining the effect of flags on the `%d` conversion (they have a similar effect on other conversions). The first line of Table 22.7 shows the effect of `%8d` without any flags. The next four lines show the effect of the `-`, `+`, *space*, and `0` flags (the # flag is never used with `%d`). The remaining lines show the effect of combinations of flags.

Table 22.7
Effect of Flags on the `%d` Conversion

Conversion Specification	*Result of Applying Conversion to 123*	*Result of Applying Conversion to –123*
`%8d`	•••••123	••••-123
`%-8d`	123•••••	-123••••
`%+8d`	••••+123	••••-123
`% 8d`	•••••123	••••-123
`%08d`	00000123	-0000123
`%-+8d`	+123••••	-123••••
`%- 8d`	•123••••	-123••••
`%+08d`	+0000123	-0000123
`% 08d`	•0000123	-0000123

Table 22.8 shows the effect of the # flag on the o, x, X, g, and G conversions.

Table 22.8
Effect of the # Flag

Conversion Specification	*Result of Applying Conversion to 123*	*Result of Applying Conversion to 123.0*
%8o	•••••173	
%#8o	••••0173	
%8x	••••••7b	
%#8x	••••0x7b	
%8X	••••••7B	
%#8X	••••0X7B	
%8g		•••••123
%#8g		•123.000
%8G		•••••123
%#8G		•123.000

In previous chapters, we've used the minimum field width and precision when displaying numbers, so there's no point in more examples here. Instead, Table 22.9 shows the effect of the minimum field width and precision on the %s conversion.

Table 22.9
Effect of Minimum Field Width and Precision on the %s Conversion

Conversion Specification	*Result of Applying Conversion to* "bogus"	*Result of Applying Conversion to* "buzzword"
%6s	•bogus	buzzword
%-6s	bogus•	buzzword
%.4s	bogu	buzz
%6.4s	••bogu	••buzz
%-6.4s	bogu••	buzz••

Table 22.10 illustrates how the %g conversion displays some numbers in %e form and others in %f form. All numbers in the table were written using the %.4g conversion specification. The first two numbers have exponents of at least 4, so they're displayed in %e form. The next eight numbers are displayed in %f form. The last two numbers have exponents less than –4, so they're displayed in %e form.

Table 22.10
Examples of the %g Conversion

Number	*Result of Applying* %.4g *Conversion to Number*
123456.	1.235e+05
12345.6	1.235e+04
1234.56	1235
123.456	123.5
12.3456	12.35
1.23456	1.235
.123456	0.1235
.0123456	0.01235
.00123456	0.001235
.000123456	0.0001235
.0000123456	1.235e-05
.00000123456	1.235e-06

In the past, we've assumed that the minimum field width and precision were constants embedded in the format string. Putting the * character where either number would normally go allows us to specify it as an argument *after* the format string. For example, the following calls of `printf` all produce the same output:

```
printf("%6.4d", i);
printf("%*.4d", 6, i);
printf("%6.*d", 4, i);
printf("%*.*d", 6, 4, i);
```

Notice that the values to be filled in for the * come just before the value to be displayed. A major advantage of *, by the way, is that it allows us to use a macro to specify the width or precision:

```
printf("%*d", WIDTH, i);
```

We can even compute the width or precision during program execution:

```
printf("%*d", page_width / num_cols, i);
```

The most unusual specifications are `%p` and `%n`. The `%p` conversion allows us to print the value of a pointer:

```
printf("%p", (void *) ptr);   /* displays value of ptr */
```

Although `%p` is occasionally useful during debugging, it's not a feature that most programmers use on a daily basis. The C standard doesn't specify what a pointer looks like when printed using `%p`, but it's likely to be shown as an octal or hexadecimal number.

The `%n` conversion is used to find out how many characters have been printed so far by a call of ...`printf`. For example, after the call

```
printf("%d%n\n", 123, &len);
```

the value of `len` will be 3, since `printf` had written 3 characters (`123`) by the time it reached `%n`. Notice that `&` must precede `len` (because `%n` requires a pointer) and that `len` itself isn't printed.

The ...`scanf` Functions

```
int fscanf(FILE * restrict stream,
           const char * restrict format, ...);
int scanf(const char * restrict format, ...);
```

fscanf
scanf

`fscanf` and `scanf` read data items from an input stream, using a format string to indicate the layout of the input. After the format string, any number of pointers—each pointing to an object—follow as additional arguments. Input items are converted (according to conversion specifications in the format string) and stored in these objects.

`scanf` always reads from `stdin` (the standard input stream), whereas `fscanf` reads from the stream indicated by its first argument:

```
scanf("%d%d", &i, &j);          /* reads from stdin */
fscanf(fp, "%d%d", &i, &j);     /* reads from fp */
```

A call of `scanf` is equivalent to a call of `fscanf` with `stdin` as the first argument.

The ...`scanf` functions return prematurely if an ***input failure*** occurs (no more input characters could be read) or if a ***matching failure*** occurs (the input characters didn't match the format string). (In C99, an input failure can also occur because of an ***encoding error,*** which means that an attempt was made to read a multibyte character, but the input characters didn't correspond to any valid multibyte character.) Both functions return the number of data items that were read and assigned to objects; they return `EOF` if an input failure occurs before any data items can be read.

C99

multibyte characters ➤25.2

Loops that test `scanf`'s return value are common in C programs. The following loop, for example, reads a series of integers one by one, stopping at the first sign of trouble:

idiom

```
while (scanf("%d", &i) == 1) {
  ...
}
```

...`scanf` Format Strings

Calls of the ...`scanf` functions resemble those of the ...`printf` functions. That similarity can be misleading, however; the ...`scanf` functions work quite differently from the ...`printf` functions. It pays to think of `scanf` and `fscanf` as "pattern-matching" functions. The format string represents a pattern that a ...`scanf` function attempts to match as it reads input. If the input doesn't match the format string, the function returns as soon as it detects the mismatch; the input character that didn't match is "pushed back" to be read in the future.

A ...`scanf` format string may contain three things:

- ***Conversion specifications.*** Conversion specifications in a ...`scanf` format string resemble those in a ...`printf` format string. Most conversion specifications skip white-space characters at the beginning of an input item (the exceptions are `%[`, `%c`, and `%n`). Conversion specifications never skip *trailing* white-space characters, however. If the input contains •`123`¤, the `%d` conversion specification consumes •, `1`, `2`, and `3`, but leaves ¤ unread. (I'm using • to represent the space character and ¤ to represent the new-line character.)

white-space characters ➤3.2

- ***White-space characters.*** One or more consecutive white-space characters in a ...`scanf` format string match zero or more white-space characters in the input stream.
- ***Non-white-space characters.*** A non-white-space character other than `%` matches the same character in the input stream.

For example, the format string `"ISBN %d-%d-%ld-%d"` specifies that the input will consist of:

the letters `ISBN`
possibly some white-space characters
an integer
the - character
an integer (possibly preceded by white-space characters)
the - character
a long integer (possibly preceded by white-space characters)
the - character
an integer (possibly preceded by white-space characters)

...scanf Conversion Specifications

Conversion specifications for ...`scanf` functions are actually a little simpler than those for ...`printf` functions. A ...`scanf` conversion specification consists of the character `%` followed by the items listed below (in the order shown).

- `*` (optional). The presence of `*` signifies ***assignment suppression:*** an input item is read but not assigned to an object. Items matched using `*` aren't included in the count that ...`scanf` returns.
- ***Maximum field width*** (optional). The maximum field width limits the number of characters in an input item; conversion of the item ends if this number is reached. White-space characters skipped at the beginning of a conversion don't count.
- ***Length modifier*** (optional). The presence of a length modifier indicates that the object in which the input item will be stored has a type that's longer or shorter than is normal for a particular conversion specification. Table 22.11 lists each length modifier, the conversion specifiers with which it may be used, and the type indicated by the combination of the two. (Any combination of length modifier and conversion specifier not shown in the table causes undefined behavior.)

Table 22.11
Length Modifiers for ...`scanf` Functions

Length Modifier	*Conversion Specifiers*	*Meaning*
hh†	d, i, o, u, x, X, n	signed char *, unsigned char *
h	d, i, o, u, x, X, n	short int *, unsigned short int *
l (*ell*)	d, i, o, u, x, X, n	long int *, unsigned long int *
	a, A, e, E, f, F, g, G	double *
	c, s, or [	wchar_t *
ll† (*ell-ell*)	d, i, o, u, x, X, n	long long int *, unsigned long long int *
j†	d, i, o, u, x, X, n	intmax_t *, uintmax_t *
z†	d, i, o, u, x, X, n	size_t *
t†	d, i, o, u, x, X, n	ptrdiff_t *
L	a, A, e, E, f, F, g, G	long double *

†C99 only

- ***Conversion specifier.*** The conversion specifier must be one of the characters listed in Table 22.12.

Table 22.12 Conversion Specifiers for ...`scanf` Functions

Conversion Specifier	*Meaning*
`d`	Matches a decimal integer; the corresponding argument is assumed to have type `int *`.
`i`	Matches an integer; the corresponding argument is assumed to have type `int *`. The integer is assumed to be in base 10 unless it begins with `0` (indicating octal) or with `0x` or `0X` (hexadecimal).
`o`	Matches an octal integer; the corresponding argument is assumed to have type `unsigned int *`.
`u`	Matches a decimal integer; the corresponding argument is assumed to have type `unsigned int *`.
`x`, `X`	Matches a hexadecimal integer; the corresponding argument is assumed to have type `unsigned int *`.
`a`†, `A`†, `e`, `E`, `f`, `F`†, `g`, `G`	Matches a floating-point number; the corresponding argument is assumed to have type `float *`. In C99, the number can be infinity or NaN.
`c`	Matches *n* characters, where *n* is the maximum field width, or one character if no field width is specified. The corresponding argument is assumed to be a pointer to a character array (or a character object, if no field width is specified). Doesn't add a null character at the end.
`s`	Matches a sequence of non-white-space characters, then adds a null character at the end. The corresponding argument is assumed to be a pointer to a character array.
`[`	Matches a nonempty sequence of characters from a scanset, then adds a null character at the end. The corresponding argument is assumed to be a pointer to a character array.
`p`	Matches a pointer value in the form that ...`printf` would have written it. The corresponding argument is assumed to be a pointer to a `void *` object.
`n`	The corresponding argument must point to an object of type `int`. Stores in this object the number of characters read so far by this call of ...`scanf`. No input is consumed and the return value of ...`scanf` isn't affected.
`%`	Matches the character `%`.

†C99 only

Numeric data items can always begin with a sign (+ or -). The `o`, `u`, `x`, and `X` specifiers convert the item to unsigned form, however, so they're not normally used to read negative numbers.

The `[` specifier is a more complicated (and more flexible) version of the `s` specifier. A complete conversion specification using `[` has the form `%[`*set*`]` or `%[^`*set*`]`, where *set* can be any set of characters. (If `]` is one of the characters in *set*, however, it must come first.) `%[`*set*`]` matches any sequence of characters in *set* (the ***scanset***). `%[^`*set*`]` matches any sequence of characters *not* in *set* (in other words, the scanset consists of all characters not in *set*). For example, `%[abc]`

matches any string containing only the letters a, b, and c, while %[^abc] matches any string that doesn't contain a, b, or c.

numeric conversion functions ➤26.2

Many of the ...scanf conversion specifiers are closely related to the numeric conversion functions in <stdlib.h>. These functions convert strings (like "-297") to their equivalent numeric values (–297). The d specifier, for example, looks for an optional + or - sign, followed by a series of decimal digits; this is exactly the same form that the strtol function requires when asked to convert a string to a decimal number. Table 22.13 shows the correspondence between conversion specifiers and numeric conversion functions.

Table 22.13 Correspondence between ...scanf Conversion Specifiers and Numeric Conversion Functions

Conversion Specifier	*Numeric Conversion Function*
d	strtol with 10 as the base
i	strtol with 0 as the base
o	strtoul with 8 as the base
u	strtoul with 10 as the base
x, X	strtoul with 16 as the base
a, A, e, E, f, F, g, G	strtod

It pays to be careful when writing calls of scanf. An invalid conversion specification in a scanf format string is just as bad as one in a printf format string; either one causes undefined behavior.

C99 Changes to ...scanf Conversion Specifications

The conversion specifications for scanf and fscanf have undergone some changes in C99, but the list isn't as extensive as it was for the ...printf functions:

- ***Additional length modifiers.*** C99 adds the hh, ll, j, z, and t length modifiers. These correspond to the length modifiers in ...printf conversion specifications.
- ***Additional conversion specifiers.*** C99 adds the F, a, and A conversion specifiers. They're provided for symmetry with ...printf; the ...scanf functions treat them the same as e, E, f, g, and G.
- ***Ability to read infinity and NaN.*** Just as the ...printf functions can write infinity and NaN, the ...scanf functions can read these values. To be read properly, they should have the same appearance as values written by the ...printf functions, with case being ignored. (For example, either INF or inf will be read as infinity.)
- ***Support for wide characters.*** The ...scanf functions are able to read multibyte characters, which are then converted to wide characters for storage. The %lc conversion specification is used to read a single multibyte character or a

sequence of multibyte characters; `%ls` is used to read a string of multibyte characters (a null character is added at the end). The `%l[`*set*`]` and `%l[^`*set*`]` conversion specifications can also read a string of multibyte characters.

`scanf` Examples

The next three tables contain sample calls of `scanf`. Each call is applied to the input characters shown to its right. Characters printed in ~~strikeout~~ are consumed by the call. The values of the variables after the call appear to the right of the input.

The examples in Table 22.14 show the effect of combining conversion specifications, white-space characters, and non-white-space characters. In three cases no value is assigned to `j`, so it retains its value from before the call of `scanf`. The examples in Table 22.15 show the effect of assignment suppression and specifying a field width. The examples in Table 22.16 illustrate the more esoteric conversion specifiers (`i`, `[`, and `n`).

Table 22.14 `scanf` Examples (Group 1)

`scanf` *Call*	*Input*	*Variables*
`n = scanf("%d%d", &i, &j);`	~~12~~•,•34¤	n: 1 i: 12 j: unchanged
`n = scanf("%d,%d", &i, &j);`	~~12~~•,•34¤	n: 1 i: 12 j: unchanged
`n = scanf("%d ,%d", &i, &j);`	~~12•,•34~~¤	n: 2 i: 12 j: 34
`n = scanf("%d, %d", &i, &j);`	~~12~~•,•34¤	n: 1 i: 12 j: unchanged

Table 22.15 `scanf` Examples (Group 2)

`scanf` *Call*	*Input*	*Variables*
`n = scanf("%*d%d", &i);`	~~12•34~~¤	n: 1 i: 34
`n = scanf("%*s%s", str);`	~~My•Fair~~•Lady¤	n: 1 str: "Fair"
`n = scanf("%1d%2d%3d",` `&i, &j, &k);`	~~12345~~¤	n: 3 i: 1 j: 23 k: 45
`n = scanf("%2d%2s%2d",` `&i, str, &j);`	~~123456~~¤	n: 3 i: 12 str: "34" j: 56

Table 22.16
scanf Examples (Group 3)

scanf *Call*	*Input*	*Variables*
`n = scanf("%i%i%i", &i, &j, &k);`	~~12•012•0x12~~¤	n: 3 i: 12 j: 10 k: 18
`n = scanf("%[0123456789]", str);`	~~123~~abc¤	n: 1 str: "123"
`n = scanf("%[0123456789]", str);`	abc123¤	n: 0 str: unchanged
`n = scanf("%[^0123456789]", str);`	~~abc~~123¤	n: 1 str: "abc"
`n = scanf("%*d%d%n", &i, &j);`	~~10•20•~~30¤	n: 1 i: 20 j: 5

Detecting End-of-File and Error Conditions

```
void clearerr(FILE *stream);
int feof(FILE *stream);
int ferror(FILE *stream);
```

If we ask a ...`scanf` function to read and store *n* data items, we expect its return value to be *n*. If the return value is less than *n*, something went wrong. There are three possibilities:

- ***End-of-file.*** The function encountered end-of-file before matching the format string completely.
- ***Read error.*** The function was unable to read characters from the stream.
- ***Matching failure.*** A data item was in the wrong format. For example, the function might have encountered a letter while searching for the first digit of an integer.

But how can we tell which kind of failure occurred? In many cases, it doesn't matter; something went wrong, and we've got to abandon the program. There may be times, however, when we'll need to pinpoint the reason for the failure.

Every stream has two indicators associated with it: an ***error indicator*** and an ***end-of-file indicator.*** These indicators are cleared when the stream is opened. Not surprisingly, encountering end-of-file sets the end-of-file indicator, and a read error sets the error indicator. (The error indicator is also set when a write error occurs on an output stream.) A matching failure doesn't change either indicator.

clearerr

Once the error or end-of-file indicator is set, it remains in that state until it's explicitly cleared, perhaps by a call of the `clearerr` function. `clearerr` clears both the end-of-file and error indicators:

```
clearerr(fp);    /* clears eof and error indicators for fp */
```

clearerr isn't needed often, since some of the other library functions clear one or both indicators as a side effect.

Q&A

feof
ferror

We can call the feof and ferror functions to test a stream's indicators to determine why a prior operation on the stream failed. The call feof(fp) returns a nonzero value if the end-of-file indicator is set for the stream associated with fp. The call ferror(fp) returns a nonzero value if the error indicator is set. Both functions return zero otherwise.

Q&A

When scanf returns a smaller-than-expected value, we can use feof and ferror to determine the reason. If feof returns a nonzero value, we've reached the end of the input file. If ferror returns a nonzero value, a read error occurred during input. If neither returns a nonzero value, a matching failure must have occurred. Regardless of what the problem was, the return value of scanf tells us how many data items were read before the problem occurred.

To see how feof and ferror might be used, let's write a function that searches a file for a line that begins with an integer. Here's how we intend to call the function:

```
n = find_int("foo");
```

"foo" is the name of the file to be searched. The function returns the value of the integer that it finds, which is then assigned to n. If a problem arises—the file can't be opened, a read error occurs, or no line begins with an integer—find_int will return an error code (–1, –2, or –3, respectively). I'll assume that no line in the file begins with a negative integer.

```
int find_int(const char *filename)
{
  FILE *fp = fopen(filename, "r");
  int n;

  if (fp == NULL)
    return -1;                  /* can't open file */

  while (fscanf(fp, "%d", &n) != 1) {
    if (ferror(fp)) {
      fclose(fp);
      return -2;                /* read error */
    }
    if (feof(fp)) {
      fclose(fp);
      return -3;                /* integer not found */
    }
    fscanf(fp, "%*[^\n]");      /* skips rest of line */
  }

  fclose(fp);
  return n;
}
```

The while loop's controlling expression calls fscanf in an attempt to read an integer from the file. If the attempt fails (fscanf returns a value other than 1),

`find_int` calls `ferror` and `feof` to see if the problem was a read error or end-of-file. If not, `fscanf` must have failed because of a matching error, so `find_int` skips the rest of the characters on the current line and tries again. Note the use of the conversion `%*[^\n]` to skip all characters up to the next new-line. (Now that we know about scansets, it's time to show off!)

22.4 Character I/O

In this section, we'll examine library functions that read and write single characters. These functions work equally well with text streams and binary streams.

You'll notice that the functions in this section treat characters as values of type `int`, not `char`. One reason is that the input functions indicate an end-of-file (or error) condition by returning `EOF`, which is a negative integer constant.

Output Functions

```
int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);
```

putchar

`putchar` writes one character to the `stdout` stream:

```
putchar(ch);      /* writes ch to stdout */
```

fputc
putc

`fputc` and `putc` are more general versions of `putchar` that write a character to an arbitrary stream:

```
fputc(ch, fp);    /* writes ch to fp */
putc(ch, fp);     /* writes ch to fp */
```

Although `putc` and `fputc` do the same thing, `putc` is usually implemented as a macro (as well as a function), while `fputc` is implemented only as a function. `putchar` itself is usually a macro defined in the following way:

```
#define putchar(c) putc((c), stdout)
```

It may seem odd that the library provides both `putc` and `fputc`. But, as we saw in Section 14.3, macros have several potential problems. The C standard allows the `putc` macro to evaluate the `stream` argument more than once, which `fputc` isn't permitted to do. Although programmers usually prefer `putc`, which gives a faster program, `fputc` is available as an alternative.

Q&A

If a write error occurs, all three functions set the error indicator for the stream and return `EOF`; otherwise, they return the character that was written.

Input Functions

```
int fgetc(FILE *stream);
int getc(FILE *stream);
int getchar(void);
int ungetc(int c, FILE *stream);
```

getchar `getchar` reads a character from the `stdin` stream:

```
ch = getchar();    /* reads a character from stdin */
```

fgetc getc `fgetc` and `getc` read a character from an arbitrary stream:

```
ch = fgetc(fp);    /* reads a character from fp */
ch = getc(fp);     /* reads a character from fp */
```

All three functions treat the character as an `unsigned char` value (which is then converted to `int` type before it's returned). As a result, they never return a negative value other than `EOF`.

The relationship between `getc` and `fgetc` is similar to that between `putc` and `fputc`. `getc` is usually implemented as a macro (as well as a function), while `fgetc` is implemented only as a function. `getchar` is normally a macro as well:

```
#define getchar() getc(stdin)
```

For reading characters from a file, programmers usually prefer `getc` over `fgetc`. Since `getc` is normally available in macro form, it tends to be faster. `fgetc` can be used as a backup if `getc` isn't appropriate. (The standard allows the `getc` macro to evaluate its argument more than once, which may be a problem.)

The `fgetc`, `getc`, and `getchar` functions behave the same if a problem occurs. At end-of-file, they set the stream's end-of-file indicator and return `EOF`. If a read error occurs, they set the stream's error indicator and return `EOF`. To differentiate between the two situations, we can call either `feof` or `ferror`.

One of the most common uses of `fgetc`, `getc`, and `getchar` is to read characters from a file, one by one, until end-of-file occurs. It's customary to use the following `while` loop for that purpose:

idiom

```
while ((ch = getc(fp)) != EOF) {
  ...
}
```

After reading a character from the file associated with `fp` and storing it in the variable `ch` (which must be of type `int`), the `while` test compares `ch` with `EOF`. If `ch` isn't equal to `EOF`, we're not at the end of the file yet, so the body of the loop is executed. If `ch` is equal to `EOF`, the loop terminates.

Always store the return value of `fgetc`, `getc`, or `getchar` in an `int` variable, not a `char` variable. Testing a `char` variable against `EOF` may give the wrong result.

ungetc

There's one other character input function, `ungetc`, which "pushes back" a character read from a stream and clears the stream's end-of-file indicator. This capability can be handy if we need a "lookahead" character during input. For instance, to read a series of digits, stopping at the first nondigit, we could write

isdigit function ➤23.5

```
while (isdigit(ch = getc(fp))) {
  ...
}
ungetc(ch, fp);   /* pushes back last character read */
```

The number of characters that can be pushed back by consecutive calls of `ungetc`—with no intervening read operations—depends on the implementation and the type of stream involved; only the first call is guaranteed to succeed. Calling a file-positioning function (`fseek`, `fsetpos`, or `rewind`) causes the pushed-back characters to be lost.

file-positioning functions ➤22.7

`ungetc` returns the character it was asked to push back. However, it returns `EOF` if an attempt is made to push back `EOF` or to push back more characters than the implementation allows.

PROGRAM

Copying a File

The following program makes a copy of a file. The names of the original file and the new file will be specified on the command line when the program is executed. For example, to copy the file `f1.c` to `f2.c`, we'd use the command

```
fcopy f1.c f2.c
```

`fcopy` will issue an error message if there aren't exactly two file names on the command line or if either file can't be opened.

fcopy.c

```
/* Copies a file */

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

int main(int argc, char *argv[])
{
  FILE *source_fp, *dest_fp;
  int ch;
```

```
  if (argc != 3) {
    fprintf(stderr, "usage: fcopy source dest\n");
    exit(EXIT_FAILURE);
  }

  if ((source_fp = fopen(argv[1], "rb")) == NULL) {
    fprintf(stderr, "Can't open %s\n", argv[1]);
    exit(EXIT_FAILURE);
  }

  if ((dest_fp = fopen(argv[2], "wb")) == NULL) {
    fprintf(stderr, "Can't open %s\n", argv[2]);
    fclose(source_fp);
    exit(EXIT_FAILURE);
  }

  while ((ch = getc(source_fp)) != EOF)
    putc(ch, dest_fp);

  fclose(source_fp);
  fclose(dest_fp);
  return 0;
}
```

Using `"rb"` and `"wb"` as the file modes enables `fcopy` to copy both text and binary files. If we used `"r"` and `"w"` instead, the program wouldn't necessarily be able to copy binary files.

22.5 Line I/O

We'll now turn to library functions that read and write lines. These functions are used mostly with text streams, although it's legal to use them with binary streams as well.

Output Functions

```
int fputs(const char * restrict s,
          FILE * restrict stream);
int puts(const char *s);
```

puts

We encountered the `puts` function in Section 13.3; it writes a string of characters to `stdout`:

```
puts("Hi, there!");    /* writes to stdout */
```

After it writes the characters in the string, `puts` always adds a new-line character.

fputs

`fputs` is a more general version of `puts`. Its second argument indicates the stream to which the output should be written:

```
fputs("Hi, there!", fp);   /* writes to fp */
```

Unlike `puts`, the `fputs` function doesn't write a new-line character unless one is present in the string.

Both functions return `EOF` if a write error occurs; otherwise, they return a nonnegative number.

Input Functions

```
char *fgets(char * restrict s, int n,
            FILE * restrict stream);
char *gets(char *s);
```

gets

The `gets` function, which we first encountered in Section 13.3, reads a line of input from `stdin`:

```
gets(str);   /* reads a line from stdin */
```

`gets` reads characters one by one, storing them in the array pointed to by `str`, until it reads a new-line character (which it discards).

fgets

`fgets` is a more general version of `gets` that can read from any stream. `fgets` is also safer than `gets`, since it limits the number of characters that it will store. Here's how we might use `fgets`, assuming that `str` is the name of a character array:

```
fgets(str, sizeof(str), fp);   /* reads a line from fp */
```

This call will cause `fgets` to read characters until it reaches the first new-line character or `sizeof(str) - 1` characters have been read, whichever happens first. If it reads the new-line character, `fgets` stores it along with the other characters. (Thus, `gets` *never* stores the new-line character, but `fgets` *sometimes* does.)

Both `gets` and `fgets` return a null pointer if a read error occurs or they reach the end of the input stream before storing any characters. (As usual, we can call `feof` or `ferror` to determine which situation occurred.) Otherwise, both return their first argument, which points to the array in which the input was stored. As you'd expect, both functions store a null character at the end of the string.

Now that you know about `fgets`, I'd suggest using it instead of `gets` in most situations. With `gets`, there's always the possibility of stepping outside the bounds of the receiving array, so it's safe to use only when the string being read is *guaranteed* to fit into the array. When there's no guarantee (and there usually isn't), it's much safer to use `fgets`. Note that `fgets` will read from the standard input stream if passed `stdin` as its third argument:

```
fgets(str, sizeof(str), stdin);
```

22.6 Block I/O

```
size_t fread(void * restrict ptr,
             size_t size, size_t nmemb,
             FILE * restrict stream);
size_t fwrite(const void * restrict ptr,
              size_t size, size_t nmemb,
              FILE * restrict stream);
```

The `fread` and `fwrite` functions allow a program to read and write large blocks of data in a single step. `fread` and `fwrite` are used primarily with binary streams, although—with care—it's possible to use them with text streams as well.

Q&A

fwrite

`fwrite` is designed to copy an array from memory to a stream. The first argument in a call of `fwrite` is the array's address, the second argument is the size of each array element (in bytes), and the third argument is the number of elements to write. The fourth argument is a file pointer, indicating where the data should be written. To write the entire contents of the array `a`, for instance, we could use the following call of `fwrite`:

```
fwrite(a, sizeof(a[0]), sizeof(a) / sizeof(a[0]), fp);
```

There's no rule that we have to write the entire array; we could just as easily write any portion of it. `fwrite` returns the number of elements (*not* bytes) actually written. This number will be less than the third argument if a write error occurs.

fread

`fread` will read the elements of an array from a stream. `fread`'s arguments are similar to `fwrite`'s: the array's address, the size of each element (in bytes), the number of elements to read, and a file pointer. To read the contents of a file into the array `a`, we might use the following call of `fread`:

```
n = fread(a, sizeof(a[0]), sizeof(a) / sizeof(a[0]), fp);
```

It's important to check `fread`'s return value, which indicates the actual number of elements (*not* bytes) read. This number should equal the third argument unless the end of the input file was reached or a read error occurred. The `feof` and `ferror` functions can be used to determine the reason for any shortage.

Be careful not to confuse `fread`'s second and third arguments. Consider the following call of `fread`:

```
fread(a, 1, 100, fp)
```

We're asking `fread` to read 100 one-byte elements, so it will return a value

between 0 and 100. The following call asks `fread` to read one block of 100 bytes:

```
fread(a, 100, 1, fp)
```

`fread`'s return value in this case will be either 0 or 1.

`fwrite` is convenient for a program that needs to store data in a file before terminating. Later, the program (or another program, for that matter) can use `fread` to read the data back into memory. Despite appearances, the data doesn't need to be in array form; `fread` and `fwrite` work just as well with variables of all kinds. Structures, in particular, can be read by `fread` or written by `fwrite`. To write a structure variable `s` to a file, for instance, we could use the following call of `fwrite`:

```
fwrite(&s, sizeof(s), 1, fp);
```

Be careful when using `fwrite` to write out structures that contain pointer values; these values aren't guaranteed to be valid when read back in.

22.7 File Positioning

```
int fgetpos(FILE * restrict stream,
            fpos_t * restrict pos);
int fseek(FILE *stream, long int offset, int whence);
int fsetpos(FILE *stream, const fpos_t *pos);
long int ftell(FILE *stream);
void rewind(FILE *stream);
```

Every stream has an associated ***file position.*** When a file is opened, the file position is set at the beginning of the file. (If the file is opened in "append" mode, however, the initial file position may be at the beginning or end of the file, depending on the implementation.) Then, when a read or write operation is performed, the file position advances automatically, allowing us to move through the file in a sequential manner.

Although sequential access is fine for many applications, some programs need the ability to jump around within a file, accessing some data here and other data there. If a file contains a series of records, for example, we might want to jump directly to a particular record and read it or update it. `<stdio.h>` supports this form of access by providing five functions that allow a program to determine the current file position or to change it.

fseek

The `fseek` function changes the file position associated with the first argument (a file pointer). The third argument specifies whether the new position is to

be calculated with respect to the beginning of the file, the current position, or the end of the file. `<stdio.h>` defines three macros for this purpose:

`SEEK_SET`	Beginning of file
`SEEK_CUR`	Current file position
`SEEK_END`	End of file

The second argument is a (possibly negative) byte count. To move to the beginning of a file, for example, the seek direction would be `SEEK_SET` and the byte count would be zero:

```
fseek(fp, 0L, SEEK_SET);      /* moves to beginning of file */
```

To move to the end of a file, the seek direction would be `SEEK_END`:

```
fseek(fp, 0L, SEEK_END);      /* moves to end of file */
```

To move back 10 bytes, the seek direction would be `SEEK_CUR` and the byte count would be –10:

```
fseek(fp, -10L, SEEK_CUR);    /* moves back 10 bytes */
```

Note that the byte count has type `long int`, so I've used `0L` and `-10L` as arguments. (`0` and `-10` would also work, of course, since arguments are converted to the proper type automatically.)

Normally, `fseek` returns zero. If an error occurs (the requested position doesn't exist, for example), `fseek` returns a nonzero value.

The file-positioning functions are best used with binary streams, by the way. C doesn't prohibit programs from using them with text streams, but care is required because of operating system differences. `fseek` in particular is sensitive to whether a stream is text or binary. For text streams, either (1) `offset` (`fseek`'s second argument) must be zero or (2) `whence` (its third argument) must be `SEEK_SET` and `offset` a value obtained by a previous call of `ftell`. (In other words, we can only use `fseek` to move to the beginning or end of a text stream or to return to a place that was visited previously.) For binary streams, `fseek` isn't required to support calls in which `whence` is `SEEK_END`.

ftell

errno variable ➤24.2

The `ftell` function returns the current file position as a long integer. (If an error occurs, `ftell` returns `-1L` and stores an error code in `errno`.) The value returned by `ftell` may be saved and later supplied to a call of `fseek`, making it possible to return to a previous file position:

```
long file_pos;
...
file_pos = ftell(fp);              /* saves current position */
...
fseek(fp, file_pos, SEEK_SET);     /* returns to old position */
```

If `fp` is a binary stream, the call `ftell(fp)` returns the current file position as a byte count, where zero represents the beginning of the file. If `fp` is a text stream, however, `ftell(fp)` isn't necessarily a byte count. As a result, it's best not to perform arithmetic on values returned by `ftell`. For example, it's not a good

idea to subtract values returned by `ftell` to see how far apart two file positions are.

rewind

The `rewind` function sets the file position at the beginning. The call `rewind(fp)` is nearly equivalent to `fseek(fp, 0L, SEEK_SET)`. The difference? `rewind` doesn't return a value but does clear the error indicator for `fp`.

fgetpos
fsetpos
Q&A

`fseek` and `ftell` have one problem: they're limited to files whose positions can be stored in a long integer. For working with very large files, C provides two additional functions: `fgetpos` and `fsetpos`. These functions can handle large files because they use values of type `fpos_t` to represent file positions. An `fpos_t` value isn't necessarily an integer; it could be a structure, for instance.

The call `fgetpos(fp, &file_pos)` stores the file position associated with `fp` in the `file_pos` variable. The call `fsetpos(fp, &file_pos)` sets the file position for `fp` to be the value stored in `file_pos`. (This value must have been obtained by a previous call of `fgetpos`.) If a call of `fgetpos` or `fsetpos` fails, it stores an error code in `errno`. Both functions return zero when they succeed and a nonzero value when they fail.

Here's how we might use `fgetpos` and `fsetpos` to save a file position and return to it later:

```
fpos_t file_pos;
...
fgetpos(fp, &file_pos);    /* saves current position */
...
fsetpos(fp, &file_pos);    /* returns to old position */
```

PROGRAM

Modifying a File of Part Records

The following program opens a binary file containing `part` structures, reads the structures into an array, sets the `on_hand` member of each structure to 0, and then writes the structures back to the file. Note that the program opens the file in `"rb+"` mode, allowing both reading and writing.

invclear.c

```
/* Modifies a file of part records by setting the quantity
   on hand to zero for all records */

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

#define NAME_LEN 25
#define MAX_PARTS 100

struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} inventory[MAX_PARTS];
```

```
int num_parts;

int main(void)
{
  FILE *fp;
  int i;

  if ((fp = fopen("inventory.dat", "rb+")) == NULL) {
    fprintf(stderr, "Can't open inventory file\n");
    exit(EXIT_FAILURE);
  }

  num_parts = fread(inventory, sizeof(struct part),
                    MAX_PARTS, fp);

  for (i = 0; i < num_parts; i++)
    inventory[i].on_hand = 0;

  rewind(fp);
  fwrite(inventory, sizeof(struct part), num_parts, fp);
  fclose(fp);

  return 0;
}
```

Calling `rewind` is critical, by the way. After the `fread` call, the file position is at the end of the file. If we were to call `fwrite` without calling `rewind` first, `fwrite` would add new data to the end of the file instead of overwriting the old data.

22.8 String I/O

The functions described in this section are a bit unusual, since they have nothing to do with streams or files. Instead, they allow us to read and write data using a string as though it were a stream. The `sprintf` and `snprintf` functions write characters into a string in the same way they would be written to a stream; the `sscanf` function reads characters from a string as though it were reading from a stream. These functions, which closely resemble `printf` and `scanf`, are quite useful. `sprintf` and `snprintf` give us access to `printf`'s formatting capabilities without actually having to write data to a stream. Similarly, `sscanf` gives us access to `scanf`'s powerful pattern-matching capabilities. The remainder of this section covers `sprintf`, `snprintf`, and `sscanf` in detail.

Three similar functions (`vsprintf`, `vsnprintf`, and `vsscanf`) also belong to `<stdio.h>`. However, these functions rely on the `va_list` type, which is declared in `<stdarg.h>`. I'll postpone discussing them until Section 26.1, which covers that header.

Output Functions

```
int sprintf(char * restrict s,
            const char * restrict format, ...);
int snprintf(char * restrict s, size_t n,
             const char * restrict format, ...);
```

Note: In this and subsequent chapters, the prototype for a function that is new in C99 will be in italics. Also, the name of the function will be italicized when it appears in the left margin.

sprintf

The `sprintf` function is similar to `printf` and `fprintf`, except that it writes output into a character array (pointed to by its first argument) instead of a stream. `sprintf`'s second argument is a format string identical to that used by `printf` and `fprintf`. For example, the call

```
sprintf(date, "%d/%d/%d", 9, 20, 2010);
```

will write `"9/20/2010"` into `date`. When it's finished writing into a string, `sprintf` adds a null character and returns the number of characters stored (not counting the null character). If an encoding error occurs (a wide character could not be translated into a valid multibyte character), `sprintf` returns a negative value.

`sprintf` has a variety of uses. For example, we might occasionally want to format data for output without actually writing it. We can use `sprintf` to do the formatting, then save the result in a string until it's time to produce output. `sprintf` is also convenient for converting numbers to character form.

snprintf

The `snprintf` function is the same as `sprintf`, except for the additional parameter `n`. No more than `n` – 1 characters will be written to the string, not counting the terminating null character, which is always written unless `n` is zero. (Equivalently, we could say that `snprintf` writes at most `n` characters to the string, the last of which is a null character.) For example, the call

```
snprintf(name, 13, "%s, %s", "Einstein", "Albert");
```

will write `"Einstein, Al"` into `name`.

`snprintf` returns the number of characters that would have been written (not including the null character) had there been no length restriction. If an encoding error occurs, `snprintf` returns a negative number. To see if `snprintf` had room to write all the requested characters, we can test whether its return value was nonnegative and less than `n`.

Input Functions

```
int sscanf(const char * restrict s,
           const char * restrict format, ...);
```

sscanf The `sscanf` function is similar to `scanf` and `fscanf`, except that it reads from a string (pointed to by its first argument) instead of reading from a stream. `sscanf`'s second argument is a format string identical to that used by `scanf` and `fscanf`.

`sscanf` is handy for extracting data from a string that was read by another input function. For example, we might use `fgets` to obtain a line of input, then pass the line to `sscanf` for further processing:

```
fgets(str, sizeof(str), stdin);  /* reads a line of input */
sscanf(str, "%d%d", &i, &j);     /* extracts two integers */
```

One advantage of using `sscanf` instead of `scanf` or `fscanf` is that we can examine an input line as many times as needed, not just once, making it easier to recognize alternate input forms and to recover from errors. Consider the problem of reading a date that's written either in the form *month/day/year* or *month-day-year*. Assuming that `str` contains a line of input, we can extract the month, day, and year as follows:

```
if (sscanf(str, "%d /%d /%d", &month, &day, &year) == 3)
  printf("Month: %d, day: %d, year: %d\n", month, day, year);
else if (sscanf(str, "%d -%d -%d", &month, &day, &year) == 3)
  printf("Month: %d, day: %d, year: %d\n", month, day, year);
else
  printf("Date not in the proper form\n");
```

Like the `scanf` and `fscanf` functions, `sscanf` returns the number of data items successfully read and stored. `sscanf` returns `EOF` if it reaches the end of the string (marked by a null character) before finding the first item.

Q & A

Q: If I use input or output redirection, will the redirected file names show up as command-line arguments? [p. 541]

A: No; the operating system removes them from the command line. Let's say that we run a program by entering

```
demo foo <in_file bar >out_file baz
```

The value of `argc` will be 4, `argv[0]` will point to the program name, `argv[1]` will point to `"foo"`, `argv[2]` will point to `"bar"`, and `argv[3]` will point to `"baz"`.

Q: I thought that the end of a line was always marked by a new-line character. Now you're saying that the end-of-line marker varies, depending on the operating system. How you explain this discrepancy? [p. 542]

A: C library functions make it *appear* as though each line ends with a single new-line

character. Regardless of whether an input file contains a carriage-return character, a line-feed character, or both, a library function such as `getc` will return a single new-line character. The output functions perform the reverse translation. If a program calls a library function to write a new-line character to a file, the function will translate the character into the appropriate end-of-line marker. C's approach makes programs more portable and easier to write; we can work with text files without having to worry about how end-of-line is actually represented. Note that input/output performed on a file opened in binary mode isn't subject to any character translation—carriage return and line feed are treated the same as the other characters.

Q: I'm writing a program that needs to save data in a file, to be read later by another program. Is it better to store the data in text form or binary form? [p. 542]

A: That depends. If the data is all text to start with, there's not much difference. If the data contains numbers, however, the decision is tougher.

Binary form is usually preferable, since it can be read and written quickly. Numbers are already in binary form when stored in memory, so copying them to a file is easy. Writing numbers in text form is much slower, since each number must be converted (usually by `fprintf`) to character form. Reading the file later will also take more time, since numbers will have to be converted from text form back to binary. Moreover, storing data in binary form often saves space, as we saw in Section 22.1.

Binary files have two disadvantages, however. They're hard for humans to read, which can hamper debugging. Also, binary files generally aren't portable from one system to another, since different kinds of computers store data in different ways. For instance, some machines store `int` values using two bytes but others use four bytes. There's also the issue of byte order (big-endian versus little-endian).

Q: C programs for UNIX never seem to use the letter `b` in the mode string, even when the file being opened is binary. What gives? [p. 544]

A: In UNIX, text files and binary files have exactly the same format, so there's never any need to use `b`. UNIX programmers should still include the `b`, however, so that their programs will be more portable to other operating systems.

Q: I've seen programs that call `fopen` and put the letter `t` in the mode string. What does `t` mean?

A: The C standard allows additional characters to appear in the mode string, provided that they follow `r`, `w`, `a`, `b`, or `+`. Some compilers allow the use of `t` to indicate that a file is to be opened in text mode instead of binary mode. Of course, text mode is the default anyway, so `t` adds nothing. Whenever possible, it's best to avoid using `t` and other nonportable features.

Q: Why bother to call `fclose` to close a file? Isn't it true that all open files are closed automatically when a program terminates? [p. 545]

A: That's usually true, but not if the program calls `abort` to terminate. Even when `abort` isn't used, though, there are still good reasons to call `fclose`. First, it reduces the number of open files. Operating systems limit the number of files that a program may have open at the same time; large programs may bump into this limit. (The macro `FOPEN_MAX`, defined in `<stdio.h>`, specifies the minimum number of files that the implementation guarantees can be open simultaneously.) Second, the program becomes easier to understand and modify; by looking for the call of `fclose`, it's easier for the reader to determine the point at which a file is no longer in use. Third, there's the issue of safety. Closing a file ensures that its contents and directory entry are updated properly; if the program should crash later, at least the file will be intact.

`abort` function ➤26.2

Q: I'm writing a program that will prompt the user to enter a file name. How long should I make the character array that will store the file name? [p. 546]

A: That depends on your operating system. Fortunately, you can use the macro `FILENAME_MAX` (defined in `<stdio.h>`) to specify the size of the array. `FILENAME_MAX` is the length of a string that will hold the longest file name that the implementation guarantees can be opened.

Q: Can `fflush` flush a stream that was opened for both reading and writing? [p. 549]

A: According to the C standard, the effect of calling `fflush` is defined for a stream that (a) was opened for output, or (b) was opened for updating and whose last operation was not a read. In all other cases, the effect of calling `fflush` is undefined. When `fflush` is passed a null pointer, it flushes all streams that satisfy either (a) or (b).

Q: Can the format string in a call of `...printf` or `...scanf` be a variable?

A: Sure; it can be any expression of type `char *`. This property makes the `...printf` and `...scanf` functions even more versatile than we've had reason to suspect. Consider the following classic example from Kernighan and Ritchie's *The C Programming Language*, which prints a program's command-line arguments, separated by spaces:

```
while (--argc > 0)
  printf((argc > 1) ? "%s " : "%s", *++argv);
```

The format string is the expression `(argc > 1) ? "%s " : "%s"`, which evaluates to `"%s "` for all command-line arguments but the last.

Q: Which library functions other than `clearerr` clear a stream's error and end-of-file indicators? [p. 565]

A: Calling `rewind` clears both indicators, as does opening or reopening the stream. Calling `ungetc`, `fseek`, or `fsetpos` clears just the end-of-file indicator.

Q: I can't get `feof` to work; it seems to return zero even at end-of-file. What am I doing wrong? [p. 565]

A: `feof` will only return a nonzero value when a previous read operation has failed; you can't use `feof` to check for end-of-file *before* attempting to read. Instead, you should first attempt to read, then check the return value from the input function. If the return value indicates that the operation was unsuccessful, you can then use `feof` to determine whether the failure was due to end-of-file. In other words, it's best not to think of calling `feof` as a way to *detect* end-of-file. Instead, think of it as a way to *confirm* that end-of-file was the reason for the failure of a read operation.

Q: I still don't understand why the I/O library provides macros named `putc` and `getc` in addition to functions named `fputc` and `fgetc`. According to Section 21.1, there are already two versions of `putc` and `getc` (a macro and a function). If we need a genuine function instead of a macro, we can expose the `putc` or `getc` function by undefining the macro. So why do `fputc` and `fgetc` exist? [p. 566]

A: Historical reasons. Prior to standardization, C had no rule that there be a true function to back up each parameterized macro in the library. `putc` and `getc` were traditionally implemented only as macros; `fputc` and `fgetc` were implemented only as functions.

***Q: What's wrong with storing the return value of `fgetc`, `getc`, or `getchar` in a `char` variable? I don't see how testing a `char` variable against `EOF` could give the wrong answer. [p. 568]**

A: There are two cases in which this test can give the wrong result. To make the following discussion concrete, I'll assume two's-complement arithmetic.

First, suppose that `char` is an unsigned type. (Recall that some compilers treat `char` as a signed type but others treat it as an unsigned type.) Now suppose that `getc` returns `EOF`, which we store in a `char` variable named `ch`. If `EOF` represents –1 (its typical value), `ch` will end up with the value 255. Comparing `ch` (an unsigned character) with `EOF` (a signed integer) requires converting `ch` to a signed integer (255, in this case). The comparison against `EOF` fails, since 255 is not equal to –1.

Now assume that `char` is a signed type instead. Consider what happens if `getc` reads a byte containing the value 255 from a binary stream. Storing 255 in the `ch` variable gives it the value –1, since `ch` is a signed character. Testing whether `ch` is equal to `EOF` will (erroneously) give a true result.

Q: The character input functions described in Section 22.4 require that the Enter key be pressed before they can read what the user has typed. How can I write a program that responds to individual keystrokes?

A: As you've noticed, the `getc`, `fgetc`, and `getchar` functions are buffered; they don't start to read input until the user has pressed the Enter key. In order to read characters as they're entered—which is important for some kinds of programs—you'll need to use a nonstandard library that's tailored to your operating system. In UNIX, for example, the `curses` library often provides this capability.

Q: When I'm reading user input, how can I skip all characters left on the current input line?

A: One possibility is to write a small function that reads and ignores all characters up to (and including) the first new-line character:

```
void skip_line(void)
{
  while (getchar() != '\n')
    ;
}
```

Another possibility is to ask `scanf` to skip all characters up to the first new-line character:

```
scanf("%*[^\n]");    /* skips characters up to new-line */
```

`scanf` will read all characters up to the first new-line character, but not store them anywhere (the `*` indicates assignment suppression). The only problem with using `scanf` is that it leaves the new-line character unread, so you may have to discard it separately.

Whatever you do, don't call the `fflush` function:

```
fflush(stdin);    /* effect is undefined */
```

Although some implementations allow the use of `fflush` to "flush" unread input, it's not a good idea to assume that all do. `fflush` is designed to flush *output* streams; the C standard states that its effect on input streams is undefined.

Q: Why is it not a good idea to use `fread` and `fwrite` with text streams? [p. 571]

A: One difficulty is that, under some operating systems, the new-line character becomes a pair of characters when written to a text file (see Section 22.1 for details). We must take this expansion into account, or else we're likely to lose track of our data. For example, if we use `fwrite` to write blocks of 80 characters, some of the blocks may end up occupying more than 80 bytes in the file because of new-line characters that were expanded.

Q: Why are there two sets of file-positioning functions (`fseek`/`ftell` and `fsetpos`/`fgetpos`)? Wouldn't one set be enough? [p. 574]

A: `fseek` and `ftell` have been part of the C library for eons. They have one drawback, though: they assume that a file position will fit in a `long int` value. Since `long int` is typically a 32-bit type, this means that `fseek` and `ftell` may not work with files containing more than 2,147,483,647 bytes. In recognition of this problem, `fsetpos` and `fgetpos` were added to `<stdio.h>` when C89 was created. These functions aren't required to treat file positions as numbers, so they're not subject to the `long int` restriction. But don't assume that you have to use `fsetpos` and `fgetpos`; if your implementation supports a 64-bit `long int` type, `fseek` and `ftell` are fine even for very large files.

Q: Why doesn't this chapter discuss screen control: moving the cursor, changing the colors of characters on the screen, and so on?

A: C provides no standard functions for screen control. The C standard addresses only issues that can reasonably be standardized across a wide range of computers and operating systems; screen control is outside this realm. The customary way to solve this problem in UNIX is to use the `curses` library, which supports screen control in a terminal-independent manner.

Similarly, there are no standard functions for building programs with a graphical user interface. However, you can most likely use C function calls to access the windowing API (application programming interface) for your operating system.

Exercises

Section 22.1

1. Indicate whether each of the following files is more likely to contain text data or binary data:
   (a) A file of object code produced by a C compiler
   (b) A program listing produced by a C compiler
   (c) An email message sent from one computer to another
   (d) A file containing a graphics image

Section 22.2

W 2. Indicate which mode string is most likely to be passed to `fopen` in each of the following situations:
   (a) A database management system opens a file containing records to be updated.
   (b) A mail program opens a file of saved messages so that it can add additional messages to the end.
   (c) A graphics program opens a file containing a picture to be displayed on the screen.
   (d) An operating system command interpreter opens a "shell script" (or "batch file") containing commands to be executed.

3. Find the error in the following program fragment and show how to fix it.

```
FILE *fp;

if (fp = fopen(filename, "r")) {
   read characters until end-of-file
}
fclose(fp);
```

Section 22.3

W 4. Show how each of the following numbers will look if displayed by `printf` with `%#012.5g` as the conversion specification:
   (a) 83.7361
   (b) 29748.6607
   (c) 1054932234.0
   (d) 0.0000235218

5. Is there any difference between the `printf` conversion specifications `%.4d` and `%04d`? If so, explain what it is.

W *6. Write a call of `printf` that prints

```
1 widget
```

if the `widget` variable (of type `int`) has the value 1, and

```
n widgets
```

otherwise, where *n* is the value of `widget`. You are not allowed to use the `if` statement or any other statement; the answer must be a single call of `printf`.

*7. Suppose that we call `scanf` as follows:

```
n = scanf("%d%f%d", &i, &x, &j);
```

(`i`, `j`, and `n` are `int` variables and `x` is a `float` variable.) Assuming that the input stream contains the characters shown, give the values of `i`, `j`, `n`, and `x` after the call. In addition, indicate which characters were consumed by the call.

(a) `10•20•30¤`
(b) `1.0•2.0•3.0¤`
(c) `0.1•0.2•0.3¤`
(d) `.1•.2•.3¤`

W 8. In previous chapters, we've used the `scanf` format string `" %c"` when we wanted to skip white-space characters and read a nonblank character. Some programmers use `"%1s"` instead. Are the two techniques equivalent? If not, what are the differences?

Section 22.4

9. Which one of the following calls is *not* a valid way of reading one character from the standard input stream?

(a) `getch()`
(b) `getchar()`
(c) `getc(stdin)`
(d) `fgetc(stdin)`

W 10. The `fcopy.c` program has one minor flaw: it doesn't check for errors as it's writing to the destination file. Errors during writing are rare, but do occasionally occur (the disk might become full, for example). Show how to add the missing error check to the program, assuming that we want it to display a message and terminate immediately if an error occurs.

11. The following loop appears in the `fcopy.c` program:

```
while ((ch = getc(source_fp)) != EOF)
  putc(ch, dest_fp);
```

Suppose that we neglected to put parentheses around `ch = getc(source_fp)`:

```
while (ch = getc(source_fp) != EOF)
  putc(ch, dest_fp);
```

Would the program compile without an error? If so, what would the program do when it's run?

12. Find the error in the following function and show how to fix it.

```
int count_periods(const char *filename)
{
  FILE *fp;
  int n = 0;
```

```
  if ((fp = fopen(filename, "r")) != NULL) {
    while (fgetc(fp) != EOF)
      if (fgetc(fp) == '.')
        n++;
    fclose(fp);
  }

  return n;
}
```

13. Write the following function:

```
int line_length(const char *filename, int n);
```

The function should return the length of line `n` in the text file whose name is `filename` (assuming that the first line in the file is line 1). If the line doesn't exist, the function should return 0.

Section 22.5

W 14. (a) Write your own version of the `fgets` function. Make it behave as much like the real `fgets` function as possible; in particular, make sure that it has the proper return value. To avoid conflicts with the standard library, don't name your function `fgets`.

(b) Write your own version of `fputs`, following the same rules as in part (a).

Section 22.7

W 15. Write calls of `fseek` that perform the following file-positioning operations on a binary file whose data is arranged in 64-byte "records." Use `fp` as the file pointer in each case.

(a) Move to the beginning of record `n`. (Assume that the first record in the file is record 0.)
(b) Move to the beginning of the last record in the file.
(c) Move forward one record.
(d) Move backward two records.

Section 22.8

16. Assume that `str` is a string that contains a "sales rank" immediately preceded by the # symbol (other characters may precede the # and/or follow the sales rank). A sales rank is a series of decimal digits possibly containing commas, such as the following examples:

```
989
24,675
1,162,620
```

Write a call of `sscanf` that extracts the sales rank (but not the # symbol) and stores it in a string variable named `sales_rank`.

Programming Projects

1. Extend the `canopen.c` program of Section 22.2 so that the user may put any number of file names on the command line:

```
canopen foo bar baz
```

The program should print a separate `can be opened` or `can't be opened` message for each file. Have the program terminate with status `EXIT_FAILURE` if one or more of the files can't be opened.

W 2. Write a program that converts all letters in a file to upper case. (Characters other than letters shouldn't be changed.) The program should obtain the file name from the command line and write its output to `stdout`.

3. Write a program named `fcat` that "concatenates" any number of files by writing them to standard output, one after the other, with no break between files. For example, the following command will display the files `f1.c`, `f2.c`, and `f3.c` on the screen:

```
fcat f1.c f2.c f3.c
```

`fcat` should issue an error message if any file can't be opened. *Hint:* Since it has no more than one file open at a time, `fcat` needs only a single file pointer variable. Once it's finished with a file, `fcat` can use the same variable when it opens the next file.

W 4. (a) Write a program that counts the number of characters in a text file.

(b) Write a program that counts the number of words in a text file. (A "word" is any sequence of non-white-space characters.)

(c) Write a program that counts the number of lines in a text file.

Have each program obtain the file name from the command line.

5. The `xor.c` program of Section 20.1 refuses to encrypt bytes that—in original or encrypted form—are control characters. We can now remove this restriction. Modify the program so that the names of the input and output files are command-line arguments. Open both files in binary mode, and remove the test that checks whether the original and encrypted characters are printing characters.

W 6. Write a program that displays the contents of a file as bytes and as characters. Have the user specify the file name on the command line. Here's what the output will look like when the program is used to display the `pun.c` file of Section 2.1:

```
Offset                Bytes                Characters
------  ------------------------------------  ----------
     0  23 69 6E 63 6C 75 64 65 20 3C  #include <
    10  73 74 64 69 6F 2E 68 3E 0D 0A  stdio.h>..
    20  0D 0A 69 6E 74 20 6D 61 69 6E  ..int main
    30  28 76 6F 69 64 29 0D 0A 7B 0D  (void)..{.
    40  0A 20 20 70 72 69 6E 74 66 28  .  printf(
    50  22 54 6F 20 43 2C 20 6F 72 20  "To C, or
    60  6E 6F 74 20 74 6F 20 43 3A 20  not to C:
    70  74 68 61 74 20 69 73 20 74 68  that is th
    80  65 20 71 75 65 73 74 69 6F 6E  e question
    90  2E 5C 6E 22 29 3B 0D 0A 20 20  .\n");..
   100  72 65 74 75 72 6E 20 30 3B 0D  return 0;.
   110  0A 7D                           .}
```

Each line shows 10 bytes from the file, as hexadecimal numbers and as characters. The number in the `Offset` column indicates the position within the file of the first byte on the line. Only printing characters (as determined by the `isprint` function) are displayed; other characters are shown as periods. Note that the appearance of a text file may vary, depending on the character set and the operating system. The example above assumes that `pun.c` is a Windows file, so 0D and 0A bytes (the ASCII carriage-return and line-feed characters) appear at the end of each line. *Hint:* Be sure to open the file in `"rb"` mode.

7. Of the many techniques for compressing the contents of a file, one of the simplest and fastest is known as ***run-length encoding***. This technique compresses a file by replacing sequences of identical bytes by a pair of bytes: a repetition count followed by a byte to be repeated. For example, suppose that the file to be compressed begins with the following sequence of bytes (shown in hexadecimal):

```
46 6F 6F 20 62 61 72 21 21 21 20 20 20 20 20
```

The compressed file will contain the following bytes:

```
01 46 02 6F 01 20 01 62 01 61 01 72 03 21 05 20
```

Run-length encoding works well if the original file contains many long sequences of identical bytes. In the worst case (a file with no repeated bytes), run-length encoding can actually double the length of the file.

(a) Write a program named `compress_file` that uses run-length encoding to compress a file. To run `compress_file`, we'd use a command of the form

`compress_file` *original-file*

`compress_file` will write the compressed version of *original-file* to *original-file*`.rle`.

For example, the command

```
compress_file foo.txt
```

will cause `compress_file` to write a compressed version of `foo.txt` to a file named `foo.txt.rle`. *Hint:* The program described in Programming Project 6 could be useful for debugging.

(b) Write a program named `uncompress_file` that reverses the compression performed by the `compress_file` program. The `uncompress_file` command will have the form

`uncompress_file` *compressed-file*

compressed-file should have the extension `.rle`. For example, the command

```
uncompress_file foo.txt.rle
```

will cause `uncompress_file` to open the file `foo.txt.rle` and write an uncompressed version of its contents to `foo.txt`. `uncompress_file` should display an error message if its command-line argument doesn't end with the `.rle` extension.

8. Modify the `inventory.c` program of Section 16.3 by adding two new operations:

   - Save the database in a specified file.
   - Load the database from a specified file.

   Use the codes `d` (dump) and `r` (restore), respectively, to represent these operations. The interaction with the user should have the following appearance:

```
Enter operation code: d
Enter name of output file: inventory.dat

Enter operation code: r
Enter name of input file: inventory.dat
```

   Hint: Use `fwrite` to write the array containing the parts to a binary file. Use `fread` to restore the array by reading it from a file.

W 9. Write a program that merges two files containing part records stored by the `inventory.c` program (see Programming Project 8). Assume that the records in each file are sorted by part number, and that we want the resulting file to be sorted as well. If both files have a part with the same number, combine the quantities stored in the records. (As a consistency check, have the program compare the part names and print an error message if they don't match.) Have the program obtain the names of the input files and the merged file from the command line.

*10. Modify the `inventory2.c` program of Section 17.5 by adding the `d` (dump) and `r` (restore) operations described in Programming Project 8. Since the part structures aren't stored in an array, the `d` operation can't save them all by a single call of `fwrite`. Instead, it will need to visit each node in the linked list, writing the part number, part name, and quan-

tity on hand to a file. (Don't save the `next` pointer; it won't be valid once the program terminates.) As it reads parts from a file, the `r` operation will rebuild the list one node at a time.

11. Write a program that reads a date from the command line and displays it in the following form:

    ```
    September 13, 2010
    ```

    Allow the user to enter the date as either `9-13-2010` or `9/13/2010`; you may assume that there are no spaces in the date. Print an error message if the date doesn't have one of the specified forms. *Hint:* Use `sscanf` to extract the month, day, and year from the command-line argument.

12. Modify Programming Project 2 from Chapter 3 so that the program reads a series of items from a file and displays the data in columns. Each line of the file will have the following form:

    item, price, mm/dd/yyyy

    For example, suppose that the file contains the following lines:

    ```
    583,13.5,10/24/2005
    3912,599.99,7/27/2008
    ```

    The output of the program should have the following appearance:

    ```
    Item          Unit          Purchase
                  Price         Date
    583           $  13.50      10/24/2005
    3912          $ 599.99      7/27/2008
    ```

    Have the program obtain the file name from the command line.

13. Modify Programming Project 8 from Chapter 5 so that the program obtains departure and arrival times from a file named `flights.dat`. Each line of the file will contain a departure time followed by an arrival time, with one or more spaces separating the two. Times will be expressed using the 24-hour clock. For example, here's what `flights.dat` might look like if it contained the flight information listed in the original project:

    ```
    8:00 10:16
    9:43 11:52
    11:19 13:31
    12:47 15:00
    14:00 16:08
    15:45 17:55
    19:00 21:20
    21:45 23:58
    ```

14. Modify Programming Project 15 from Chapter 8 so that the program prompts the user to enter the name of a file containing the message to be encrypted:

    ```
    Enter name of file to be encrypted: message.txt
    Enter shift amount (1-25): 3
    ```

    The program then writes the encrypted message to a file with the same name but an added extension of `.enc`. In this example, the original file name is `message.txt`, so the encrypted message will be stored in a file named `message.txt.enc`. There's no limit on the size of the file to be encrypted or on the length of each line in the file.

15. Modify the `justify` program of Section 15.3 so that it reads from one text file and writes to another. Have the program obtain the names of both files from the command line.

16. Modify the `fcopy.c` program of Section 22.4 so that it uses `fread` and `fwrite` to copy the file in blocks of 512 bytes. (The last block may contain fewer than 512 bytes, of course.)

17. Write a program that reads a series of phone numbers from a file and displays them in a standard format. Each line of the file will contain a single phone number, but the numbers may be in a variety of formats. You may assume that each line contains 10 digits, possibly mixed with other characters (which should be ignored). For example, suppose that the file contains the following lines:

```
404.817.6900
(215) 686-1776
312-746-6000
877 275 5273
6173434200
```

The output of the program should have the following appearance:

```
(404) 817-6900
(215) 686-1776
(312) 746-6000
(877) 275-5273
(617) 343-4200
```

Have the program obtain the file name from the command line.

18. Write a program that reads integers from a text file whose name is given as a command-line argument. Each line of the file may contain any number of integers (including none) separated by one or more spaces. Have the program display the largest number in the file, the smallest number, and the median (the number closest to the middle if the integers were sorted). If the file contains an even number of integers, there will be two numbers in the middle; the program should display their average (rounded down). You may assume that the file contains no more than 10,000 integers. *Hint:* Store the integers in an array and then sort the array.

19. (a) Write a program that converts a Windows text file to a UNIX text file. (See Section 22.1 for a discussion of the differences between Windows and UNIX text files.)

(b) Write a program that converts a UNIX text file to a Windows text file.

In each case, have the program obtain the names of both files from the command line. *Hint:* Open the input file in `"rb"` mode and the output file in `"wb"` mode.

23 Library Support for Numbers and Character Data

Prolonged contact with the computer turns mathematicians into clerks and vice versa.

This chapter describes the five most important library headers that provide support for working with numbers, characters, and character strings. Sections 23.1 and 23.2 cover the `<float.h>` and `<limits.h>` headers, which contain macros describing the characteristics of numeric and character types. Sections 23.3 and 23.4 describe the `<math.h>` header, which provides mathematical functions. Section 23.3 discusses the C89 version of `<math.h>`; Section 23.4 covers the C99 additions, which are so extensive that I've chosen to cover them separately. Sections 23.5 and 23.6 are devoted to the `<ctype.h>` and `<string.h>` headers, which provide character functions and string functions, respectively.

C99 adds several headers that also deal with numbers, characters, and strings. The `<wchar.h>` and `<wctype.h>` headers are discussed in Chapter 25. Chapter 27 covers `<complex.h>`, `<fenv.h>`, `<inttypes.h>`, `<stdint.h>`, and `<tgmath.h>`.

23.1 The `<float.h>` Header: Characteristics of Floating Types

The `<float.h>` header provides macros that define the range and accuracy of the `float`, `double`, and `long double` types. There are no types or functions in `<float.h>`.

Two macros apply to all floating types. The `FLT_ROUNDS` macro represents the current rounding direction for floating-point addition. Table 23.1 shows the possible values of `FLT_ROUNDS`. (Values not shown in the table indicate implementation-defined rounding behavior.)

rounding direction ➤23.4

Table 23.1
Rounding Directions

Value	*Meaning*
-1	Indeterminable
0	Toward zero
1	To nearest
2	Toward positive infinity
3	Toward negative infinity

Unlike the other macros in `<float.h>`, which represent constant expressions, the value of `FLT_ROUNDS` may change during execution. (The `fesetround` function allows a program to change the current rounding direction.) The other macro, `FLT_RADIX`, specifies the radix of exponent representation; it has a minimum value of 2 (indicating binary representation).

`fesetround` function ➤27.6

The remaining macros, which I'll present in a series of tables, describe the characteristics of specific types. Each macro begins with either `FLT`, `DBL`, or `LDBL`, depending on whether it refers to the `float`, `double`, or `long double` type. The C standard provides extremely detailed definitions of these macros; my descriptions will be less precise but easier to understand. The tables indicate maximum or minimum values for some macros, as specified in the standard.

Table 23.2 lists macros that define the number of significant digits guaranteed by each floating type.

Table 23.2
Significant-Digit Macros in `<float.h>`

Name	*Value*	*Description*

`FLT_MANT_DIG`
`DBL_MANT_DIG`
`LDBL_MANT_DIG`
		Number of significant digits (base `FLT_RADIX`)

`FLT_DIG`	≥6	Number of significant digits (base 10)
`DBL_DIG`	≥10	
`LDBL_DIG`	≥10	

Table 23.3 lists macros having to do with exponents.

Table 23.3
Exponent Macros in `<float.h>`

Name	*Value*	*Description*

`FLT_MIN_EXP`
`DBL_MIN_EXP`
`LDBL_MIN_EXP`
		Smallest (most negative) power to which `FLT_RADIX` can be raised

`FLT_MIN_10_EXP`	≤–37	Smallest (most negative) power to which 10 can be raised
`DBL_MIN_10_EXP`	≤–37	
`LDBL_MIN_10_EXP`	≤–37	

`FLT_MAX_EXP`
`DBL_MAX_EXP`
`LDBL_MAX_EXP`
		Largest power to which `FLT_RADIX` can be raised

`FLT_MAX_10_EXP`	≥+37	Largest power to which 10 can be raised
`DBL_MAX_10_EXP`	≥+37	
`LDBL_MAX_10_EXP`	≥+37	

Table 23.4 lists macros that describe how large numbers can be, how close to zero they can get, and how close two consecutive numbers can be.

Table 23.4
Max, Min, and Epsilon Macros in <float.h>

Name	*Value*	*Description*
FLT_MAX	$\geq 10^{+37}$	Largest finite value
DBL_MAX	$\geq 10^{+37}$	
LDBL_MAX	$\geq 10^{+37}$	
FLT_MIN	$\leq 10^{-37}$	Smallest positive value
DBL_MIN	$\leq 10^{-37}$	
LDBL_MIN	$\leq 10^{-37}$	
FLT_EPSILON	$\leq 10^{-5}$	Smallest representable difference between two numbers
DBL_EPSILON	$\leq 10^{-9}$	
LDBL_EPSILON	$\leq 10^{-9}$	

C99

C99 provides two other macros, DECIMAL_DIG and FLT_EVAL_METHOD. DECIMAL_DIG represents the number of significant digits (base 10) in the widest supported floating type; it has a minimum value of 10. The value of FLT_EVAL_METHOD indicates whether an implementation will perform floating-point arithmetic using greater range and precision than is strictly necessary. If this macro has the value 0, for example, then adding two float values would be done in the normal way. If it has the value 1, however, then the float values would be converted to double before the addition is performed. Table 23.5 lists the possible values of FLT_EVAL_METHOD. (Negative values not shown in the table indicate implementation-defined behavior.)

Table 23.5
Evaluation Methods

Value	*Meaning*
-1	Indeterminable
0	Evaluate all operations and constants just to the range and precision of the type
1	Evaluate operations and constants of type float and double to the range and precision of the double type
2	Evaluate all operations and constants to the range and precision of the long double type

Most of the macros in <float.h> are of interest only to experts in numerical analysis, making it probably one of the least-used headers in the standard library.

23.2 The <limits.h> Header: Sizes of Integer Types

The <limits.h> header provides macros that define the range of each integer type (including the character types). <limits.h> declares no types or functions.

One set of macros in <limits.h> deals with the character types: char, signed char, and unsigned char. Table 23.6 lists these macros and shows the maximum or minimum value of each.

The other macros in <limits.h> deal with the remaining integer types: short int, unsigned short int, int, unsigned int, long int, and

Table 23.6
Character Macros in `<limits.h>`

Name	*Value*	*Description*
`CHAR_BIT`	≥8	Number of bits per byte
`SCHAR_MIN`	≤–127	Minimum `signed char` value
`SCHAR_MAX`	≥+127	Maximum `signed char` value
`UCHAR_MAX`	≥255	Maximum `unsigned char` value
`CHAR_MIN`	†	Minimum `char` value
`CHAR_MAX`	††	Maximum `char` value
`MB_LEN_MAX`	≥1	Maximum number of bytes per multibyte character in any supported locale (see Section 25.2)

†`CHAR_MIN` is equal to `SCHAR_MIN` if `char` is treated as a signed type; otherwise, `CHAR_MIN` is 0.
††`CHAR_MAX` has the same value as either `SCHAR_MAX` or `UCHAR_MAX`, depending on whether `char` is treated as a signed type or an unsigned type.

`unsigned long int`. Table 23.7 lists these macros and shows the maximum or minimum value of each; the formula used to compute each value is also given. Note that C99 provides three macros that describe the characteristics of the `long long int` types.

C99

Table 23.7
Integer Macros in `<limits.h>`

Name	*Value*	*Formula*	*Description*
`SHRT_MIN`	≤–32767	$-(2^{15}-1)$	Minimum `short int` value
`SHRT_MAX`	≥+32767	$2^{15}-1$	Maximum `short int` value
`USHRT_MAX`	≥65535	$2^{16}-1$	Maximum `unsigned short int` value
`INT_MIN`	≤–32767	$-(2^{15}-1)$	Minimum `int` value
`INT_MAX`	≥+32767	$2^{15}-1$	Maximum `int` value
`UINT_MAX`	≥65535	$2^{16}-1$	Maximum `unsigned int` value
`LONG_MIN`	≤–2147483647	$-(2^{31}-1)$	Minimum `long int` value
`LONG_MAX`	≥+2147483647	$2^{31}-1$	Maximum `long int` value
`ULONG_MAX`	≥4294967295	$2^{32}-1$	Maximum `unsigned long int` value
`LLONG_MIN`†	≤–9223372036854775807	$-(2^{63}-1)$	Minimum `long long int` value
`LLONG_MAX`†	≥+9223372036854775807	$2^{63}-1$	Maximum `long long int` value
`ULLONG_MAX`†	≥18446744073709551615	$2^{64}-1$	Maximum `unsigned long long int` value

†C99 only

The macros in `<limits.h>` are handy for checking whether a compiler supports integers of a particular size. For example, to determine whether the `int` type can store numbers as large as 100,000, we might use the following preprocessing directives:

```
#if INT_MAX < 100000
#error int type is too small
#endif
```

`#error` directive ➤14.5

If the `int` type isn't adequate, the `#error` directive will cause the preprocessor to display an error message.

Going a step further, we might use the macros in <limits.h> to help a program *choose* how to represent a type. Let's say that variables of type Quantity must be able to hold integers as large as 100,000. If INT_MAX is at least 100,000, we can define Quantity to be int; otherwise, we'll need to make it long int:

```
#if INT_MAX >= 100000
typedef int Quantity;
#else
typedef long int Quantity;
#endif
```

23.3 The <math.h> Header (C89): Mathematics

The functions in the C89 version of <math.h> fall into five groups:

Trigonometric functions
Hyperbolic functions
Exponential and logarithmic functions
Power functions
Nearest integer, absolute value, and remainder functions

C99 adds a number of functions to these groups as well as introducing other categories of math functions. The C99 changes to <math.h> are so extensive that I've chosen to cover them in a separate section that follows this one. That way, readers who are primarily interested in the C89 version of the header—or who are using a compiler that doesn't support C99—won't be overwhelmed by all the C99 additions.

Before we delve into the functions provided by <math.h>, let's take a brief look at how these functions deal with errors.

Errors

The <math.h> functions handle errors in a way that's different from other library functions. When an error occurs, most <math.h> functions store an error code in a special variable named errno (declared in the <errno.h> header). In addition, when the return value of a function would be larger than the largest double value, the functions in <math.h> return a special value, represented by the macro HUGE_VAL (defined in <math.h>). HUGE_VAL is of type double, but it isn't necessarily an ordinary number. (The IEEE standard for floating-point arithmetic defines a value named "infinity"—a logical choice for HUGE_VAL.)

<errno.h> header ➤24.2

infinity ➤23.4

The functions in <math.h> detect two kinds of errors:

- ***Domain error:*** An argument is outside a function's domain. If a domain error occurs, the function's return value is implementation-defined and EDOM

("domain error") is stored in `errno`. In some implementations of `<math.h>`, functions return a special value known as NaN ("not a number") when a domain error occurs.

NaN ➤23.4

- ***Range error:*** The return value of a function is outside the range of `double` values. If the return value's magnitude is too large (overflow), the function returns positive or negative `HUGE_VAL`, depending on the sign of the correct result. In addition, `ERANGE` ("range error") is stored in `errno`. If the return value's magnitude is too small to represent (underflow), the function returns zero; some implementations may also store `ERANGE` in `errno`.

underflow ➤23.4

We'll ignore the possibility of error for the remainder of this section. However, the function descriptions in Appendix D explain the circumstances that lead to each type of error.

Trigonometric Functions

```
double acos(double x);
double asin(double x);
double atan(double x);
double atan2(double y, double x);
double cos(double x);
double sin(double x);
double tan(double x);
```

cos
sin
tan

The `cos`, `sin`, and `tan` functions compute the cosine, sine, and tangent, respectively. If `PI` is defined to be `3.14159265`, passing `PI/4` to `cos`, `sin`, and `tan` produces the following results:

```
cos(PI/4) ⇒ 0.707107
sin(PI/4) ⇒ 0.707107
tan(PI/4) ⇒ 1.0
```

Note that arguments to `cos`, `sin`, and `tan` are expressed in radians, not degrees.

acos
asin
atan

`acos`, `asin`, and `atan` compute the arc cosine, arc sine, and arc tangent:

```
acos(1.0) ⇒ 0.0
asin(1.0) ⇒ 1.5708
atan(1.0) ⇒ 0.785398
```

Applying `acos` to a value returned by `cos` won't necessarily yield the original argument to `cos`, since `acos` always returns a value between 0 and π. `asin` and `atan` return a value between $-\pi/2$ and $\pi/2$.

atan2

`atan2` computes the arc tangent of `y/x`, where `y` is the function's first argument and `x` is its second. The return value of `atan2` is between $-\pi$ and π. The call `atan(x)` is equivalent to `atan2(x, 1.0)`.

Hyperbolic Functions

```
double cosh(double x);
double sinh(double x);
double tanh(double x);
```

cosh
sinh
tanh

The cosh, sinh, and tanh functions compute the hyperbolic cosine, sine, and tangent:

```
cosh(0.5) ⇒ 1.12763
sinh(0.5) ⇒ 0.521095
tanh(0.5) ⇒ 0.462117
```

Arguments to cosh, sinh, and tanh must be expressed in radians, not degrees.

Exponential and Logarithmic Functions

```
double exp(double x);
double frexp(double value, int *exp);
double ldexp(double x, int exp);
double log(double x);
double log10(double x);
double modf(double value, double *iptr);
```

exp

The exp function returns *e* raised to a power:

```
exp(3.0) ⇒ 20.0855
```

log
log10

log is the inverse of exp—it computes the logarithm of a number to the base *e*. log10 computes the "common" (base 10) logarithm:

```
log(20.0855) ⇒ 3.0
log10(1000)  ⇒ 3.0
```

Computing the logarithm to a base other than *e* or 10 isn't difficult. The following function, for example, computes the logarithm of x to the base b, for arbitrary x and b:

```
double log_base(double x, double b)
{
  return log(x) / log(b);
}
```

modf

The modf and frexp functions decompose a double value into two parts. modf splits its first argument into integer and fractional parts. It returns the fractional part and stores the integer part in the object pointed to by the second argument:

`modf(3.14159, &int_part)` ⇒ 0.14159 (`int_part` is assigned 3.0)

Although `int_part` must have type `double`, we can always cast it to `int` or `long int` later.

frexp The `frexp` function splits a floating-point number into a fractional part f and an exponent n in such a way that the original number equals $f \times 2^n$, where either $0.5 \le f < 1$ or $f = 0$. `frexp` returns f and stores n in the (integer) object pointed to by the second argument:

```
frexp(12.0, &exp)  ⇒  .75 (exp is assigned 4)
frexp(0.25, &exp)  ⇒  0.5 (exp is assigned -1)
```

ldexp `ldexp` undoes the work of `frexp` by combining a fraction and an exponent into a single number:

```
ldexp(.75, 4)   ⇒  12.0
ldexp(0.5, -1)  ⇒  0.25
```

In general, the call `ldexp(x, exp)` returns $x \times 2^{exp}$.

The `modf`, `frexp`, and `ldexp` functions are primarily used by other functions in `<math.h>`. They are rarely called directly by programs.

Power Functions

```
double pow(double x, double y);
double sqrt(double x);
```

pow The `pow` function raises its first argument to the power specified by its second argument:

```
pow(3.0, 2.0)   ⇒  9.0
pow(3.0, 0.5)   ⇒  1.73205
pow(3.0, -3.0)  ⇒  0.037037
```

sqrt `sqrt` computes the square root:

```
sqrt(3.0)  ⇒  1.73205
```

Using `sqrt` to find square roots is preferable to calling `pow`, since `sqrt` is usually a much faster function.

Nearest Integer, Absolute Value, and Remainder Functions

```
double ceil(double x);
double fabs(double x);
double floor(double x);
double fmod(double x, double y);
```

ceil
floor

The `ceil` ("ceiling") function returns—as a `double` value—the smallest integer that's greater than or equal to its argument. `floor` returns the largest integer that's less than or equal to its argument:

```
ceil(7.1)   ⇒  8.0
ceil(7.9)   ⇒  8.0
ceil(-7.1)  ⇒ -7.0
ceil(-7.9)  ⇒ -7.0

floor(7.1)  ⇒  7.0
floor(7.9)  ⇒  7.0
floor(-7.1) ⇒ -8.0
floor(-7.9) ⇒ -8.0
```

In other words, `ceil` "rounds up" to the nearest integer, while `floor` "rounds down." C89 lacks a standard function that rounds to the nearest integer, but we can easily use `ceil` and `floor` to write our own:

```
double round_nearest(double x)
{
  return x < 0.0 ? ceil(x - 0.5) : floor(x + 0.5);
}
```

C99 C99 provides several functions that round to the nearest integer, as we'll see in the next section.

fabs `fabs` computes the absolute value of a number:

```
fabs(7.1)  ⇒ 7.1
fabs(-7.1) ⇒ 7.1
```

fmod `fmod` returns the remainder when its first argument is divided by its second argument:

```
fmod(5.5, 2.2) ⇒ 1.1
```

C doesn't allow the `%` operator to have floating-point operands, but `fmod` is a more-than-adequate substitute.

23.4 The <math.h> Header (C99): Mathematics

The C99 version of the `<math.h>` header includes the entire C89 version, plus a host of additional types, macros, and functions. The changes to this header are so numerous that I've chosen to cover them separately. There are several reasons why the standards committee added so many capabilities to `<math.h>`:

- ***Provide better support for the IEEE floating-point standard.*** C99 doesn't mandate the use of the IEEE standard; other ways of representing floating-point

numbers are permitted. However, it's safe to say that the vast majority of C programs are executed on systems that support this standard.

- ***Provide more control over floating-point arithmetic.*** Better control over floating-point arithmetic may allow programs to achieve greater accuracy and speed.
- ***Make C more attractive to Fortran programmers.*** The addition of many math functions, along with enhancements elsewhere in C99 (such as support for complex numbers), was intended to increase C's appeal to programmers who might have used other programming languages (primarily Fortran) in the past.

Another reason that I've decided to cover C99's `<math.h>` header in a separate section is that it's not likely to be of much interest to the average C programmer. Those using C for its traditional applications, which include systems programming and embedded systems, probably won't need the additional functions that C99 provides. However, programmers developing engineering, mathematics, or science applications may find these functions to be quite useful.

IEEE Floating-Point Standard

One motivation for the changes to the `<math.h>` header is better support for IEEE Standard 754, the most widely used representation for floating-point numbers. The full title of the standard is "IEEE Standard for Binary Floating-Point Arithmetic" (ANSI/IEEE Standard 754-1985). It's also known as IEC 60559, which is how the C99 standard refers to it.

Section 7.2 described some of the basic properties of the IEEE standard. We saw that the standard provides two primary formats for floating-point numbers: single precision (32 bits) and double precision (64 bits). Numbers are stored in a form of scientific notation, with each number having three parts: a sign, an exponent, and a fraction. That limited knowledge of the IEEE standard is enough to use the C89 version of `<math.h>` effectively. Understanding the C99 version, however, requires knowing more about the standard. Here's some additional information that we'll need:

- ***Positive/negative zero.*** One of the bits in the IEEE representation of a floating-point number represents the number's sign. As a result, the number zero can be either positive or negative, depending on the value of this bit. The fact that zero has two representations may sometimes require us to treat it differently from other floating-point numbers.
- ***Subnormal numbers.*** When a floating-point operation is performed, the result may be too small to represent, a condition known as ***underflow***. Think of what happens if you repeatedly divide a number using a hand calculator: eventually the result is zero, because it becomes too small to represent using the calculator's number representation. The IEEE standard has a way to reduce the impact of this phenomenon. Ordinary floating-point numbers are stored in a "normalized" format, in which the number is scaled so that there's exactly one

digit to the left of the binary point. When a number gets small enough, however, it's stored in a different format in which it's not normalized. These ***subnormal numbers*** (also known as ***denormalized numbers*** or ***denormals***) can be much smaller than normalized numbers; the trade-off is that they get progressively less accurate as they get smaller.

- ***Special values.*** Each floating-point format allows the representation of three special values: ***positive infinity, negative infinity,*** and ***NaN*** ("not a number"). Dividing a positive number by zero produces positive infinity. Dividing a negative number by zero yields negative infinity. The result of a mathematically undefined operation, such as dividing zero by zero, is NaN. (It's more accurate to say "the result is *a* NaN" rather than "the result is NaN," because the IEEE standard has multiple representations for NaN. The exponent part of a NaN value is all 1 bits, but the fraction can be any nonzero sequence of bits.) Special values can be operands in subsequent operations. Infinity behaves just as it does in ordinary mathematics. For example, dividing a positive number by positive infinity yields zero. (Note that an arithmetic expression could produce infinity as an intermediate result but have a noninfinite value overall.) Performing any operation on NaN gives NaN as the result.
- ***Rounding direction.*** When a number can't be stored exactly using a floating-point representation, the current ***rounding direction*** (or ***rounding mode***) determines which floating-point value will be selected to represent the number. There are four rounding directions: (1) *Round toward nearest.* Rounds to the nearest representable value. If a number falls halfway between two values, it is rounded to the "even" value (the one whose least significant bit is zero). (2) *Round toward zero.* (3) *Round toward positive infinity.* (4) *Round toward negative infinity.* The default rounding direction is round toward nearest.
- ***Exceptions.*** There are five types of floating-point exceptions: *overflow, underflow, division by zero, invalid operation* (the result of an arithmetic operation was NaN), and *inexact* (the result of an arithmetic operation had to be rounded). When one of these conditions is detected, we say that the exception is ***raised.***

Types

C99 adds two types, `float_t` and `double_t`, to `<math.h>`. The `float_t` type is at least as "wide" as the `float` type (meaning that it could be the `float` type or any wider type, such as `double`). Similarly, `double_t` is required to be at least as wide as the `double` type. (It must also be at least as wide as `float_t`.) These types are provided for the programmer who's trying to maximize the performance of floating-point arithmetic. `float_t` should be the most efficient floating-point type that's at least as wide as `float`; `double_t` should be the most efficient floating-point type that's at least as wide as `double`.

FLT_EVAL_METHOD ➤23.1

The `float_t` and `double_t` types are related to the `FLT_EVAL_METHOD` macro, as shown in Table 23.8.

Table 23.8 Relationship between FLT_EVAL_METHOD and the float_t and double_t Types

Value of FLT_EVAL_METHOD	*Meaning of* float_t	*Meaning of* double_t
0	float	double
1	double	double
2	long double	long double
Other	Implementation-defined	Implementation-defined

Macros

C99 adds a number of macros to <math.h>. I'll mention just two of them at this point. INFINITY represents the float version of positive or unsigned infinity. (If the implementation doesn't support infinity, then INFINITY represents a float value that overflows at compile time.) The NAN macro represents the float version of "not a number." More specifically, it represents a "quiet" NaN (one that doesn't raise an exception if used in an arithmetic expression). If quiet NaNs aren't supported, the NAN macro won't be defined.

I'll cover the function-like macros in <math.h> later in the section, along with ordinary functions. Macros that are relevant only to a specific function will be described with the function itself.

Errors

For the most part, the C99 version of <math.h> deals with errors in the same way as the C89 version. However, there are a few twists that we'll need to discuss.

First, C99 provides several macros that give implementations a choice of how errors are signaled: via a value stored in errno, via a floating-point exception, or both. The macros MATH_ERRNO and MATH_ERREXCEPT represent the integer constants 1 and 2, respectively. A third macro, math_errhandling, represents an int expression whose value is either MATH_ERRNO, MATH_ERREXCEPT, or the bitwise OR of the two values. (It's also possible that math_errhandling isn't really a macro; it might be an identifier with external linkage.) The value of math_errhandling can't be changed within a program.

Now, let's see what happens when a domain error occurs during a call of one of the functions in <math.h>. The C89 standard says that EDOM is stored in errno. The C99 standard, on the other hand, states that if the expression math_errhandling & MATH_ERRNO is nonzero (i.e., the MATH_ERRNO bit is set), then EDOM is stored in errno. If the expression math_errhandling & MATH_ERREXCEPT is nonzero, the *invalid* floating-point exception is raised. Thus, either or both actions are possible, depending on the value of math_errhandling.

Finally, let's turn to the actions that take place when a range error is detected during a function call. There are two cases, based on the magnitude of the function's return value.

Overflow. If the magnitude is too large, the C89 standard requires the function to return positive or negative HUGE_VAL, depending on the sign of the correct

result. In addition, ERANGE is stored in errno. The C99 standard describes a more complicated set of actions when overflow occurs:

- If default rounding is in effect or if the return value is an "exact infinity" (such as log(0.0)), then the function returns either HUGE_VAL, HUGE_VALF, or HUGE_VALL, depending on the function's return type. (HUGE_VALF and HUGE_VALL—the float and long double versions of HUGE_VAL—are new in C99. Like HUGE_VAL, they may represent positive infinity.) The value returned has the sign of the correct result.
- If the value of math_errhandling & MATH_ERRNO is nonzero, ERANGE is stored in errno.
- If the value of math_errhandling & MATH_ERREXCEPT is nonzero, the *divide-by-zero* floating-point exception is raised if the mathematical result is an exact infinity. Otherwise, the *overflow* exception is raised.

Underflow. If the magnitude is too small to represent, the C89 standard requires the function to return zero; some implementations may also store ERANGE in errno. The C99 standard prescribes a somewhat different set of actions:

- The function returns a value whose magnitude is less than or equal to the smallest normalized positive number belonging to the function's return type. (This value might be zero or a subnormal number.)
- If the value of math_errhandling & MATH_ERRNO is nonzero, an implementation may store ERANGE in errno.
- If the value of math_errhandling & MATH_ERREXCEPT is nonzero, an implementation may raise the *underflow* floating-point exception.

Notice the word "may" in the latter two cases. For reasons of efficiency, an implementation is not required to modify errno or raise the *underflow* exception.

Functions

We're now ready to tackle the functions that C99 adds to <math.h>. I'll present the functions in groups, using the same categories as the C99 standard. These categories differ somewhat from the ones in Section 23.3, which came from the C89 standard.

One of the biggest changes in the C99 version of <math.h> is the addition of two more versions of most functions. In C89, there's only a single version of each math function; typically, it takes at least one argument of type double and/or returns a double value. In C99, however, there are two additional versions: one for float and one for long double. The names of these functions are identical to the name of the original function except for the addition of an f or l suffix. For example, the original sqrt function, which takes the square root of a double value, is now joined by sqrtf (the float version) and sqrtl (the long double version). I'll list the prototypes for the new versions (in italics, as is my custom for functions that are new in C99). I won't describe the functions further, though, since they're virtually identical to their C89 counterparts.

The C99 version of `<math.h>` also includes a number of completely new functions (and function-like macros). I'll give a brief description of each one. As in Section 23.3, I won't discuss error conditions for these functions, but Appendix D—which lists all standard library functions in alphabetical order—provides this information. I won't list the names of all the new functions in the left margin; instead, I'll show just the name of the primary function. For example, there are three new functions that compute the arc hyperbolic cosine: `acosh`, `acoshf`, and `acoshl`. I'll describe `acosh` and display only its name in the left margin.

Keep in mind that many of the new functions are highly specialized. As a result, the descriptions of these functions may seem sketchy. A discussion of what these functions are used for is outside the scope of this book.

Classification Macros

```
int fpclassify(real-floating x);
int isfinite(real-floating x);
int isinf(real-floating x);
int isnan(real-floating x);
int isnormal(real-floating x);
int signbit(real-floating x);
```

Our first category consists of function-like macros that are used to determine whether a floating-point value is a "normal" number or a special value such as infinity or NaN. The macros in this group are designed to accept arguments of any real floating type (`float`, `double`, or `long double`).

fpclassify

The `fpclassify` macro classifies its argument, returning the value of one of the number-classification macros shown in Table 23.9. An implementation may support other classifications by defining additional macros whose names begin with `FP_` and an upper-case letter.

Table 23.9 Number-Classification Macros

Name	*Meaning*
`FP_INFINITE`	Infinity (positive or negative)
`FP_NAN`	Not a number
`FP_NORMAL`	Normal (not zero, subnormal, infinite, or NaN)
`FP_SUBNORMAL`	Subnormal
`FP_ZERO`	Zero (positive or negative)

isfinite *isinf* *isnan* *isnormal*

The `isfinite` macro returns a nonzero value if its argument has a finite value (zero, subnormal, or normal, but not infinite or NaN). `isinf` returns a nonzero value if its argument has the value infinity (positive or negative). `isnan` returns a nonzero value if its argument is a NaN value. `isnormal` returns a nonzero value if its argument has a normal value (not zero, subnormal, infinite, or NaN).

signbit

The last classification macro is a bit different from the others. `signbit` returns a nonzero value if the sign of its argument is negative. The argument need not be a finite number; `signbit` also works for infinity and NaN.

Trigonometric Functions

```
float acosf(float x);                          see acos
long double acosl(long double x);              see acos

float asinf(float x);                          see asin
long double asinl(long double x);              see asin

float atanf(float x);                          see atan
long double atanl(long double x);              see atan

float atan2f(float y, float x);                see atan2
long double atan2l(long double y,
                   long double x);             see atan2

float cosf(float x);                           see cos
long double cosl(long double x);               see cos

float sinf(float x);                           see sin
long double sinl(long double x);               see sin

float tanf(float x);                           see tan
long double tanl(long double x);               see tan
```

The only new trigonometric functions in C99 are analogs of C89 functions. For descriptions, see the corresponding functions in Section 23.3.

Hyperbolic Functions

```
double acosh(double x);
float acoshf(float x);
long double acoshl(long double x);

double asinh(double x);
float asinhf(float x);
long double asinhl(long double x);

double atanh(double x);
float atanhf(float x);
long double atanhl(long double x);

float coshf(float x);                          see cosh
long double coshl(long double x);              see cosh

float sinhf(float x);                          see sinh
long double sinhl(long double x);              see sinh

float tanhf(float x);                          see tanh
long double tanhl(long double x);              see tanh
```

acosh
asinh
atanh

Six functions in this group correspond to the C89 functions `cosh`, `sinh`, and `tanh`. The new functions are `acosh`, which computes the arc hyperbolic cosine; `asinh`, which computes the arc hyperbolic sine; and `atanh`, which computes the arc hyperbolic tangent.

Exponential and Logarithmic Functions

```
float expf(float x);                                  see exp
long double expl(long double x);                      see exp

double exp2(double x);
float exp2f(float x);
long double exp2l(long double x);

double expm1(double x);
float expm1f(float x);
long double expm1l(long double x);

float frexpf(float value, int *exp);                  see frexp
long double frexpl(long double value,
                   int *exp);                         see frexp

int ilogb(double x);
int ilogbf(float x);
int ilogbl(long double x);

float ldexpf(float x, int exp);                       see ldexp
long double ldexpl(long double x, int exp);           see ldexp

float logf(float x);                                  see log
long double logl(long double x);                      see log

float log10f(float x);                                see log10
long double log10l(long double x);                    see log10

double log1p(double x);
float log1pf(float x);
long double log1pl(long double x);

double log2(double x);
float log2f(float x);
long double log2l(long double x);

double logb(double x);
float logbf(float x);
long double logbl(long double x);

float modff(float value, float *iptr);                see modf
long double modfl(long double value,
                  long double *iptr);                 see modf
```

```
double scalbn(double x, int n);
float scalbnf(float x, int n);
long double scalbnl(long double x, int n);
double scalbln(double x, long int n);
float scalblnf(float x, long int n);
long double scalblnl(long double x, long int n);
```

exp2 *expm1* In additional to new versions of `exp`, `frexp`, `ldexp`, `log`, `log10`, and `modf`, there are several entirely new functions in this category. Two of these, `exp2` and `expm1`, are variations on the `exp` function. When applied to the argument `x`, the `exp2` function returns 2^x, and `expm1` returns $e^x - 1$.

Q&A

logb *ilogb* *log1p* *log2* The `logb` function returns the exponent of its argument. More precisely, the call `logb(x)` returns $\log_r(|x|)$, where r is the radix of floating-point arithmetic (defined by the macro `FLT_RADIX`, which typically has the value 2). The `ilogb` function returns the value of `logb` after it has been cast to `int` type. The `log1p` function returns $\ln(1 + x)$ when given `x` as its argument. The `log2` function computes the base-2 logarithm of its argument.

scalbn *scalbln* The `scalbn` function returns $x \times \texttt{FLT_RADIX}^n$, which it computes in an efficient way (not by explicitly raising `FLT_RADIX` to the nth power). `scalbln` is the same as `scalbn`, except that its second parameter has type `long int` instead of `int`.

Power and Absolute Value Functions

```
double cbrt(double x);
float cbrtf(float x);
long double cbrtl(long double x);

float fabsf(float x);                          see fabs
long double fabsl(long double x);              see fabs

double hypot(double x, double y);
float hypotf(float x, float y);
long double hypotl(long double x, long double y);

float powf(float x, float y);                  see pow
long double powl(long double x,
                 long double y);               see pow

float sqrtf(float x);                          see sqrt
long double sqrtl(long double x);              see sqrt
```

Several functions in this group are new versions of old ones (`fabs`, `pow`, and `sqrt`). Only the functions `cbrt` and `hypot` (and their variants) are entirely new.

cbrt The `cbrt` function computes the cube root of its argument. The `pow` function can also be used for this purpose, but `pow` is unable to handle negative arguments

(a domain error occurs). `cbrt`, on the other hand, is defined for both positive and negative arguments. When its argument is negative, `cbrt` returns a negative result.

hypot When applied to arguments `x` and `y`, the `hypot` function returns $\sqrt{x^2 + y^2}$. In other words, this function computes the hypotenuse of a right triangle with legs `x` and `y`.

Error and Gamma Functions

```
double erf(double x);
float erff(float x);
long double erfl(long double x);

double erfc(double x);
float erfcf(float x);
long double erfcl(long double x);

double lgamma(double x);
float lgammaf(float x);
long double lgammal(long double x);

double tgamma(double x);
float tgammaf(float x);
long double tgammal(long double x);
```

erf *erfc* The `erf` function computes the ***error function*** erf (also known as the ***Gaussian error function***), which is used in probability, statistics and partial differential equations. The mathematical definition of erf is

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$$

`erfc` computes the ***complementary error function***, $\mathrm{erfc}(x) = 1 - \mathrm{erf}(x)$.

lgamma *tgamma* **Q&A** The ***gamma function*** Γ is an extension of the factorial function that can be applied to real numbers as well as to integers. When applied to an integer n, $\Gamma(n) = (n-1)!$; the definition of Γ for nonintegers is more complicated. The `tgamma` function computes Γ. The `lgamma` function computes $\ln(|\Gamma(x)|)$, the natural logarithm of the absolute value of the gamma function. `lgamma` can sometimes be more useful than the gamma function itself, because Γ grows so quickly that using it in calculations may cause overflow.

Nearest Integer Functions

```
float ceilf(float x);                          see ceil
long double ceill(long double x);              see ceil

float floorf(float x);                         see floor
long double floorl(long double x);             see floor
```

```
double nearbyint(double x);
float nearbyintf(float x);
long double nearbyintl(long double x);

double rint(double x);
float rintf(float x);
long double rintl(long double x);

long int lrint(double x);
long int lrintf(float x);
long int lrintl(long double x);
long long int llrint(double x);
long long int llrintf(float x);
long long int llrintl(long double x);

double round(double x);
float roundf(float x);
long double roundl(long double x);

long int lround(double x);
long int lroundf(float x);
long int lroundl(long double x);
long long int llround(double x);
long long int llroundf(float x);
long long int llroundl(long double x);

double trunc(double x);
float truncf(float x);
long double truncl(long double x);
```

Besides additional versions of `ceil` and `floor`, C99 has a number of new functions that convert a floating-point value to the nearest integer. Be careful when using these functions: although all of them return an integer, some functions return it in floating-point format (as a `float`, `double`, or `long double` value) and some return it in integer format (as a `long int` or `long long int` value).

nearbyint
rint

The `nearbyint` function rounds its argument to an integer, returning it as a floating-point number. `nearbyint` uses the current rounding direction and does not raise the *inexact* floating-point exception. `rint` is the same as `nearbyint`, except that it may raise the *inexact* floating-point exception if the result has a different value than the argument.

lrint
llrint

The `lrint` function rounds its argument to the nearest integer, according to the current rounding direction. `lrint` returns a `long int` value. `llrint` is the same as `lrint`, except that it returns a `long long int` value.

round

The `round` function rounds its argument to the nearest integer value, returning it as a floating-point number. `round` always rounds away from zero (so 3.5 is rounded to 4.0, for example).

lround *llround* The `lround` function rounds its argument to the nearest integer value, returning it as a `long int` value. Like `round`, it rounds away from zero. `llround` is the same as `lround`, except that it returns a `long long int` value.

trunc The `trunc` function rounds its argument to the nearest integer not larger in magnitude. (In other words, it truncates the argument toward zero.) `trunc` returns the result as a floating-point number.

Remainder Functions

```
float fmodf(float x, float y);                          see fmod
long double fmodl(long double x,
                  long double y);                       see fmod

double remainder(double x, double y);
float remainderf(float x, float y);
long double remainderl(long double x,
                       long double y);

double remquo(double x, double y, int *quo);
float remquof(float x, float y, int *quo);
long double remquol(long double x, long double y,
                    int *quo);
```

Besides additional versions of `fmod`, this category includes new remainder functions named `remainder` and `remquo`.

remainder The `remainder` function returns `x` REM `y`, where REM is a function defined in the IEEE standard. For `y` $\neq 0$, the value of `x` REM `y` is $r = x - ny$, where n is the integer nearest the exact value of `x/y`. (If `x/y` is halfway between two integers, n is even.) If $r = 0$, it has the same sign as `x`.

remquo The `remquo` function returns the same value as `remainder` when given the same first two arguments. In addition, `remquo` modifies the object pointed to by the `quo` parameter so that it contains n low-order bits of the integer quotient |`x/y`|, where n depends on the implementation but must be at least three. The value stored in this object will be negative if `x/y` < 0.

Manipulation Functions

```
double copysign(double x, double y);
float copysignf(float x, float y);
long double copysignl(long double x, long double y);

double nan(const char *tagp);
float nanf(const char *tagp);
long double nanl(const char *tagp);

double nextafter(double x, double y);
float nextafterf(float x, float y);
```

```
long double nextafterl(long double x, long double y);

double nexttoward(double x, long double y);
float nexttowardf(float x, long double y);
long double nexttowardl(long double x,
                        long double y);
```

The mysteriously named "manipulation functions" are all new in C99. They provide access to the low-level details of floating-point numbers.

copysign The `copysign` function copies the sign of one number to another number. The call `copysign(x, y)` returns a value with the magnitude of `x` and the sign of `y`.

nan The `nan` function converts a string to a NaN value. The call `nan("`*n-char-sequence*`")` is equivalent to `strtod("NAN(`*n-char-sequence*`)", (char**) NULL)`. (See the discussion of `strtod` for a description of the format of *n-char-sequence*.) The call `nan("")` is equivalent to `strtod("NAN()", (char**) NULL)`. If the argument in a call of `nan` doesn't have the value `"`*n-char-sequence*`"` or `""`, the call is equivalent to `strtod("NAN", (char**) NULL)`. If quiet NaNs aren't supported, `nan` returns zero. Calls of `nanf` and `nanl` are equivalent to calls of `strtof` and `strtold`, respectively. This function is used to construct a NaN value containing a specific binary pattern. (Recall from earlier in this section that the fraction part of a NaN value is arbitrary.)

`strtod` function ➤26.2

nextafter The `nextafter` function determines the next representable value of a number `x` (if all values of `x`'s type were listed in order, the number that would come just before or just after `x`). The value of `y` determines the direction: if `y < x`, then the function returns the value just before `x`; if `x < y`, it returns the value just after `x`. If `x` and `y` are equal, `nextafter` returns `y`.

Q&A

nexttoward The `nexttoward` function is the same as `nextafter`, except that the `y` parameter has type `long double` instead of `double`. If `x` and `y` are equal, `nexttoward` returns `y` converted to the function's return type. The advantage of `nexttoward` is that a value of any (real) floating type can be passed as the second argument without the danger of it being incorrectly converted to a narrower type.

Maximum, Minimum, and Positive Difference Functions

```
double fdim(double x, double y);
float fdimf(float x, float y);
long double fdiml(long double x, long double y);

double fmax(double x, double y);
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);

double fmin(double x, double y);
float fminf(float x, float y);
long double fminl(long double x, long double y);
```

fdim The `fdim` function computes the positive difference of `x` and `y`:

$$\begin{cases} x - y & \text{if } x > y \\ +0 & \text{if } x \le y \end{cases}$$

fmax *fmin* The `fmax` function returns the larger of its two arguments. `fmin` returns the value of the smaller argument.

Floating Multiply-Add

```
double fma(double x, double y, double z);
float fmaf(float x, float y, float z);
long double fmal(long double x, long double y,
                 long double z);
```

fma The `fma` function multiplies its first two arguments, then adds the third argument. In other words, we could replace the statement

```
a = b * c + d;
```

with

```
a = fma(b, c, d);
```

This function was added to C99 because some newer CPUs have a "fused multiply-add" instruction that both multiplies and adds. Calling `fma` tells the compiler to use this instruction (if available), which can be faster than performing separate multiply and add instructions. Moreover, the fused multiply-add instruction performs only one rounding operation, not two, so it may produce a more accurate result. It's particularly useful for algorithms that perform a series of multiplications and additions, such as the algorithms for finding the dot product of two vectors or multiplying two matrices.

To determine whether calling the `fma` function is a good idea, a C99 program can test whether the `FP_FAST_FMA` macro is defined. If it is, then calling `fma` should be faster than—or at least as fast as—performing separate multiply and add operations. The `FP_FAST_FMAF` and `FP_FAST_FMAL` macros play the same role for the `fmaf` and `fmal` functions, respectively.

Performing a combined multiply and add is an example of what the C99 standard calls "contraction," where two or more mathematical operations are combined and performed as a single operation. As we saw with the `fma` function, contraction often leads to better speed and greater accuracy. However, programmers may wish to control whether contraction is done automatically (as opposed to calls of `fma`, which are explicit requests for contraction), since contraction can lead to slightly different results. In extreme cases, contraction can avoid a float-point exception that would otherwise be raised.

#pragma directive ➤ 14.5

C99 provides a pragma named FP_CONTRACT that gives the programmer control over contraction. Here's how the pragma is used:

```
#pragma STDC FP_CONTRACT on-off-switch
```

The value of *on-off-switch* is either ON, OFF, or DEFAULT. If ON is selected, the compiler is allowed to contract expressions; if OFF is selected, the compiler is prohibited from contracting expressions. DEFAULT is useful for restoring the default setting (which may be either ON or OFF). If the pragma is used at the outer level of a program (outside any function definitions), it remains in effect until a subsequent FP_CONTRACT pragma appears in the same file, or until the file ends. If the pragma is used inside a compound statement (including the body of a function), it must appear first, before any declarations or statements; it remains in effect until the end of the statement, unless overridden by another pragma. A program may still call fma to perform an explicit contraction even when FP_CONTRACT has been used to prohibit automatic contraction of expressions.

Comparison Macros

```
int isgreater(real-floating x, real-floating y);
int isgreaterequal(real-floating x, real-floating y);
int isless(real-floating x, real-floating y);
int islessequal(real-floating x, real-floating y);
int islessgreater(real-floating x, real-floating y);
int isunordered(real-floating x, real-floating y);
```

Our final category consists of function-like macros that compare two numbers. These macros are designed to accept arguments of any real floating type.

The comparison macros exist because of a problem that can arise when floating-point numbers are compared using the ordinary relational operators such as < and >. If either operand (or both) is a NaN, such a comparison may cause the *invalid* floating-point exception to be raised, because NaN values—unlike other floating-point values—are considered to be unordered. The comparison macros can be used to avoid this exception. These macros are said to be "quiet" versions of the relational operators because they do their job without raising an exception.

isgreater
isgreaterequal
isless
islessequal

The isgreater, isgreaterequal, isless, and islessequal macros perform the same operation as the >, >=, <, and <= operators, respectively, except that they don't cause the *invalid* floating-point exception to be raised when the arguments are unordered.

islessgreater

The call islessgreater(x, y) is equivalent to (x) < (y) || (x) > (y), except that it guarantees not to evaluate x and y twice, and—like the previous macros—doesn't cause the *invalid* floating-point exception to be raised when x and y are unordered.

isunordered

The isunordered macro returns 1 if its arguments are unordered (at least one of them is a NaN) and 0 otherwise.

23.5 The <ctype.h> Header: Character Handling

The <ctype.h> header provides two kinds of functions: character-classification functions (like isdigit, which tests whether a character is a digit) and character case-mapping functions (like toupper, which converts a lower-case letter to upper case).

Although C doesn't require that we use the functions in <ctype.h> to test characters and perform case conversions, it's a good idea to do so. First, these functions have been optimized for speed (in fact, many are implemented as macros). Second, we'll end up with a more portable program, since these functions work with any character set. Third, the <ctype.h> functions adjust their behavior when the locale is changed, which helps us write programs that run properly in different parts of the world.

locales ➤25.1

The functions in <ctype.h> all take int arguments and return int values. In many cases, the argument is already stored in an int variable (often as a result of having been read by a call of fgetc, getc, or getchar). If the argument has char type, however, we need to be careful. C can automatically convert a char argument to int type; if char is an unsigned type or if we're using a seven-bit character set such as ASCII, the conversion will go smoothly. But if char is a signed type and if some characters require eight bits, then converting such a character from char to int will give a negative result. The behavior of the <ctype.h> functions is undefined for negative arguments (other than EOF), potentially causing serious problems. In such a situation, the argument should be cast to unsigned char for safety. (For maximum portability, some programmers always cast a char value to unsigned char when passing it to a <ctype.h> function.)

Character-Classification Functions

```
int isalnum(int c);
int isalpha(int c);
int isblank(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
```

Each character-classification function returns a nonzero value if its argument has a particular property. Table 23.10 lists the property that each function tests.

Table 23.10 Character-Classification Functions

Function	*Test*
`isalnum(c)`	Is `c` alphanumeric?
`isalpha(c)`	Is `c` alphabetic?
`isblank(c)`	Is `c` a blank?†
`iscntrl(c)`	Is `c` a control character?††
`isdigit(c)`	Is `c` a decimal digit?
`isgraph(c)`	Is `c` a printing character (other than a space)?
`islower(c)`	Is `c` a lower-case letter?
`isprint(c)`	Is `c` a printing character (including a space)?
`ispunct(c)`	Is `c` punctuation?†††
`isspace(c)`	Is `c` a white-space character?††††
`isupper(c)`	Is `c` an upper-case letter?
`isxdigit(c)`	Is `c` a hexadecimal digit?

†The standard blank characters are space and horizontal tab (`\t`). This function is new in C99.
††In ASCII, the control characters are `\x00` through `\x1f` plus `\x7f`.
†††All printing characters except those for which `isspace` or `isalnum` are true are considered punctuation.
††††The white-space characters are space, form feed (`\f`), new-line (`\n`), carriage return (`\r`), horizontal tab (`\t`), and vertical tab (`\v`).

C99 The C99 definition of `ispunct` is slightly different than the one in C89. In C89, `ispunct(c)` tests whether `c` is a printing character but not a space or a character for which `isalnum(c)` is true. In C99, `ispunct(c)` tests whether `c` is a printing character for which neither `isspace(c)` nor `isalnum(c)` is true.

PROGRAM

Testing the Character-Classification Functions

The following program demonstrates the character-classification functions (with the exception of `isblank`, which is new in C99) by applying them to the characters in the string `"azAZ0 !\t"`.

tclassify.c

```
/* Tests the character-classification functions */

#include <ctype.h>
#include <stdio.h>

#define TEST(f) printf("  %c  ", f(*p) ? 'x' : ' ')

int main(void)
{
  char *p;

  printf("     alnum    cntrl    graph    print"
         "     space    xdigit\n"
         "          alpha    digit    lower    punct"
         "     upper\n");
```

```
  for (p = "azAZ0 !\t"; *p != '\0'; p++) {
    if (iscntrl(*p))
      printf("\\x%02x:", *p);
    else
      printf("   %c:", *p);
    TEST(isalnum);
    TEST(isalpha);
    TEST(iscntrl);
    TEST(isdigit);
    TEST(isgraph);
    TEST(islower);
    TEST(isprint);
    TEST(ispunct);
    TEST(isspace);
    TEST(isupper);
    TEST(isxdigit);
    printf("\n");
  }

  return 0;
}
```

The program produces the following output:

```
      alnum   cntrl   graph   print   space   xdigit
      alpha   digit   lower   punct   upper
  a:   x      x       x       x       x       x
  z:   x      x       x       x       x
  A:   x      x       x       x       x       x
  Z:   x      x       x       x       x
  0:   x              x       x       x       x
   :                          x       x
  !:                  x       x       x
\x09:       x                       x
```

Character Case-Mapping Functions

```
int tolower(int c);
int toupper(int c);
```

tolower
toupper

The `tolower` function returns the lower-case version of a letter passed to it as an argument, while `toupper` returns the upper-case version. If the argument to either function is not a letter, it returns the character unchanged.

PROGRAM

Testing the Case-Mapping Functions

The following program applies the case-mapping functions to the characters in the string `"aA0!"`.

tcasemap.c

```
/* Tests the case-mapping functions */

#include <ctype.h>
#include <stdio.h>

int main(void)
{
  char *p;

  for (p = "aA0!"; *p != '\0'; p++) {
    printf("tolower('%c') is '%c'; ", *p, tolower(*p));
    printf("toupper('%c') is '%c'\n", *p, toupper(*p));
  }
  return 0;
}
```

The program produces the following output:

```
tolower('a') is 'a'; toupper('a') is 'A'
tolower('A') is 'a'; toupper('A') is 'A'
tolower('0') is '0'; toupper('0') is '0'
tolower('!') is '!'; toupper('!') is '!'
```

23.6 The <string.h> Header: String Handling

We first encountered the <string.h> header in Section 13.5, which covered the most basic string operations: copying strings, concatenating strings, comparing strings, and finding the length of a string. As we'll see now, there are quite a few string-handling functions in <string.h>, as well as functions that operate on character arrays that aren't necessarily null-terminated. Functions in the latter category have names that begin with mem, to suggest that these functions deal with blocks of memory rather than strings. These memory blocks may contain data of any type, hence the arguments to the mem functions have type void * rather than char *.

<string.h> provides five kinds of functions:

- ***Copying functions.*** Functions that copy characters from one place in memory to another place.
- ***Concatenation functions.*** Functions that add characters to the end of a string.
- ***Comparison functions.*** Functions that compare character arrays.
- ***Search functions.*** Functions that search an array for a particular character, a set of characters, or a string.
- ***Miscellaneous functions.*** Functions that initialize a memory block or compute the length of a string.

We'll now discuss these functions, one group at a time.

Copying Functions

```
void *memcpy(void * restrict s1,
             const void * restrict s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
char *strcpy(char * restrict s1,
             const char * restrict s2);
char *strncpy(char * restrict s1,
              const char * restrict s2, size_t n);
```

Q&A The functions in this category copy characters (bytes) from one place in memory (the "source") to another (the "destination"). Each function requires that the first argument point to the destination and the second point to the source. All copying functions return the first argument (a pointer to the destination).

memcpy
memmove

`memcpy` copies `n` characters from the source to the destination, where `n` is the function's third argument. If the source and destination overlap, the behavior of `memcpy` is undefined. `memmove` is the same as `memcpy`, except that it works correctly when the source and destination overlap.

strcpy
strncpy

`strcpy` copies a null-terminated string from the source to the destination. `strncpy` is similar to `strcpy`, but it won't copy more than `n` characters, where `n` is the function's third argument. (If `n` is too small, `strncpy` won't be able to copy a terminating null character.) If it encounters a null character in the source, `strncpy` adds null characters to the destination until it has written a total of `n` characters. `strcpy` and `strncpy`, like `memcpy`, aren't guaranteed to work if the source and destination overlap.

The following examples illustrate the copying functions; the comments show which characters are copied.

```
char source[] = {'h', 'o', 't', '\0', 't', 'e', 'a'};
char dest[7];

memcpy(dest, source, 3);    /* h, o, t                  */
memcpy(dest, source, 4);    /* h, o, t, \0              */
memcpy(dest, source, 7);    /* h, o, t, \0, t, e, a     */

memmove(dest, source, 3);   /* h, o, t                  */
memmove(dest, source, 4);   /* h, o, t, \0              */
memmove(dest, source, 7);   /* h, o, t, \0, t, e, a     */

strcpy(dest, source);       /* h, o, t, \0              */

strncpy(dest, source, 3);   /* h, o, t                  */
strncpy(dest, source, 4);   /* h, o, t, \0              */
strncpy(dest, source, 7);   /* h, o, t, \0, \0, \0, \0 */
```

Note that `memcpy`, `memmove`, and `strncpy` don't require a null-terminated string; they work just as well with any block of memory. The `strcpy` function, on the other hand, doesn't stop copying until it reaches a null character, so it works only with null-terminated strings.

Section 13.5 gives examples of how `strcpy` and `strncpy` are typically used. Although neither function is completely safe, `strncpy` at least provides a way to limit the number of characters it will copy.

Concatenation Functions

```
char *strcat(char * restrict s1,
             const char * restrict s2);
char *strncat(char * restrict s1,
              const char * restrict s2, size_t n);
```

strcat

`strcat` appends its second argument to the end of the first argument. Both arguments must be null-terminated strings; `strcat` puts a null character at the end of the concatenated string. Consider the following example:

```
char str[7] = "tea";

strcat(str, "bag");   /* adds b, a, g, \0 to end of str */
```

The letter `b` overwrites the null character after the `a` in `"tea"`, so that `str` now contains the string `"teabag"`. `strcat` returns its first argument (a pointer).

strncat

`strncat` is the same as `strcat`, except that its third argument limits the number of characters it will copy:

```
char str[7] = "tea";

strncat(str, "bag", 2);   /* adds b, a, \0 to str    */
strncat(str, "bag", 3);   /* adds b, a, g, \0 to str */
strncat(str, "bag", 4);   /* adds b, a, g, \0 to str */
```

As these examples show, `strncat` always leaves the resulting string properly null-terminated.

In Section 13.5, we saw that a call of `strncat` often has the following appearance:

```
strncat(str1, str2, sizeof(str1) - strlen(str1) - 1);
```

The third argument calculates the amount of space remaining in `str1` (given by the expression `sizeof(str1) - strlen(str1)`) and then subtracts 1 to ensure that there will be room for the null character.

Comparison Functions

```
int memcmp(const void *s1, const void *s2, size_t n);
int strcmp(const char *s1, const char *s2);
int strcoll(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2,
            size_t n);
size_t strxfrm(char * restrict s1,
               const char * restrict s2, size_t n);
```

The comparison functions fall into two groups. Functions in the first group (memcmp, strcmp, and strncmp) compare the contents of two character arrays. Functions in the second group (strcoll and strxfrm) are used if the locale needs to be taken into account.

locales ➤25.1

memcmp
strcmp
strncmp

The memcmp, strcmp, and strncmp functions have much in common. All three expect to be passed pointers to character arrays. The characters in the first array are then compared one by one with the characters in the second array. All three functions return as soon as a mismatch is found. Also, all three return a negative, zero, or positive integer, depending on whether the stopping character in the first array was less than, equal to, or greater than the stopping character in the second.

The differences among the three functions have to do with when to stop comparing characters if no mismatch is found. The memcmp function is passed a third argument, n, that limits the number of comparisons performed; it pays no particular attention to null characters. strcmp doesn't have a preset limit, stopping instead when it reaches a null character in either array. (As a result, strcmp works only with null-terminated strings.) strncmp is a blend of memcmp and strcmp; it stops when n comparisons have been performed or a null character is reached in either array.

The following examples illustrate memcmp, strcmp, and strncmp:

```
char s1[] = {'b', 'i', 'g', '\0', 'c', 'a', 'r'};
char s2[] = {'b', 'i', 'g', '\0', 'c', 'a', 't'};

if (memcmp(s1, s2, 3) == 0) ...    /* true  */
if (memcmp(s1, s2, 4) == 0) ...    /* true  */
if (memcmp(s1, s2, 7) == 0) ...    /* false */

if (strcmp(s1, s2) == 0) ...       /* true  */

if (strncmp(s1, s2, 3) == 0) ...   /* true  */
if (strncmp(s1, s2, 4) == 0) ...   /* true  */
if (strncmp(s1, s2, 7) == 0) ...   /* true  */
```

strcoll

The strcoll function is similar to strcmp, but the outcome of the comparison depends on the current locale.

strxfrm

Most of the time, strcoll is fine for performing a locale-dependent string comparison. Occasionally, however, we might need to perform the comparison more than once (a potential problem, since strcoll isn't especially fast) or change the locale without affecting the outcome of the comparison. In these situations, the strxfrm ("string transform") function is available as an alternative to strcoll.

strxfrm transforms its second argument (a string), placing the result in the array pointed to by the first argument. The third argument limits the number of characters written to the array, including the terminating null character. Calling strcmp with two transformed strings should produce the same outcome (negative, zero, or positive) as calling strcoll with the original strings.

strxfrm returns the length of the transformed string. As a result, it's typically called twice: once to determine the length of the transformed string and once to perform the transformation. Here's an example:

```
size_t len;
char *transformed;

len = strxfrm(NULL, original, 0);
transformed = malloc(len + 1);
strxfrm(transformed, original, len);
```

Search Functions

```
void *memchr(const void *s, int c, size_t n);
char *strchr(const char *s, int c);
size_t strcspn(const char *s1, const char *s2);
char *strpbrk(const char *s1, const char *s2);
char *strrchr(const char *s, int c);
size_t strspn(const char *s1, const char *s2);
char *strstr(const char *s1, const char *s2);
char *strtok(char * restrict s1,
             const char * restrict s2);
```

strchr The strchr function searches a string for a particular character. The following example shows how we might use strchr to search a string for the letter f.

```
char *p, str[] = "Form follows function.";

p = strchr(str, 'f');   /* finds first 'f' */
```

strchr returns a pointer to the first occurrence of f in str (the one in the word follows). Locating multiple occurrences of a character is easy; for example, the call

```
p = strchr(p + 1, 'f');   /* finds next 'f' */
```

finds the second f in str (the one in function). If it can't locate the desired character, strchr returns a null pointer.

memchr memchr is similar to strchr, but it stops searching after a set number of characters instead of stopping at the first null character. memchr's third argument limits the number of characters it can examine—a useful capability if we don't want to search an entire string or if we're searching a block of memory that's not terminated by a null character. The following example uses memchr to search an array of characters that lacks a null character at the end:

```
char *p, str[22] = "Form follows function.";

p = memchr(str, 'f', sizeof(str));
```

Like the `strchr` function, `memchr` returns a pointer to the first occurrence of the character. If it can't find the desired character, `memchr` returns a null pointer.

strrchr

`strrchr` is similar to `strchr`, but it searches the string in *reverse* order:

```
char *p, str[] = "Form follows function.";

p = strrchr(str, 'f');    /* finds last 'f' */
```

In this example, `strrchr` will first search for the null character at the end of the string, then go backwards to locate the letter `f` (the one in `function`). Like `strchr` and `memchr`, `strrchr` returns a null pointer if it fails to find the desired character.

strpbrk

`strpbrk` is more general than `strchr`; it returns a pointer to the leftmost character in the first argument that matches *any* character in the second argument:

```
char *p, str[] = "Form follows function.";

p = strpbrk(str, "mn");    /* finds first 'm' or 'n' */
```

In this example, `p` will point to the letter `m` in `Form`. `strpbrk` returns a null pointer if no match is found.

strspn
strcspn
Q&A

The `strspn` and `strcspn` functions, unlike the other search functions, return an integer (of type `size_t`), representing a position within a string. When given a string to search and a set of characters to look for, `strspn` returns the index of the first character that's *not* in the set. When passed similar arguments, `strcspn` returns the index of the first character that's *in* the set. Here are examples of both functions:

```
size_t n;
char str[] = "Form follows function.";

n = strspn(str, "morF");      /* n = 4 */
n = strspn(str, " \t\n");     /* n = 0 */
n = strcspn(str, "morF");     /* n = 0 */
n = strcspn(str, " \t\n");    /* n = 4 */
```

strstr

`strstr` searches its first argument (a string) for a match with its second argument (also a string). In the following example, `strstr` searches for the word `fun`:

```
char *p, str[] = "Form follows function.";

p = strstr(str, "fun");    /* locates "fun" in str */
```

`strstr` returns a pointer to the first occurrence of the search string; it returns a null pointer if it can't locate the string. After the call above, `p` will point to the letter `f` in `function`.

strtok

`strtok` is the most complicated of the search functions. It's designed to search a string for a "token"—a sequence of characters that doesn't include certain delimiting characters. The call `strtok(s1, s2)` scans the `s1` string for a nonempty sequence of characters that are *not* in the `s2` string. `strtok` marks the end

of the token by storing a null character in `s1` just after the last character in the token; it then returns a pointer to the first character in the token.

What makes `strtok` especially useful is that later calls can find additional tokens in the same string. The call `strtok(NULL, s2)` continues the search begun by the previous `strtok` call. As before, `strtok` marks the end of the token with a null character, then returns a pointer to the beginning of the token. The process can be repeated until `strtok` returns a null pointer, indicating that no token was found.

To see how `strtok` works, we'll use it to extract a month, day, and year from a date written in the form

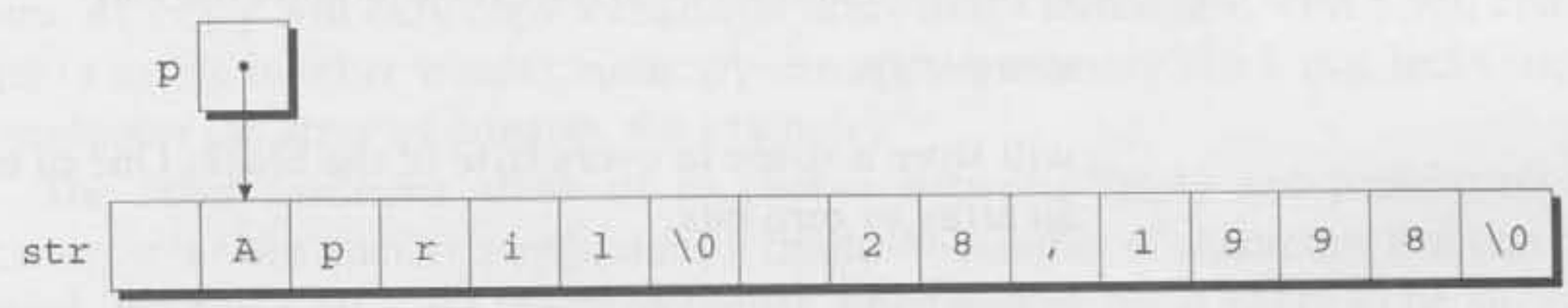
month day, year

where spaces and/or tabs separate the month from the day and the day from the year. In addition, spaces and tabs may precede the comma. Let's say that the string `str` has the following appearance to start with:

After the call

```
p = strtok(str, " \t");
```

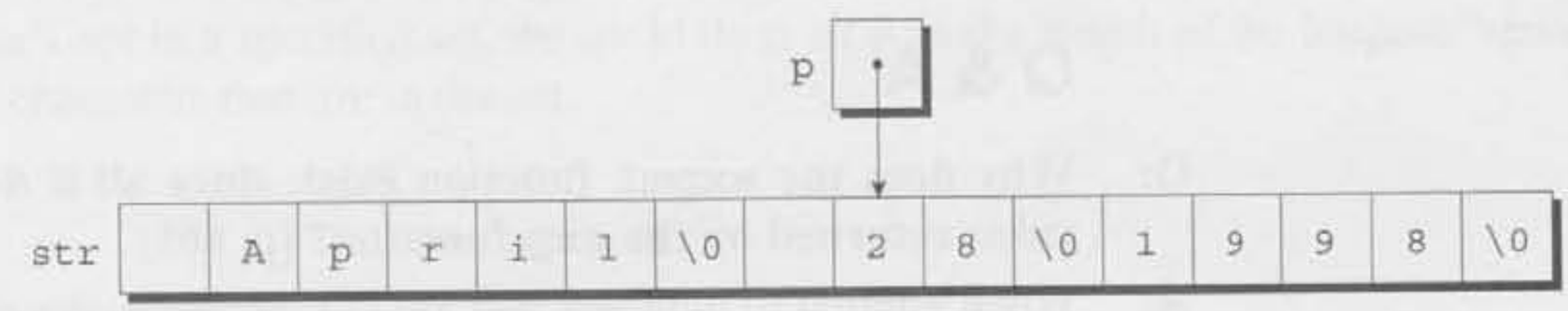
`str` will have the following appearance:



`p` points to the first character in the month string, which is now terminated by a null character. Calling `strtok` with a null pointer as its first argument causes it to resume the search from where it left off:

```
p = strtok(NULL, " \t,");
```

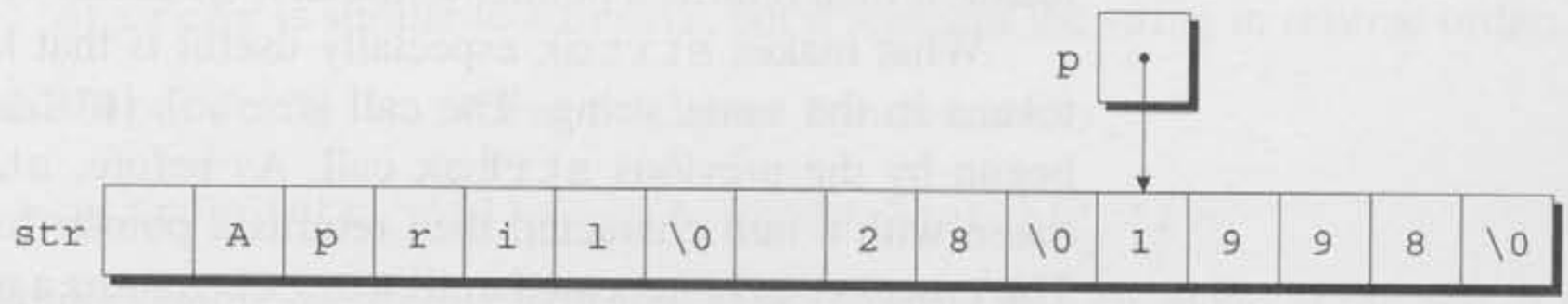
After this call, `p` points to the first character in the day:



A final call of `strtok` locates the year:

```
p = strtok(NULL, " \t");
```

After this call, `str` will have the following appearance:



When `strtok` is called repeatedly to break a string into tokens, the second argument isn't required to be the same in each call. In our example, the second call of `strtok` has the argument `" \t,"` instead of `" \t"`.

`strtok` has several well-known problems that limit its usefulness; I'll mention just a couple. First, it works with only one string at a time; it can't conduct simultaneous searches through two different strings. Also, `strtok` treats a sequence of delimiters in the same way as a single delimiter, making it unsuitable for applications in which a string contains a series of fields separated by a delimiter (such as a comma) and some of the fields are empty.

Miscellaneous Functions

```
void *memset(void *s, int c, size_t n);
size_t strlen(const char *s);
```

memset

`memset` stores multiple copies of a character in a specified area of memory. If `p` points to a block of `N` bytes, for example, the call

```
memset(p, ' ', N);
```

will store a space in every byte of the block. One of `memset`'s uses is initializing an array to zero bits:

```
memset(a, 0, sizeof(a));
```

`memset` returns its first argument (a pointer).

strlen

`strlen` returns the length of a string, not counting the null character. See Section 13.5 for examples of `strlen` calls.

`strerror` function ➤24.2

There's one other miscellaneous string function, `strerror`, which is covered along with the `<errno.h>` header.

Q & A

Q: Why does the `expm1` function exist, since all it does is subtract 1 from the value returned by the `exp` function? [p. 605]

A: When applied to numbers that are close to zero, the `exp` function returns a value that's very close to 1. The result of subtracting 1 from the value returned by `exp` may not be accurate because of round-off error. `expm1` is designed to give a more accurate result in this situation.

The `log1p` function exists for a similar reason. For values of `x` that are close to zero, `log1p(x)` should be more accurate than `log(1 + x)`.

Q: Why is the function that computes the gamma function named `tgamma` instead of just `gamma`? [p. 606]

A: At the time the C99 standard was being written, some compilers provided a function named `gamma`, but it computed the log of the gamma function. This function was later renamed `lgamma`. Choosing the name `gamma` for the gamma function would have conflicted with existing practice, so the C99 committee decided on the name `tgamma` ("true gamma") instead.

Q: Why does the description of the `nextafter` function say that if `x` and `y` are equal, `nextafter` returns `y`? If `x` and `y` are equal, what's the difference between returning `x` or `y`? [p. 609]

A: Consider the call `nextafter(-0.0, +0.0)`, in which the arguments are mathematically equal. By returning `y` instead of `x`, the function has a return value of +0.0 (rather than –0.0, which would be counterintuitive). Similarly, the call `nextafter(+0.0, -0.0)` returns –0.0.

Q: Why does `<string.h>` provide so many ways to do the same thing? Do we really need four copying functions (`memcpy`, `memmove`, `strcpy`, and `strncpy`)? [p. 616]

A: Let's start with `memcpy` and `strcpy`. These functions are used for different purposes. `strcpy` will only copy a character array that's terminated with a null character (a string, in other words); `memcpy` can copy a memory block that lacks such a terminator (an array of integers, for example).

The other functions allow us to choose between safety and performance. `strncpy` is safer than `strcpy`, since it limits the number of characters that can be copied. We pay a price for safety, however, since `strncpy` is likely to be slower than `strcpy`. Using `memmove` involves a similar trade-off. `memmove` will copy bytes from one region of memory into a possibly overlapping region. `memcpy` isn't guaranteed to work properly in this situation; however, if we can guarantee no overlap, `memcpy` is likely to be faster than `memmove`.

Q: Why does the `strspn` function have such an odd name? [p. 620]

A: Instead of thinking of `strspn`'s return value as the index of the first character that's *not* in a specified set, we could think of it as the length of the longest "span" of characters that *are* in the set.

Exercises

Section 23.3

W 1. Extend the `round_nearest` function so that it rounds a floating-point number x to n digits after the decimal point. For example, the call `round_nearest(3.14159, 3)` would

return 3.142. *Hint:* Multiply x by 10^n, round to the nearest integer, then divide by 10^n. Be sure that your function works correctly for both positive and negative values of x.

Section 23.4

2. (C99) Write the following function:

```
double evaluate_polynomial(double a[], int n, double x);
```

The function should return the value of the polynomial $a_n x^n + a_{n-1}x^{n-1} + \ldots + a_0$, where the a_i's are stored in corresponding elements of the array `a`, which has length `n + 1`. Have the function use Horner's Rule to compute the value of the polynomial:

$$((\ldots((a_n x + a_{n-1})x + a_{n-2})x + \ldots)x + a_1)x + a_0$$

Use the `fma` function to perform the multiplications and additions.

3. (C99) Check the documentation for your compiler to see if it performs contraction on arithmetic expressions and, if so, under what circumstances.

Section 23.5

4. Using `isalpha` and `isalnum`, write a function that checks whether a string has the syntax of a C identifier (it consists of letters, digits, and underscores, with a letter or underscore at the beginning).

5. Using `isxdigit`, write a function that checks whether a string represents a valid hexadecimal number (it consists solely of hexadecimal digits). If so, the function returns the value of the number as a `long int`. Otherwise, the function returns –1.

Section 23.6

W 6. In each of the following cases, indicate which function would be the best to use: `memcpy`, `memmove`, `strcpy`, or `strncpy`. Assume that the indicated action is to be performed by a single function call.

(a) Moving all elements of an array "down" one position in order to leave room for a new element in position 0.

(b) Deleting the first character in a null-terminated string by moving all other characters back one position.

(c) Copying a string into a character array that may not be large enough to hold it. If the array is too small, assume that the string is to be truncated; no null character is necessary at the end.

(d) Copying the contents of one array variable into another.

7. Section 23.6 explains how to call `strchr` repeatedly to locate all occurrences of a character within a string. Is it possible to locate all occurrences *in reverse order* by calling `strrchr` repeatedly?

W 8. Use `strchr` to write the following function:

```
int numchar(const char *s, char ch);
```

`numchar` returns the number of times the character `ch` occurs in the string `s`.

9. Replace the test condition in the following `if` statement by a single call of `strchr`:

```
if (ch == 'a' || ch == 'b' || ch == 'c') ...
```

W 10. Replace the test condition in the following `if` statement by a single call of `strstr`:

```
if (strcmp(str, "foo") == 0 || strcmp(str, "bar") == 0 ||
    strcmp(str, "baz") == 0) ...
```

Hint: Combine the string literals into a single string, separating them with a special character. Does your solution assume anything about the contents of `str`?

W 11. Write a call of `memset` that replaces the last `n` characters in a null-terminated string `s` with `!` characters.

12. Many versions of `<string.h>` provide additional (nonstandard) functions, such as those listed below. Write each function using only the features of the C standard.

(a) `strdup(s)` — Returns a pointer to a copy of `s` stored in memory obtained by calling `malloc`. Returns a null pointer if enough memory couldn't be allocated.
(b) `stricmp(s1, s2)` — Similar to `strcmp`, but ignores the case of letters.
(c) `strlwr(s)` — Converts upper-case letters in `s` to lower case, leaving other characters unchanged; returns `s`.
(d) `strrev(s)` — Reverses the characters in `s` (except the null character); returns `s`.
(e) `strset(s, ch)` — Fills `s` with copies of the character `ch`; returns `s`.

If you test any of these functions, you may need to alter its name. Functions whose names begin with `str` are reserved by the C standard.

13. Use `strtok` to write the following function:

```
int count_words(char *sentence);
```

`count_words` returns the number of words in the string `sentence`, where a "word" is any sequence of non-white-space characters. `count_words` is allowed to modify the string.

Programming Projects

1. Write a program that finds the roots of the equation $ax^2 + bx + c = 0$ using the formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Have the program prompt for the values of a, b, and c, then print both values of x. (If $b^2 - 4ac$ is negative, the program should instead print a message to the effect that the roots are complex.)

W 2. Write a program that copies a text file from standard input to standard output, removing all white-space characters from the beginning of each line. A line consisting entirely of white-space characters will not be copied.

3. Write a program that copies a text file from standard input to standard output, capitalizing the first letter in each word.

4. Write a program that prompts the user to enter a series of words separated by single spaces, then prints the words in reverse order. Read the input as a string, and then use `strtok` to break it into words.

5. Suppose that money is deposited into a savings account and left for t years. Assume that the annual interest rate is r and that interest is compounded continuously. The formula $A(t) = Pe^{rt}$ can be used to calculate the final value of the account, where P is the original amount deposited. For example, \$1000 left on deposit for 10 years at 6% interest would be worth $\$1000 \times e^{.06\times10} = \$1000 \times e^{.6} = \$1000 \times 1.8221188 = \$1{,}822.12$. Write a program that displays the result of this calculation after prompting the user to enter the original amount deposited, the interest rate, and the number of years.

6. Write a program that copies a text file from standard input to standard output, replacing each control character (other than `\n`) by a question mark.

7. Write a program that counts the number of sentences in a text file (obtained from standard input). Assume that each sentence ends with a `.`, `?`, or `!` followed by a white-space character (including `\n`).

24 Error Handling

There are two ways to write error-free programs; only the third one works.

Although student programs often fail when subjected to unexpected input, commercial programs need to be "bulletproof"—able to recover gracefully from errors instead of crashing. Making programs bulletproof requires that we anticipate errors that might arise during the execution of the program, include a check for each one, and provide a suitable action for the program to perform if the error should occur.

This chapter describes two ways for programs to check for errors: by using the `assert` macro and by testing the `errno` variable. Section 24.1 covers the `<assert.h>` header, where `assert` is defined. Section 24.2 discusses the `<errno.h>` header, to which the `errno` variable belongs. This section also includes coverage of the `perror` and `strerror` functions. These functions, which come from `<stdio.h>` and `<string.h>`, respectively, are closely related to the `errno` variable.

Section 24.3 explains how programs can detect and handle conditions known as signals, some of which represent errors. The functions that deal with signals are declared in the `<signal.h>` header.

Finally, Section 24.4 explores the `setjmp/longjmp` mechanism, which is often used for responding to errors. Both `setjmp` and `longjmp` belong to the `<setjmp.h>` header.

Error detection and handling aren't among C's strengths. C indicates run-time errors in a variety of ways rather than in a single, uniform way. Furthermore, it's the programmer's responsibility to include code to test for errors. It's easy to overlook potential errors; if one of these should actually occur, the program often continues running, albeit not very well. Newer languages such as C++, Java, and C# have an "exception handling" feature that makes it easier to detect and respond to errors.

24.1 The <assert.h> Header: Diagnostics

```
void assert(scalar expression);
```

assert

assert, which is defined in the <assert.h> header, allows a program to monitor its own behavior and detect possible problems at an early stage.

Although assert is actually a macro, it's designed to be used like a function. It has one argument, which must be an "assertion"—an expression that we expect to be true under normal circumstances. Each time assert is executed, it tests the value of its argument. If the argument has a nonzero value, assert does nothing. If the argument's value is zero, assert writes a message to stderr (the standard error stream) and calls the abort function to terminate program execution.

stderr stream ➤22.1
abort function ➤26.2

For example, let's say that the file demo.c declares an array a of length 10. We're concerned that the statement

```
a[i] = 0;
```

in demo.c might cause the program to fail because i isn't between 0 and 9. We can use assert to check this condition before we perform the assignment to a[i]:

```
assert(0 <= i && i < 10);    /* checks subscript first */
a[i] = 0;                    /* now does the assignment */
```

If i's value is less than 0 or greater than or equal to 10, the program will terminate after displaying a message like the following one:

```
Assertion failed: 0 <= i && i < 10, file demo.c, line 109
```

C99

C99 changes assert in a couple of minor ways. The C89 standard states that the argument to assert must have int type. The C99 standard relaxes this requirement, allowing the argument to have any scalar type (hence the word *scalar* in the prototype for assert). This change allows the argument to be a floating-point number or a pointer, for example. Also, C99 requires that a failed assert display the name of the function in which it appears. (C89 requires only that assert display the argument—in text form—along with the name of the source file and the source line number). The suggested form of the message is

```
Assertion failed: expression, function abc, file xyz, line nnn.
```

The exact form of the message produced by assert may vary from one compiler to another, although it should always contain the information required by the standard. For example, the GCC compiler produces the following message in the situation described earlier:

```
a.out: demo.c:109: main: Assertion `0 <= i && i < 10' failed.
```

assert has one disadvantage: it slightly increases the running time of a program because of the extra check it performs. Using assert once in a while probably won't have any great effect on a program's speed, but even this small time penalty may be unacceptable in critical applications. As a result, many programmers use assert during testing, then disable it when the program is finished. Disabling assert is easy: we need only define the macro NDEBUG prior to including the <assert.h> header:

```
#define NDEBUG
#include <assert.h>
```

The value of NDEBUG doesn't matter, just the fact that it's defined. If the program should fail later, we can reactivate assert by removing NDEBUG's definition.

Avoid putting an expression that has a side effect—including a function call—inside an assert; if assert is disabled at a later date, the expression won't be evaluated. Consider the following example:

```
assert((p = malloc(n)) != NULL);
```

If NDEBUG is defined, assert will be ignored and malloc won't be called.

24.2 The <errno.h> Header: Errors

Some functions in the standard library indicate failure by storing an error code (a positive integer) in errno, an int variable declared in <errno.h>. (errno may actually be a macro. If so, the C standard requires that it represent an lvalue, allowing us to use it like a variable.) Most of the functions that rely on errno belong to <math.h>, but there are a few in other parts of the library.

lvalues ➤4.2

Let's say that we need to use a library function that signals an error by storing a value in errno. After calling the function, we can check whether the value of errno is nonzero; if so, an error occurred during the function call. For example, suppose that we want to check whether a call of the sqrt (square root) function has failed. Here's what the code would look like:

sqrt function ➤23.3

```
errno = 0;
y = sqrt(x);
if (errno != 0) {
  fprintf(stderr, "sqrt error; program terminated.\n");
  exit(EXIT_FAILURE);
}
```

Q&A When errno is used to detect an error in a call of a library function, it's important to store zero in errno before calling the function. Although errno is zero at the beginning of program execution, it could have been altered by a later function call. Library functions never clear errno; that's the program's responsibility.

Q&A

The value stored in `errno` when an error occurs is often either `EDOM` or `ERANGE`. (Both are macros defined in `<errno.h>`.) These macros represent the two kinds of errors that can occur when a math function is called:

- ***Domain errors*** (`EDOM`): An argument passed to a function is outside the function's domain. For example, passing a negative number to `sqrt` causes a domain error.
- ***Range errors*** (`ERANGE`): A function's return value is too large to be represented in the function's return type. For example, passing 1000 to the `exp` function usually causes a range error, because e^{1000} is too large to represent as a `double` on most computers.

exp function ➤23.3

Some functions can experience both kinds of errors; by comparing `errno` to `EDOM` or `ERANGE`, we can determine which error occurred.

C99

C99 adds the `EILSEQ` macro to `<errno.h>`. Library functions in certain headers—especially the `<wchar.h>` header—store the value of `EILSEQ` in `errno` when an encoding error occurs.

`<wchar.h>` header ➤25.5
encoding error ➤22.3

The `perror` and `strerror` Functions

```
void perror(const char *s);                    from <stdio.h>
char *strerror(int errnum);                    from <string.h>
```

We'll now look at two functions that are related to the `errno` variable, although neither function belongs to `<errno.h>`.

perror

When a library function stores a nonzero value in `errno`, we may want to display a message that indicates the nature of the error. One way to do this is to call the `perror` function (declared in `<stdio.h>`), which prints the following items, in the order shown: (1) its argument, (2) a colon, (3) a space, (4) an error message determined by the value of `errno`, and (5) a new-line character. `perror` writes to the `stderr` stream, not to standard output.

`stderr` stream ➤22.1

Here's how we might use `perror`:

```
errno = 0;
y = sqrt(x);
if (errno != 0) {
  perror("sqrt error");
  exit(EXIT_FAILURE);
}
```

If the call of `sqrt` fails because of a domain error, `perror` will generate the following output:

```
sqrt error: Numerical argument out of domain
```

The error message that `perror` displays after `sqrt error` is implementation-defined. In this example, `Numerical argument out of domain` is the mes-

sage that corresponds to the EDOM error. An ERANGE error usually produces a different message, such as `Numerical result out of range`.

strerror

The `strerror` function belongs to `<string.h>`. When passed an error code, `strerror` returns a pointer to a string describing the error. For example, the call

```
puts(strerror(EDOM));
```

might print

```
Numerical argument out of domain
```

The argument to `strerror` is usually one of the values of `errno`, but `strerror` will return a string for any integer passed to it.

`strerror` is closely related to the `perror` function. The error message that `perror` displays is the same message that `strerror` would return if passed `errno` as its argument.

24.3 The <signal.h> Header: Signal Handling

The `<signal.h>` header provides facilities for handling exceptional conditions, known as ***signals***. Signals fall into two categories: run-time errors (such as division by zero) and events caused outside the program. Many operating systems, for example, allow users to interrupt or kill running programs; these events are treated as signals in C. When an error or external event occurs, we say that a signal has been ***raised***. Many signals are asynchronous: they can happen at any time during program execution, not just at certain points that are known to the programmer. Since signals may occur at unexpected times, they have to be dealt with in a unique way.

This section covers signals as they're described in the C standard. Signals play a more prominent role in UNIX than you might expect from their limited coverage here. For information about UNIX signals, consult one of the UNIX programming books listed in the bibliography.

Signal Macros

Q&A `<signal.h>` defines a number of macros that represent signals; Table 24.1 lists these macros and their meanings. The value of each macro is a positive integer constant. C implementations are allowed to provide other signal macros, as long as their names begin with `SIG` followed by an upper-case letter. (UNIX implementations, in particular, provide a large number of additional signal macros.)

The C standard doesn't require that the signals in Table 24.1 be raised automatically, since not all of them may be meaningful for a particular computer and operating system. Most implementations support at least some of these signals.

Table 24.1
Signals

Name	*Meaning*
SIGABRT	Abnormal termination (possibly caused by a call of abort)
SIGFPE	Error during an arithmetic operation (possibly division by zero or overflow)
SIGILL	Invalid instruction
SIGINT	Interrupt
SIGSEGV	Invalid storage access
SIGTERM	Termination request

The signal Function

```
void (*signal(int sig, void (*func)(int)))(int);
```

signal

<signal.h> provides two functions: raise and signal. We'll start with signal, which installs a signal-handling function for use later if a given signal should occur. signal is much easier to use than you might expect from its rather intimidating prototype. Its first argument is the code for a particular signal; the second argument is a pointer to a function that will handle the signal if it's raised later in the program. For example, the following call of signal installs a handler for the SIGINT signal:

```
signal(SIGINT, handler);
```

handler is the name of a signal-handling function. If the SIGINT signal occurs later during program execution, handler will be called automatically.

Every signal-handling function must have an int parameter and a return type of void. When a particular signal is raised and its handler is called, the handler will be passed the code for the signal. Knowing which signal caused it to be called can be useful for a signal handler; in particular, it allows us to use the same handler for several different signals.

abort function ➤26.2
static storage duration ➤18.2
Q&A

A signal-handling function can do a variety of things. Possibilities include ignoring the signal, performing some sort of error recovery, or terminating the program. Unless it's invoked by abort or raise, however, a signal handler shouldn't call a library function or attempt to use a variable with static storage duration. (There are a few exceptions to these rules, however.)

If a signal-handling function returns, the program resumes executing from the point at which the signal occurred, except in two cases: (1) If the signal was SIGABRT, the program will terminate (abnormally) when the handler returns. (2) The effect of returning from a function that has handled SIGFPE is undefined. (In other words, don't do it.)

Although signal has a return value, it's often discarded. The return value, a pointer to the previous handler for the specified signal, can be saved in a variable if desired. In particular, if we plan to restore the original signal handler later, we need to save signal's return value:

```
void (*orig_handler)(int);    /* function pointer variable */
...
```

```
orig_handler = signal(SIGINT, handler);
```

This statement installs `handler` as the handler for `SIGINT` and then saves a pointer to the original handler in the `orig_handler` variable. To restore the original handler later, we'd write

```
signal(SIGINT, orig_handler); /* restores original handler */
```

Predefined Signal Handlers

Instead of writing our own signal handlers, we have the option of using one of the predefined handlers that `<signal.h>` provides. There are two of these, each represented by a macro:

- **`SIG_DFL`.** `SIG_DFL` handles signals in a "default" way. To install `SIG_DFL`, we'd use a call such as

  ```
  signal(SIGINT, SIG_DFL);    /* use default handler */
  ```

  The effect of calling `SIG_DFL` is implementation-defined, but in most cases it causes program termination.

- **`SIG_IGN`.** The call

  ```
  signal(SIGINT, SIG_IGN);    /* ignore SIGINT signal */
  ```

  specifies that `SIGINT` is to be ignored if it should be raised later.

In addition to `SIG_DFL` and `SIG_IGN`, the `<signal.h>` header may provide other signal handlers; their names must begin with `SIG_` followed by an uppercase letter. At the beginning of program execution, the handler for each signal is initialized to either `SIG_DFL` or `SIG_IGN`, depending on the implementation.

`<signal.h>` defines another macro, `SIG_ERR`, that looks like it should be a signal handler. `SIG_ERR` is actually used to test for an error when installing a signal handler. If a call of `signal` is unsuccessful—it can't install a handler for the specified signal—it returns `SIG_ERR` and stores a positive value in `errno`. Thus, to test whether `signal` has failed, we could write

```
if (signal(SIGINT, handler) == SIG_ERR) {
  perror("signal(SIGINT, handler) failed");
  ...
}
```

There's one tricky aspect to the entire signal-handling mechanism: what happens if a signal is raised by the function that handles that signal? To prevent infinite recursion, the C89 standard prescribes a two-step process when a signal is raised for which a signal-handling function has been installed by the programmer. First, either the handler for that signal is reset to `SIG_DFL` (the default handler) or else the signal is blocked from occurring while the handler is executing. (`SIGILL` is a special case; neither action is required when `SIGILL` is raised.) Only then is the handler provided by the programmer called.

After a signal has been handled, whether or not the handler needs to be reinstalled is implementation-defined. UNIX implementations typically leave the signal handler installed after it's been used, but other implementations may reset the handler to `SIG_DFL`. In the latter case, the handler can reinstall itself by calling `signal` before it returns.

C99

C99 changes the signal-handling process in a few minor ways. When a signal is raised, an implementation may choose to disable not just that signal but others as well. If a signal-handling function returns from handling a `SIGILL` or `SIGSEGV` signal (as well as a `SIGFPE` signal), the effect is undefined. C99 also adds the restriction that if a signal occurs as a result of calling the `abort` function or the `raise` function, the signal handler itself must not call `raise`.

The `raise` Function

```
int raise(int sig);
```

raise

Although signals usually arise from run-time errors or external events, it's occasionally handy for a program to cause a signal to occur. The `raise` function does just that. The argument to `raise` specifies the code for the desired signal:

```
raise(SIGABRT);    /* raises the SIGABRT signal */
```

The return value of `raise` can be used to test whether the call was successful: zero indicates success, while a nonzero value indicates failure.

PROGRAM

Testing Signals

The following program illustrates the use of signals. First, it installs a custom handler for the `SIGINT` signal (carefully saving the original handler), then calls `raise_sig` to raise that signal. Next, it installs `SIG_IGN` as the handler for the `SIGINT` signal and calls `raise_sig` again. Finally, it reinstalls the original handler for `SIGINT`, then calls `raise_sig` one last time.

tsignal.c

```
/* Tests signals */

#include <signal.h>
#include <stdio.h>

void handler(int sig);
void raise_sig(void);

int main(void)
{
  void (*orig_handler)(int);
```

```
  printf("Installing handler for signal %d\n", SIGINT);
  orig_handler = signal(SIGINT, handler);
  raise_sig();

  printf("Changing handler to SIG_IGN\n");
  signal(SIGINT, SIG_IGN);
  raise_sig();

  printf("Restoring original handler\n");
  signal(SIGINT, orig_handler);
  raise_sig();

  printf("Program terminates normally\n");
  return 0;
}

void handler(int sig)
{
  printf("Handler called for signal %d\n", sig);
}

void raise_sig(void)
{
  raise(SIGINT);
}
```

Incidentally, the call of `raise` doesn't need to be in a separate function. I defined `raise_sig` simply to make a point: regardless of where a signal is raised—whether it's in `main` or in some other function—it will be caught by the most recently installed handler for that signal.

The output of this program can vary somewhat. Here's one possibility:

```
Installing handler for signal 2
Handler called for signal 2
Changing handler to SIG_IGN
Restoring original handler
```

From this output, we see that our implementation defines `SIGINT` to be 2 and that the original handler for `SIGINT` must have been `SIG_DFL`. (If it had been `SIG_IGN`, we'd also see the message `Program terminates normally`.) Finally, we observe that `SIG_DFL` caused the program to terminate without displaying an error message.

24.4 The <setjmp.h> Header: Nonlocal Jumps

```
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

goto statement ➤6.4

Normally, a function returns to the point at which it was called. We can't use a `goto` statement to make it go elsewhere, because a `goto` can jump only to a label within the same function. The `<setjmp.h>` header, however, makes it possible for one function to jump directly to another function without returning.

The most important items in `<setjmp.h>` are the `setjmp` macro and the `longjmp` function. `setjmp` "marks" a place in a program; `longjmp` can then be used to return to that place later. Although this powerful mechanism has a variety of potential applications, it's used primarily for error handling.

setjmp

To mark the target of a future jump, we call `setjmp`, passing it a variable of type `jmp_buf` (declared in `<setjmp.h>`). `setjmp` stores the current "environment" (including a pointer to the location of the `setjmp` itself) in the variable for later use in a call of `longjmp`; it then returns zero.

Q&A

longjmp

Returning to the point of the `setjmp` is done by calling `longjmp`, passing it the same `jmp_buf` variable that we passed to `setjmp`. After restoring the environment represented by the `jmp_buf` variable, `longjmp` will—here's where it gets tricky—*return from the* `setjmp` *call*. `setjmp`'s return value this time is `val`, the second argument to `longjmp`. (If `val` is 0, `setjmp` returns 1.)

⚠ Be sure that the argument to `longjmp` was previously initialized by a call of `setjmp`. It's also important that the function containing the original call of `setjmp` must not have returned prior to the call of `longjmp`. If either restriction is violated, calling `longjmp` results in undefined behavior. (The program will probably crash.)

To summarize, `setjmp` returns zero the first time it's called; later, `longjmp` transfers control back to the original call of `setjmp`, which this time returns a nonzero value. Got it? Perhaps we need an example...

PROGRAM Testing `setjmp/longjmp`

The following program uses `setjmp` to mark a place in `main`; the function `f2` later returns to that place by calling `longjmp`.

tsetjmp.c

```
/* Tests setjmp/longjmp */

#include <setjmp.h>
#include <stdio.h>

jmp_buf env;

void f1(void);
void f2(void);

int main(void)
{
  if (setjmp(env) == 0)
    printf("setjmp returned 0\n");
```

```
    else {
      printf("Program terminates: longjmp called\n");
      return 0;
    }

    f1();
    printf("Program terminates normally\n");
    return 0;
  }

  void f1(void)
  {
    printf("f1 begins\n");
    f2();
    printf("f1 returns\n");
  }

  void f2(void)
  {
    printf("f2 begins\n");
    longjmp(env, 1);
    printf("f2 returns\n");
  }
```

The output of this program will be

```
setjmp returned 0
f1 begins
f2 begins
Program terminates: longjmp called
```

The original call of `setjmp` returns 0, so `main` calls `f1`. Next, `f1` calls `f2`, which uses `longjmp` to transfer control back to `main` instead of returning to `f1`. When `longjmp` is executed, control goes back to the `setjmp` call. This time, `setjmp` returns 1 (the value specified in the `longjmp` call).

Q & A

Q: You said that it's important to store zero in `errno` before calling a library function that may change it, but I've seen UNIX programs that test `errno` without ever setting it to zero. What's the story? [p. 629]

A: UNIX programs often contain calls of functions that belong to the operating system. These ***system calls*** rely on `errno`, but they use it in a slightly different way than described in this chapter. When such a call fails, it returns a special value (such as –1 or a null pointer) in addition to storing a value in `errno`. Programs don't need to store zero in `errno` before such a call, because the function's return value alone indicates that an error occurred. Some functions in the C standard library work this way as well, using `errno` not so much to signal an error as to specify which error it was.

Q: My version of `<errno.h>` defines other macros besides `EDOM` and `ERANGE`. Is this practice legal? [p. 630]

A: Yes. The C standard allows macros that represent other error conditions, provided that their names begin with the letter `E` followed by a digit or an upper-case letter. UNIX implementations typically define a huge number of such macros.

Q: Some of the macros that represent signals have cryptic names, like `SIGFPE` and `SIGSEGV`. Where do these names come from? [p. 631]

A: The names of these signals date back to the early C compilers, which ran on a DEC PDP-11. The PDP-11 hardware could detect errors with names like "Floating Point Exception" and "Segmentation Violation."

Q: OK, I'm curious. Unless it's invoked by `abort` or `raise`, a signal handler shouldn't call a standard library function, but you said there were exceptions to this rule. What are they? [p. 632]

A: A signal handler is allowed to call the `signal` function, provided that the first argument is the signal that it's handling at the moment. This proviso is important, because it allows a signal handler to reinstall itself. In C99, a signal handler may also call the `abort` function or the `_Exit` function.

C99

`_Exit` function ➤26.2

***Q: Following up on the previous question, a signal handler normally isn't supposed to access variables with static storage duration. What's the exception to this rule?**

A: That one's a bit harder. The answer involves a type named `sig_atomic_t` that's declared in the `<signal.h>` header. `sig_atomic_t` is an integer type that can be accessed "as an atomic entity," according to the C standard. In other words, the CPU can fetch a `sig_atomic_t` value from memory or store one in memory with a single machine instruction, rather than using two or more machine instructions. `sig_atomic_t` is often defined to be `int`, since most CPUs can load or store an `int` value in one instruction.

That brings us to the exception to the rule that a signal-handling function isn't supposed to access static variables. The C standard allows a signal handler to store a value in a `sig_atomic_t` variable—even one with static storage duration—provided that it's declared `volatile`. To see the reason for this arcane rule, consider what might happen if a signal handler were to modify a static variable that's of a type that's wider than `sig_atomic_t`. If the program had fetched part of the variable from memory just before the signal occurred, then completed the fetch after the signal is handled, it could end up with a garbage value. `sig_atomic_t` variables can be fetched in a single step, so this problem doesn't occur. Declaring the variable to be `volatile` warns the compiler that the variable's value may change at any time. (A signal could suddenly be raised, invoking a signal handler that modifies the variable.)

`volatile` type qualifier ➤20.3

Q: The `tsignal.c` program calls `printf` from inside a signal handler. Isn't that illegal?

A: A signal-handling function invoked as a result of `raise` or `abort` may call library functions. `tsignal.c` uses `raise` to invoke the signal handler.

Q: How can `setjmp` modify the argument that's passed to it? I thought that C always passed arguments by value. [p. 636]

A: The C standard says that `jmp_buf` must be an array type, so `setjmp` is actually being passed a pointer.

Q: I'm having trouble with `setjmp`. Are there any restrictions on how it can be used?

A: According to the C standard, there are only two legal ways to use `setjmp`:

- As the expression in an expression statement (possibly cast to `void`).
- As part of the controlling expression in an `if`, `switch`, `while`, `do`, or `for` statement. The entire controlling expression must have one of the following forms, where *constexpr* is an integer constant expression and *op* is a relational or equality operator:

```
setjmp(...)
!setjmp(...)
constexpr op setjmp(...)
setjmp(...) op constexpr
```

Using `setjmp` in any other way causes undefined behavior.

Q: After a program has executed a call of `longjmp`, what are the values of the variables in the program?

A: Most variables retain the values they had at the time of the `longjmp`. However, an automatic variable inside the function that contains the `setjmp` has an indeterminate value unless it was declared `volatile` or it hasn't been modified since the `setjmp` was performed.

Q: Is it legal to call `longjmp` inside a signal handler?

A: Yes, provided that the signal handler wasn't invoked because of a signal raised during the execution of a signal handler. (C99 removes this restriction.)

C99

Exercises

Section 24.1

1. (a) Assertions can be used to test for two kinds of problems: (1) problems that should never occur if the program is correct, and (2) problems that are beyond the control of the program. Explain why `assert` is best suited for problems in the first category.

   (b) Give three examples of problems that are beyond the control of the program.

2. Write a call of `assert` that causes a program to terminate if a variable named `top` has the value `NULL`.

3. Modify the `stackADT2.c` file of Section 19.4 so that it uses `assert` to test for errors instead of using `if` statements. (Note that the `terminate` function is no longer necessary and can be removed.)

Section 24.2

W 4. (a) Write a "wrapper" function named `try_math_fcn` that calls a math function (assumed to have a `double` argument and return a `double` value) and then checks whether the call succeeded. Here's how we might use `try_math_fcn`:

```
y = try_math_fcn(sqrt, x, "Error in call of sqrt");
```

If the call `sqrt(x)` is successful, `try_math_fcn` returns the value computed by `sqrt`. If the call fails, `try_math_fcn` calls `perror` to print the message `Error in call of sqrt`, then calls `exit` to terminate the program.

(b) Write a macro that has the same effect as `try_math_fcn` but builds the error message from the function's name:

```
y = TRY_MATH_FCN(sqrt, x);
```

If the call of `sqrt` fails, the message will be `Error in call of sqrt`. *Hint:* Have `TRY_MATH_FCN` call `try_math_fcn`.

Section 24.4

W 5. In the `inventory.c` program (see Section 16.3), the `main` function has a `for` loop that prompts the user to enter an operation code, reads the code, and then calls either `insert`, `search`, `update`, or `print`. Add a call of `setjmp` to `main` in such a way that a subsequent call of `longjmp` will return to the `for` loop. (After the `longjmp`, the user will be prompted for an operation code, and the program will continue normally.) `setjmp` will need a `jmp_buf` variable; where should it be declared?

25 International Features

If your computer speaks English
it was probably made in Japan.

For many years, C wasn't especially suitable for use in non-English-speaking countries. C originally assumed that characters were always single bytes and that all computers recognized the characters #, [, \,], ^, {, |, }, and ~, which are needed to write programs. Unfortunately, these assumptions aren't valid in all parts of the world. As a result, the experts who created C89 added language features and libraries in an effort to make C a more international language.

In 1994, Amendment 1 to the ISO C standard was approved, creating an enhanced version of C89 that's sometimes known as C94 or C95. This amendment provides additional library support for international programming via the digraph language feature and the `<iso646.h>`, `<wchar.h>`, and `<wctype.h>` headers. C99 adds even more support for internationalization in the form of universal character names. This chapter covers all of C's international features, whether they come from C89, Amendment 1, or C99. I'll flag the Amendment 1 changes as C99 changes, although they actually predate C99.

The `<locale.h>` header (Section 25.1) provides functions that allow a program to tailor its behavior to a particular "locale"—often a country or other geographical area in which a particular language is spoken. Multibyte characters and wide characters (Section 25.2) enable programs to work with large character sets such as those found in Asian countries. Digraphs, trigraphs, and the `<iso646.h>` header (Section 25.3) make it possible to write programs on computers that lack some of the characters normally used in C programming. Universal character names (Section 25.4) allow programmers to embed characters from the Universal Character Set into the source code of a program. The `<wchar.h>` header (Section 25.5) supplies functions for wide-character input/output and wide-string manipulation. Finally, the `<wctype.h>` header (Section 25.6) provides wide-character classification and case-mapping functions.

25.1 The <locale.h> Header: Localization

The <locale.h> header provides functions to control portions of the C library whose behavior varies from one locale to another. (A ***locale*** is typically a country or a region in which a particular language is spoken.)

Locale-dependent aspects of the library include:

- ***Formatting of numerical quantities.*** In some locales, for example, the decimal point is a period (297.48), while in others it's a comma (297,48).
- ***Formatting of monetary quantities.*** For example, the currency symbol varies from country to country.
- ***Character set.*** The character set often depends on the language in a particular locale. Asian countries usually require a much larger character set than Western countries.
- ***Appearance of date and time.*** In some locales, it's customary to put the month first when writing a date (8/24/2012); in others, the day goes first (24/8/2012).

Categories

By changing locale, a program can adapt its behavior to a different area of the world. But a locale change can affect many parts of the library, some of which we might prefer not to alter. Fortunately, we're not required to change all aspects of a locale at the same time. Instead, we can use one of the following macros to specify a ***category:***

<string.h> header ➤23.6

<ctype.h> header ➤23.5

- LC_COLLATE. Affects the behavior of two string-comparison functions, strcoll and strxfrm. (Both functions are declared in <string.h>).
- LC_CTYPE. Affects the behavior of the functions in <ctype.h> (except isdigit and isxdigit). Also affects the multibyte and wide-character functions discussed in this chapter.
- LC_MONETARY. Affects the monetary formatting information returned by the localeconv function.

numeric conversion functions ➤26.2

- LC_NUMERIC. Affects the decimal-point character used by formatted I/O functions (like printf and scanf) and the numeric conversion functions (such as strtod) in <stdlib.h>. Also affects the nonmonetary formatting information returned by localeconv.

strftime function ➤26.3

C99

wcsftime function ➤25.5

- LC_TIME. Affects the behavior of the strftime function (declared in <time.h>), which converts a time into a character string. In C99, also affects the behavior of the wcsftime function.

Implementations are free to provide additional categories and define LC_ macros not listed above. For example, most UNIX systems provide an LC_MESSAGES category, which affects the format of affirmative and negative system responses.

The setlocale Function

```
char *setlocale(int category, const char *locale);
```

setlocale

The `setlocale` function changes the current locale, either for a single category or for all categories. If the first argument is one of the macros `LC_COLLATE`, `LC_CTYPE`, `LC_MONETARY`, `LC_NUMERIC`, or `LC_TIME`, a call of `setlocale` affects only a single category. If the first argument is `LC_ALL`, the call affects all categories. The C standard defines only two values for the second argument: `"C"` and `""`. Other locales, if any, depend on the implementation.

At the beginning of program execution, the call

```
setlocale(LC_ALL, "C");
```

occurs behind the scenes. In the `"C"` locale, library functions behave in the "normal" way, and the decimal point is a period.

Changing locale after the program has begun execution requires an explicit call of `setlocale`. Calling `setlocale` with `""` as the second argument switches to the ***native locale,*** allowing the program to adapt its behavior to the local environment. The C standard doesn't define the exact effect of switching to the native locale. Some implementations of `setlocale` check the execution environment (in the same way as `getenv`) for an environment variable with a particular name (perhaps the same as the category macro). Other implementations don't do anything at all. (The standard doesn't require `setlocale` to have any effect. Of course, a library whose version of `setlocale` does nothing isn't likely to sell too well in some parts of the world.)

getenv function ➤26.2

Locales

Locales other than `"C"` and `""` vary from one compiler to another. The GNU C library, known as `glibc`, provides a `"POSIX"` locale, which is the same as the `"C"` locale. `glibc`, which is used by Linux, allows additional locales to be installed if desired. These locales have the form

language [_*territory*] [.*codeset*] [@*modifier*]

where each bracketed item is optional. Possible values for *language* are listed in a standard known as ISO 639, *territory* comes from another standard (ISO 3166), and *codeset* specifies a character set or an encoding of a character set. Here are a few examples:

`"swedish"`
`"en_GB"` (English – United Kingdom)
`"en_IE"` (English – Ireland)
`"fr_CH"` (French – Switzerland)

There are several variations on the `"en_IE"` locale, including `"en_IE@euro"` (using the euro currency), `"en_IE.iso88591"` (using the ISO/IEC 8859-1 character set),

"en_IE.iso885915@euro" (using the ISO/IEC 8859-15 character set and the euro currency), and "en_IE.utf8" (using the UTF-8 encoding of the Unicode character set).

UTF-8 ➤25.2

Linux and other versions of UNIX support the locale command, which can be used to get locale information. One use of the locale command is to get a list of all available locales, which can be done by entering

```
locale -a
```

at the command line.

Because locale information is becoming increasingly important, the Unicode Consortium created the Common Locale Data Repository (CLDR) project to establish a standard set of locales. More information about the CLDR project can be found at *www.unicode.org/cldr/*.

When a call of setlocale succeeds, it returns a pointer to a string associated with the category in the new locale. (The string might be the locale name itself, for example.) On failure, setlocale returns a null pointer.

setlocale can also be used as a query function. If its second argument is a null pointer, setlocale returns a pointer to a string associated with the category in the *current* locale. This feature is especially useful if the first argument is LC_ALL, since it allows us to fetch the current settings for all categories. A string returned by setlocale can be saved (by copying it into a variable) and then used in a later call of setlocale.

Q&A

The localeconv Function

```
struct lconv *localeconv(void);
```

localeconv

Although we can ask setlocale about the current locale, the information that it returns isn't necessarily in the most useful form. To find out highly specific information about the current locale (What's the decimal-point character? What's the currency symbol?), we need localeconv, the only other function declared in <locale.h>.

localeconv returns a pointer to a structure of type struct lconv. The members of this structure contain detailed information about the current locale. The structure has static storage duration and may be modified by a later call of localeconv or setlocale. Be sure to extract the desired information from the lconv structure before it's wiped out by one of these functions.

Some members of the lconv structure have char * type; other members have char type. Table 25.1 lists the char * members. The first three members describe the formatting of nonmonetary quantities, while the others deal with monetary quantities. The table also shows the value of each member in the "C" locale (the default); a value of "" means "not available."

The grouping and mon_grouping members deserve special mention.

Table 25.1
char * Members of
lconv Structure

	Name	Value in "C" Locale	Description
Nonmonetary	decimal_point	"."	Decimal-point character
	thousands_sep	""	Character used to separate groups of digits before decimal point
	grouping	""	Sizes of digit groups
Monetary	mon_decimal_point	""	Decimal-point character
	mon_thousands_sep	""	Character used to separate groups of digits before decimal point
	mon_grouping	""	Sizes of digit groups
	positive_sign	""	String indicating nonnegative quantity
	negative_sign	""	String indicating negative quantity
	currency_symbol	""	Local currency symbol
	int_curr_symbol	""	International currency symbol†

†A three-letter abbreviation followed by a separator (often a space or a period). For example, the international currency symbols for Switzerland, the United Kingdom, and the United States are "CHF ", "GBP ", and "USD ", respectively.

Each character in these strings specifies the size of one group of digits. (Grouping takes place from right to left, starting at the decimal point.) A value of CHAR_MAX indicates that no further grouping is to be performed; 0 indicates that the previous element should be used for the remaining digits. For example, the string "\3" (\3 followed by \0) indicates that the first group should have 3 digits, then all other digits should be grouped in 3's as well.

The char members of the lconv structure are divided into two groups. The members of the first group (Table 25.2) affect the *local* formatting of monetary quantities. The members of the second group (Table 25.3) affect the *international* formatting of monetary quantities. All but one of the members in Table 25.3 were added in C99. As Tables 25.2 and 25.3 show, the value of each char member in the "C" locale is CHAR_MAX, which means "not available."

C99

Table 25.2
char Members of
lconv Structure
(Local Formatting)

Name	Value in "C" Locale	Description
frac_digits	CHAR_MAX	Number of digits after decimal point
p_cs_precedes	CHAR_MAX	1 if currency_symbol precedes nonnegative quantity; 0 if it succeeds quantity
n_cs_precedes	CHAR_MAX	1 if currency_symbol precedes negative quantity; 0 if it succeeds quantity
p_sep_by_space	CHAR_MAX	Separation of currency_symbol and sign string from nonnegative quantity (see Table 25.4)
n_sep_by_space	CHAR_MAX	Separation of currency_symbol and sign string from negative quantity (see Table 25.4)
p_sign_posn	CHAR_MAX	Position of positive_sign for nonnegative quantity (see Table 25.5)
n_sign_posn	CHAR_MAX	Position of negative_sign for negative quantity (see Table 25.5)

Table 25.3 char Members of lconv Structure (International Formatting)

Name	*Value in "C" Locale*	*Description*
int_frac_digits	CHAR_MAX	Number of digits after decimal point
int_p_cs_precedes†	CHAR_MAX	1 if int_curr_symbol precedes nonnegative quantity; 0 if it succeeds quantity
int_n_cs_precedes†	CHAR_MAX	1 if int_curr_symbol precedes negative quantity; 0 if it succeeds quantity
int_p_sep_by_space†	CHAR_MAX	Separation of int_curr_symbol and sign string from nonnegative quantity (see Table 25.4)
int_n_sep_by_space†	CHAR_MAX	Separation of int_curr_symbol and sign string from negative quantity (see Table 25.4)
int_p_sign_posn†	CHAR_MAX	Position of positive_sign for nonnegative quantity (see Table 25.5)
int_n_sign_posn†	CHAR_MAX	Position of negative_sign for negative quantity (see Table 25.5)

†C99 only

Table 25.4 explains the meaning of the values of the p_sep_by_space, n_sep_by_space, int_p_sep_by_space, and int_n_sep_by_space members. The meaning of p_sep_by_space and n_sep_by_space has changed in C99. In C89, there are only two possible values for these members: 1 (if there's a space between currency_symbol and a monetary quantity) or 0 (if there's not).

C99

Table 25.4 Values of ...sep_by_space Members

Value	*Meaning*
0	No space separates currency symbol and quantity.
1	If currency symbol and sign are adjacent, a space separates them from quantity; otherwise, a space separates currency symbol from quantity.
2	If currency symbol and sign are adjacent, a space separates them; otherwise, a space separates sign from quantity.

Table 25.5 explains the meaning of the values of the p_sign_posn, n_sign_posn, int_p_sign_posn and int_n_sign_posn members.

Table 25.5 Values of ...sign_posn Members

Value	*Meaning*
0	Parentheses surround quantity and currency symbol
1	Sign precedes quantity and currency symbol
2	Sign succeeds quantity and currency symbol
3	Sign immediately precedes currency symbol
4	Sign immediately succeeds currency symbol

To see how the members of the lconv structure might vary from one locale to another, let's look at two examples. Table 25.6 shows typical values of the monetary lconv members for the U.S.A. and Finland (which uses the euro as its currency).

Table 25.6 Typical Values of Monetary `lconv` Members for U.S.A. and Finland

Member	*U.S.A.*	*Finland*
mon_decimal_point	"."	","
mon_thousands_sep	","	" "
mon_grouping	"\3"	"\3"
positive_sign	""	""
negative_sign	"-"	"-"
currency_symbol	"$"	"EUR"
frac_digits	2	2
p_cs_precedes	1	0
n_cs_precedes	1	0
p_sep_by_space	0	2
n_sep_by_space	0	2
p_sign_posn	1	1
n_sign_posn	1	1
int_curr_symbol	"USD "	"EUR "
int_frac_digits	2	2
int_p_cs_precedes	1	0
int_n_cs_precedes	1	0
int_p_sep_by_space	1	2
int_n_sep_by_space	1	2
int_p_sign_posn	1	1
int_n_sign_posn	1	1

Here's how the monetary quantity 7593.86 would be formatted in the two locales, depending on the sign of the quantity and whether the formatting is local or international:

	U.S.A.	*Finland*
Local format (positive)	`$7,593.86`	`7 593,86 EUR`
Local format (negative)	`-$7,593.86`	`- 7 593,86 EUR`
International format (positive)	`USD 7,593.86`	`7 593,86 EUR`
International format (negative)	`-USD 7,593.86`	`- 7 593,86 EUR`

Keep in mind that none of C's library functions are able to format monetary quantities automatically. It's up to the programmer to use the information in the `lconv` structure to accomplish the formatting.

25.2 Multibyte Characters and Wide Characters

Latin-1 ➤7.3

One of the biggest problems in adapting programs to different locales is the character-set issue. ASCII and its extensions, which include Latin-1, are the most popular character sets in North America. Elsewhere, the situation is more complicated. In many countries, computers employ character sets that are similar to ASCII, but lack certain characters; we'll discuss this issue further in Section 25.3. Other countries, especially those in Asia, face a different problem: written languages that require a very large character set, usually numbering in the thousands.

Changing the meaning of type `char` to handle larger character sets isn't possible, since `char` values are—by definition—limited to single bytes. Instead, C allows compilers to provide an ***extended character set.*** This character set may be used for writing C programs (in comments and strings, for example), in the environment in which the program is run, or in both places. C provides two techniques for encoding an extended character set: multibyte characters and wide characters. It also supplies functions that convert from one kind of encoding to the other.

Q&A

Multibyte Characters

In a ***multibyte character*** encoding, each extended character is represented by a sequence of one or more bytes. The number of bytes may vary, depending on the character. C requires that any extended character set include certain essential characters (letters, digits, operators, punctuation, and white-space characters); these characters must be single bytes. Other bytes can be interpreted as the beginning of a multibyte character.

Japanese Character Sets

The Japanese employ several different writing systems. The most complex, *kanji*, consists of thousands of symbols—far too many to represent in a one-byte encoding. (*Kanji* symbols actually come from Chinese, which has a similar problem with large character sets.) There's no single way to encode *kanji*; common encodings include JIS (Japanese Industrial Standard), Shift-JIS (the most popular encoding), and EUC (Extended UNIX Code).

Some multibyte character sets rely on a ***state-dependent encoding.*** In this kind of encoding, each sequence of multibyte characters begins in an ***initial shift state.*** Certain bytes encountered later (known as a ***shift sequence***) may change the shift state, affecting the meaning of subsequent bytes. Japan's JIS encoding, for example, mixes one-byte codes with two-byte codes; "escape sequences" embedded in strings indicate when to switch between one-byte and two-byte modes. (In contrast, the Shift-JIS encoding is not state-dependent. Each character requires either one or two bytes, but the first byte of a two-byte character can always be distinguished from a one-byte character.)

In any encoding, the C standard requires that a zero byte always represent a null character, regardless of shift state. Also, a zero byte can't be the second (or later) byte of a multibyte character.

The C library provides two macros, `MB_LEN_MAX` and `MB_CUR_MAX`, that are related to multibyte characters. Both macros specify the maximum number of bytes in a multibyte character. `MB_LEN_MAX` (defined in `<limits.h>`) gives the maximum for any supported locale; `MB_CUR_MAX` (defined in `<stdlib.h>`) gives the maximum for the current locale. (Changing locales may affect the interpretation of multibyte characters.) Obviously, `MB_CUR_MAX` can't be larger than `MB_LEN_MAX`.

Any string may contain multibyte characters, although the length of such a string (as determined by the `strlen` function) is the number of bytes in the string, not the number of characters. In particular, the format strings in calls of the `...printf` and `...scanf` functions may contain multibyte characters. As a result, the C99 standard defines the term ***multibyte string*** to be a synonym for *string*.

C99

Wide Characters

The other way to encode an extended character set is to use wide characters. A ***wide character*** is an integer whose value represents a character. Unlike multibyte characters, which may vary in length, all wide characters supported by a particular implementation require the same number of bytes. A ***wide string*** is a string consisting of wide characters, with a null wide character at the end. (A ***null wide character*** is a wide character whose numerical value is zero.)

Wide characters have the type `wchar_t` (declared in `<stddef.h>` and certain other headers), which must be an integer type able to represent the largest extended character set for any supported locale. For example, if two bytes are enough to represent any extended character set, then `wchar_t` could be defined as `unsigned short int`.

C supports both wide character constants and wide string literals. Wide character constants resemble ordinary character constants but are prefixed by the letter `L`:

```
L'a'
```

Wide string literals are also prefixed by `L`:

```
L"abc"
```

This string represents an array containing the wide characters `L'a'`, `L'b'`, and `L'c'`, followed by a null wide character.

Unicode and the Universal Character Set

The differences between multibyte characters and wide characters become apparent when discussing ***Unicode***. Unicode is an enormous character set developed by the Unicode Consortium, an organization established by a group of computer manufacturers to create an international character set for computer use. The first 256 characters of Unicode are identical to Latin-1 (and therefore the first 128 characters of Unicode match the ASCII character set). However, Unicode goes far beyond Latin-1, providing the characters needed for nearly all modern and ancient languages. Unicode also includes a number of specialized symbols, such as those used in mathematics and music. The Unicode standard was first published in 1991.

Unicode is closely related to international standard ISO/IEC 10646, which defines a character encoding known as the ***Universal Character Set (UCS)***. UCS was developed by the International Organization for Standardization (ISO), starting at about the same time that Unicode was initially defined. Although UCS originally differed from Unicode, the two character sets were later unified. ISO now

works closely with the Unicode Consortium to ensure that ISO/IEC 10646 remains consistent with Unicode. Because Unicode and UCS are so similar, I'll use the two terms interchangeably.

Q&A

Unicode was originally limited to 65,536 characters (the number of characters that can be represented using 16 bits). That limit was later found to be insufficient; Unicode currently has over 100,000 characters. (For the most recent version, visit *www.unicode.org*.) The first 65,536 characters of Unicode—which include the most frequently used characters—are known as the ***Basic Multilingual Plane (BMP)***.

Encodings of Unicode

Unicode assigns a unique number (known as a ***code point***) to each character. There are a number of ways to represent these code points using bytes; I'll mention two of the simpler techniques. One of these encodings uses wide characters; the other uses multibyte characters.

UCS-2 is a wide-character encoding in which each Unicode code point is stored as two bytes. UCS-2 can represent all the characters in the Basic Multilingual Plane (those with code points between 0000 and FFFF in hexadecimal), but it is unable to represent Unicode characters that don't belong to the BMP.

A popular alternative is the ***8-bit UCS Transformation Format (UTF-8)***, which uses multibyte characters. UTF-8 was devised by Ken Thompson and his Bell Labs colleague Rob Pike in 1992. (Yes, that's the same Ken Thompson who designed the B language, the predecessor of C.) UTF-8 has the useful property that ASCII characters look identical in UTF-8: each character is one byte and has the same binary encoding. Thus, software designed to read UTF-8 data can also handle ASCII data with no change. For these reasons, UTF-8 is widely used on the Internet for text-based applications such as web pages and email.

In UTF-8, each code point requires between one and four bytes. UTF-8 is organized so that the most commonly used characters require fewer bytes, as shown in Table 25.7.

Table 25.7
UTF-8 Encoding

Code Point Range (Hexadecimal)	*UTF-8 Byte Sequence (Binary)*			
000000-00007F	0xxxxxxx			
000080-0007FF	110xxxxx	10xxxxxx		
000800-00FFFF	1110xxxx	10xxxxxx	10xxxxxx	
010000-10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

UTF-8 takes the bits in the code point value, divides them into groups (represented by the x's in Table 25.7), and assigns each group to a different byte. The simplest case is a code point in the range 0–7F (an ASCII character), which is represented by a 0 followed by the seven bits in the original number.

A code point in the range 80–7FF (which includes all the Latin-1 characters) would have its bits split into groups of five bits and six bits. The five-bit group is

prefixed by 110 and the six-bit group is prefixed by 10. For example, the code point for the character *ä* is E4 (hexadecimal) or 11100100 (binary). In UTF-8, it would be represented by the two-byte sequence 110<u>00011</u> 10<u>100100</u>. Note how the underlined portions, when joined together, spell out 00011100100.

Characters whose code points fall in the range 800–FFFF, which includes the remaining characters in the Basic Multilingual Plane, require three bytes. All other Unicode characters (most of them rarely used) are assigned four bytes.

UTF-8 has a number of useful properties:

- Each of the 128 ASCII characters is represented by one byte. A string consisting solely of ASCII characters looks exactly the same in UTF-8.
- Any byte in a UTF-8 string whose leftmost bit is 0 must be an ASCII character, because all other bytes begin with a 1 bit.
- The first byte of a multibyte character indicates how long the character will be. If the number of 1 bits at the beginning of the byte is two, the character is two bytes long. If the number of 1 bits is three or four, the character is three or four bytes long, respectively.
- Every other byte in a multibyte sequence has 10 as its leftmost bits.

The last three properties are especially important, because they guarantee that no sequence of bytes within a multibyte character can possibly represent another valid multibyte character. This makes it possible to search a multibyte string for a particular character or sequence of characters by simply doing byte comparisons.

So how does UTF-8 stack up against UCS-2? UCS-2 has the advantage that characters are stored in their most natural form. On the other hand, UTF-8 can handle all Unicode characters (not just those in the BMP), often requires less space than UCS-2, and retains compatibility with ASCII. UCS-2 isn't nearly as popular as UTF-8, although it was used in the Windows NT operating system. A newer version that uses four bytes (***UCS-4***) is gradually taking its place. Some systems extend UCS-2 into a multibyte encoding by allowing a variable number of byte pairs to represent a character (unlike UCS-2, which uses a single byte pair per character). This encoding, known as ***UTF-16,*** has the advantage that it's compatible with UCS-2.

Multibyte/Wide-Character Conversion Functions

```
int mblen(const char *s, size_t n);        from <stdlib.h>
int mbtowc(wchar_t * restrict pwc,
           const char * restrict s,
           size_t n);                      from <stdlib.h>
int wctomb(char *s, wchar_t wc);           from <stdlib.h>
```

Although the C89 standard introduced the concepts of multibyte characters and wide characters, it provides only five functions for working with these kinds of

characters. We'll now describe these functions, which belong to the `<stdlib.h>` header. C99's `<wchar.h>` and `<wctype.h>` headers, which are discussed in Sections 25.5 and 25.6, supply a number of additional multibyte and wide-character functions.

C89's multibyte/wide-character functions are divided into two groups. The first group converts single characters from multibyte form to wide form and vice versa. The behavior of these functions depends on the `LC_CTYPE` category of the current locale. If the multibyte encoding is state-dependent, the behavior also depends on the current ***conversion state.*** The conversion state consists of the current shift state as well as the current position within a multibyte character. Calling any of these functions with a null pointer as the value of its `char *` parameter sets the function's internal conversion state to the ***initial conversion state,*** signifying that no multibyte character is yet in progress and that the initial shift state is in effect. Later calls of the function cause its internal conversion state to be updated.

mblen

The `mblen` function checks whether its first argument points to a series of bytes that form a valid multibyte character. If so, the function returns the number of bytes in the character; if not, it returns –1. As a special case, `mblen` returns 0 if the first argument points to a null character. The second argument limits the number of bytes that `mblen` will examine; typically, we'll pass `MB_CUR_MAX`.

The following function, which comes from P. J. Plauger's *The Standard C Library*, uses `mblen` to determine whether a string consists of valid multibyte characters. The function returns zero if `s` points to a valid string.

```
int mbcheck(const char *s)
{
  int n;

  for (mblen(NULL, 0); ; s += n)
    if ((n = mblen(s, MB_CUR_MAX)) <= 0)
      return n;
}
```

Two aspects of the `mbcheck` function deserve special mention. First, there's the mysterious call `mblen(NULL, 0)`, which sets `mblen`'s internal conversion state to the initial conversion state (in case the multibyte encoding is state-dependent). Second, there's the matter of termination. Keep in mind that `s` points to an ordinary character string, which is assumed to end with a null character. `mblen` will return zero when it reaches this null character, causing `mbcheck` to return. `mbcheck` will return sooner if `mblen` returns –1 because of an invalid multibyte character.

mbtowc

The `mbtowc` function converts a multibyte character (pointed to by the second argument) into a wide character. The first argument points to a `wchar_t` variable into which the function will store the result. The third argument limits the number of bytes that `mbtowc` will examine. `mbtowc` returns the same value as `mblen`: the number of bytes in the multibyte character if it's valid, –1 if it's not, and zero if the second argument points to a null character.

wctomb

The `wctomb` function converts a wide character (the second argument) into a multibyte character, which it stores into the array pointed to by the first argument. `wctomb` may store as many as `MB_LEN_MAX` characters in the array, but doesn't append a null character. `wctomb` returns the number of bytes in the multibyte character or –1 if the wide character doesn't correspond to any valid multibyte character. (Note that `wctomb` returns 1 if asked to convert a null wide character.)

The following function (also from Plauger's *The Standard C Library*) uses `wctomb` to determine whether a string of wide characters can be converted to valid multibyte characters:

```
int wccheck(wchar_t *wcs)
{
  char buf[MB_LEN_MAX];
  int n;

  for (wctomb(NULL, 0); ; ++wcs)
    if ((n = wctomb(buf, *wcs)) <= 0)
      return -1;                /* invalid character */
    else if (buf[n-1] == '\0')
      return 0;                 /* all characters are valid */
}
```

Incidentally, all three functions—`mblen`, `mbtowc`, and `wctomb`—can be used to test whether a multibyte encoding is state-dependent. When passed a null pointer as its `char *` argument, each function returns a nonzero value if multibyte characters have state-dependent encodings or zero if they don't.

Multibyte/Wide-String Conversion Functions

```
size_t mbstowcs(wchar_t * restrict pwcs,
                const char * restrict s,
                size_t n);                  from <stdlib.h>
size_t wcstombs(char * restrict s,
                const wchar_t * restrict pwcs,
                size_t n);                  from <stdlib.h>
```

The remaining C89 multibyte/wide-character functions convert a string containing multibyte characters to a wide-character string and vice versa. How the conversion is performed depends on the `LC_CTYPE` category of the current locale.

mbstowcs

The `mbstowcs` function converts a sequence of multibyte characters into wide characters. The second argument points to an array containing the multibyte characters to be converted. The first argument points to a wide-character array; the third argument limits the number of wide characters that can be stored in the array. `mbstowcs` stops when it reaches the limit or encounters a null character (which it stores in the wide-character array). It returns the number of array elements modified, not including the terminating null wide character, if any. `mbstowcs` returns –1 (cast to type `size_t`) if it encounters an invalid multibyte character.

wcstombs

The `wcstombs` function is the opposite of `mbstowcs`: it converts a sequence of wide characters into multibyte characters. The second argument points to the wide-character string. The first argument points to the array in which the multibyte characters are to be stored. The third argument limits the number of bytes that can be stored in the array. `wcstombs` stops when it reaches the limit or encounters a null character (which it stores). It returns the number of bytes stored, not including the terminating null character, if any. `wcstombs` returns –1 (cast to type `size_t`) if it encounters a wide character that doesn't correspond to any multibyte character.

The `mbstowcs` function assumes that the string to be converted begins in the initial shift state. The string created by `wcstombs` always begins in the initial shift state.

25.3 Digraphs and Trigraphs

Programmers in certain countries have traditionally had trouble entering C programs because their keyboards lacked some of the characters that are required by C. This has been especially true in Europe, where older keyboards provided the accented characters used in European languages in place of the characters that C needs, such as `#`, `[`, `\`, `]`, `^`, `{`, `|`, `}`, and `~`. C89 introduced trigraphs—three-character codes that represent problematic characters—as a solution to this problem. Trigraphs proved to be unpopular, however, so Amendment 1 to the standard added two improvements: digraphs, which are more readable than trigraphs, and the `<iso646.h>` header, which defines macros that represent certain C operators.

Trigraphs

A ***trigraph sequence*** (or simply, a ***trigraph***) is a three-character code that can be used as an alternative to an ASCII character. Table 25.8 gives a complete list of trigraphs. All trigraphs begin with `??`, which makes them, if not exactly attractive, at least easy to spot.

Table 25.8
Trigraph Sequences

Trigraph Sequence	*ASCII Equivalent*
`??=`	`#`
`??(`	`[`
`??/`	`\`
`??)`	`]`
`??'`	`^`
`??<`	`{`
`??!`	`|`
`??>`	`}`
`??-`	`~`

Trigraphs can be freely substituted for their ASCII equivalents. For example, the program

```
#include <stdio.h>

int main(void)
{
  printf("hello, world\n");
  return 0;
}
```

could be written

```
??=include <stdio.h>

int main(void)
??<
  printf("hello, world??/n");
  return 0;
??>
```

Compilers that conform to the C89 or C99 standards are required to accept trigraphs, even though they're rarely used. Occasionally, this feature can cause problems.

⚠ Be careful about putting `??` in a string literal—it's possible that the compiler will treat it as the beginning of a trigraph. If this should happen, turn the second `?` character into an escape sequence by preceding it with a `\` character. The resulting `?\?` combination can't be mistaken for the beginning of a trigraph.

C99 Digraphs

Acknowledging that trigraphs are difficult to read, Amendment 1 to the C89 standard added an alternative notation known as ***digraphs.*** As the name implies, a digraph requires just two characters instead of three. Digraphs are available as substitutes for the six tokens shown in Table 25.9.

tokens ➤2.8

Table 25.9
Digraphs

Digraph	*Token*
`<:`	`[`
`:>`	`]`
`<%`	`{`
`%>`	`}`
`%:`	`#`
`%:%:`	`##`

Digraphs—unlike trigraphs—are *token* substitutes, not *character* substitutes. Thus, digraphs won't be recognized inside a string literal or character constant. For example, the string `"<::>"` has length four; it contains the characters: `<`, `:`, `:`,

and >, not the characters [and]. In contrast, the string "??(??)" has length two, because the compiler replaces the trigraph ??(by the character [and the trigraph ??) by the character].

Digraphs are more limited than trigraphs. First, as we've seen, digraphs are of no use inside a string literal or character constant; trigraphs are still needed in these situations. Second, digraphs don't solve the problem of providing alternate representations for the characters \, ^, |, and ~. The <iso646.h> header, described next, helps with this problem.

C99 The <iso646.h> Header: Alternative Spellings

The <iso646.h> header is quite simple. It contains nothing but the definitions of the eleven macros shown in Table 25.10. Each macro represents a C operator that contains one of the characters &, |, ~, !, or ^, making it possible to use the operators listed in the table even when these characters are absent from the keyboard.

Table 25.10 Macros Defined in <iso646.h>

Macro	*Value*
and	&&
and_eq	&=
bitand	&
bitor	|
compl	~
not	!
not_eq	!=
or	||
or_eq	|=
xor	^
xor_eq	^=

The name of the header comes from ISO/IEC 646, an older standard for an ASCII-like character set. This standard allows for "national variants," in which countries substitute local characters for certain ASCII characters, thereby causing the problem that digraphs and the <iso646.h> header are trying to solve.

25.4 Universal Character Names (C99)

Section 25.2 discussed the Universal Character Set (UCS), which is closely related to Unicode. C99 provides a special feature, ***universal character names,*** that allows us to use UCS characters in the source code of a program.

A universal character name resembles an escape sequence. However, unlike ordinary escape sequences, which can appear only in character constants and string literals, universal character names may also be used in identifiers. This feature allows programmers to use their native languages when defining names for variables, functions, and the like.

There are two ways to write a universal character name (\u*dddd* and \U*dddddddd*), where each *d* is a hexadecimal digit. In the form \U*dddddddd*, the *d*'s form an eight-digit hexadecimal number that identifies the UCS code point of the desired character. The form \u*dddd* can be used for characters whose code points have hexadecimal values of FFFF or less, which includes all characters in the Basic Multilingual Plane.

For example, the UCS code point for the Greek letter β is 000003B2, so the universal character name for this character is `\U000003B2` (or `\U000003b2`, since the case of hexadecimal digits doesn't matter). Because the first four hexadecimal digits of the UCS code point are 0, we can also use the `\u` notation, writing the character as `\u03B2` or `\u03b2`. The code point values for UCS (which match those for Unicode) can be found at *www.unicode.org/charts/*.

Not all universal character names may be used in identifiers; the C99 standard contains a list of which ones are allowed. Also, an identifier may not begin with a universal character name that represents a digit.

25.5 The <wchar.h> Header (C99) Extended Multibyte and Wide-Character Utilities

The `<wchar.h>` header provides functions for wide-character input/output and wide-string manipulation. The vast majority of functions in `<wchar.h>` are wide-character versions of functions from other headers (primarily `<stdio.h>` and `<string.h>`).

The `<wchar.h>` header declares several types and macros, including the following:

- `mbstate_t` — A value of this type can be used to store the conversion state when a sequence of multibyte characters is converted to a sequence of wide characters or vice versa.
- `wint_t` — An integer type whose values represent extended characters.
- `WEOF` — A macro representing a `wint_t` value that's different from any extended character. `WEOF` is used in much the same way as `EOF`, typically to indicate an error or end-of-file condition.

EOF macro ➤22.2

Note that `<wchar.h>` provides functions for wide characters but not multibyte characters. That's because C's ordinary library functions are capable of dealing with multibyte characters, so no special functions are needed. For example, the `fprintf` function allows its format string to contain multibyte characters.

Most wide-character functions behave the same as a function that belongs to another part of the standard library. Usually, the only changes involve arguments and return values of type `wchar_t` instead of `char` (or `wchar_t *` instead of `char *`). In addition, arguments and return values that represent character counts are measured in wide characters rather than bytes. In the remainder of this section, I'll indicate which other library function (if any) corresponds to each

wide-character function. I won't discuss the wide-character function further unless there's a significant difference between it and its "non-wide" counterpart.

Stream Orientation

Before we look at the input/output functions provided by `<wchar.h>`, it's important to understand ***stream orientation,*** a concept that doesn't exist in C89.

Every stream is either ***byte-oriented*** (the traditional orientation) or ***wide-oriented*** (data is written to the stream as wide characters). When a stream is first opened, it has no orientation. (In particular, the standard streams `stdin`, `stdout`, and `stderr` have no orientation at the beginning of program execution.) Performing an operation on the stream using a byte input/output function causes the stream to become byte-oriented; performing an operation using a wide-character input/output function causes the stream to become wide-oriented. The orientation of a stream can also be selected by calling the `fwide` function (described later in this section). A stream retains its orientation as long as it remains open. Calling the `freopen` function to reopen the stream will remove its orientation.

standard streams ➤22.1

freopen function ➤22.2

When wide characters are written to a wide-oriented stream, they are converted to multibyte characters before being stored in the file that is associated with the stream. Conversely, when input is read from a wide-oriented stream, the multibyte characters found in the stream are converted to wide characters. The multibyte encoding used in a file is similar to that used for characters and strings within a program, except that encodings used in files may contain embedded null bytes.

Each wide-oriented stream has an associated `mbstate_t` object, which keeps track of the stream's conversion state. An encoding error occurs when a wide character written to a stream doesn't correspond to any multibyte character, or when a sequence of characters read from a stream doesn't form a valid multibyte character. In either case, the value of the `EILSEQ` macro (defined in the `<errno.h>` header) is stored in the `errno` variable to indicate the nature of the error.

errno variable ➤24.2

Once a stream is byte-oriented, it's illegal to apply a wide-character input/output function to that stream. Similarly, it's illegal to apply a byte input/output function to a wide-oriented stream. Other stream functions may be applied to streams of either orientation, although there are a few special considerations for wide-oriented streams:

- Binary wide-oriented streams are subject to the file-positioning restrictions of both text and binary streams.
- After a file-positioning operation on a wide-oriented stream, a wide-character output function may end up overwriting part of a multibyte character. Doing so leaves the rest of the file in an indeterminate state.
- Calling `fgetpos` for a wide-oriented stream retrieves the stream's `mbstate_t` object as part of the `fpos_t` object associated with the stream. A later call of `fsetpos` using this `fpos_t` object will restore the `mbstate_t` object to its previous value.

fgetpos function ➤22.7

fsetpos function ➤22.7

Formatted Wide-Character Input/Output Functions

```
int fwprintf(FILE * restrict stream,
             const wchar_t * restrict format, ...);
int fwscanf(FILE * restrict stream,
            const wchar_t * restrict format, ...);
int swprintf(wchar_t * restrict s, size_t n,
             const wchar_t * restrict format, ...);
int swscanf(const wchar_t * restrict s,
            const wchar_t * restrict format, ...);
int vfwprintf(FILE * restrict stream,
              const wchar_t * restrict format,
              va_list arg);
int vfwscanf(FILE * restrict stream,
             const wchar_t * restrict format,
             va_list arg);
int vswprintf(wchar_t * restrict s, size_t n,
              const wchar_t * restrict format,
              va_list arg);
int vswscanf(const wchar_t * restrict s,
             const wchar_t * restrict format,
             va_list arg);
int vwprintf(const wchar_t * restrict format,
             va_list arg);
int vwscanf(const wchar_t * restrict format,
            va_list arg);
int wprintf(const wchar_t * restrict format, ...);
int wscanf(const wchar_t * restrict format, ...);
```

The functions in this group are wide-character versions of the formatted input/output functions found in `<stdio.h>` and described in Section 22.3. The `<wchar.h>` functions have arguments of type `wchar_t *` instead of `char *`, but their behavior is mostly the same as the `<stdio.h>` functions. Table 25.11 shows the correspondence between the `<stdio.h>` functions and their wide-character counterparts. Unless mentioned otherwise, each function in the left column behaves the same as the function(s) to its right.

All functions in this group share several characteristics:

- All have a format string, which consists of *wide* characters.
- ...`printf` functions, which return the number of characters written, now return the count in *wide* characters.
- The `%n` conversion specifier refers to the number of *wide* characters written so far (in the case of a ...`printf` function) or read so far (in the case of a ...`scanf` function).

Table 25.11
Formatted Wide-Character Input/Output Functions and Their `<stdio.h>` Equivalents

<wchar.h> Function	*<stdio.h> Equivalent*
fwprintf	fprintf
fwscanf	fscanf
swprintf	snprintf, sprintf
swscanf	sscanf
vfwprintf	vfprintf
vfwscanf	vfscanf
vswprintf	vsnprintf, vsprintf
vswscanf	vsscanf
vwprintf	vprintf
vwscanf	vscanf
wprintf	printf
wscanf	scanf

fwprintf Additional differences between `fwprintf` and `fprintf` include the following:

- The `%c` conversion specifier is used when the corresponding argument has type `int`. If the `l` length modifier is present (making the conversion `%lc`), the argument is assumed to have type `wint_t`. In either case, the corresponding argument is written as a wide character.
- The `%s` conversion specifier is used with a pointer to a character array, which may contain multibyte characters. (`fprintf` has no special provision for multibyte characters.) If the `l` length modifier is present, as in `%ls`, the corresponding argument should be an array containing wide characters. In either case, the characters in the array are written as wide characters. (With `fprintf`, the `%ls` specification also indicates an array of wide characters, but they're converted to multibyte characters before being written.)

fwscanf Unlike `fscanf`, the `fwscanf` function reads wide characters. The `%c`, `%s`, and `%[` conversions require special mention. Each of these causes wide characters to be read and then converted to multibyte characters before being stored in a character array. `fwscanf` uses an `mbstate_t` object to keep track of the state of the conversion during this process; the object is set to zero at the beginning of each conversion. If the `l` length modifier is present (making the conversion `%lc`, `%ls`, or `%l[`), then the input characters are not converted but instead are stored directly in an array of `wchar_t` elements. Thus, it's necessary to use `%ls` when reading a string of wide characters if the intent is to store them as wide characters. If `%s` is used instead, wide characters will be read from the input stream but converted to multibyte characters before being stored.

swprintf `swprintf` writes wide characters into an array of `wchar_t` elements. It's similar to `sprintf` and `snprintf` but not identical to either one. Like `snprintf`, it uses the parameter `n` to limit the number of (wide) characters that it will write. However, `swprintf` returns the number of wide characters actually written, not including the null character. In this respect, it resembles `sprintf` rather than `snprintf`, which returns the number of characters that would have been written (not including the null character) had there been no length restriction.

swprintf returns a negative value if the number of wide characters to be written is n or more, which differs from the behavior of both sprintf and snprintf.

vswprintf vswprintf is equivalent to swprintf, with arg replacing the variable argument list of swprintf. Like swprintf, which is similar—but not identical—to sprintf and snprintf, the vswprintf function is a combination of vsprintf and vsnprintf. If an attempt is made to write n or more wide characters, vswprintf returns a negative integer, in a manner similar to swprintf.

Wide-Character Input/Output Functions

```
wint_t fgetwc(FILE *stream);
wchar_t *fgetws(wchar_t * restrict s, int n,
                FILE * restrict stream);
wint_t fputwc(wchar_t c, FILE *stream);
int fputws(const wchar_t * restrict s,
           FILE * restrict stream);
int fwide(FILE *stream, int mode);
wint_t getwc(FILE *stream);
wint_t getwchar(void);
wint_t putwc(wchar_t c, FILE *stream);
wint_t putwchar(wchar_t c);
wint_t ungetwc(wint_t c, FILE *stream);
```

The functions in this group are wide-character versions of the character input/output functions found in <stdio.h> and described in Section 22.4. Table 25.12 shows the correspondence between the <stdio.h> functions and their wide-character counterparts. As the table shows, fwide is the only truly new function.

Table 25.12 Wide-Character Input/Output Functions and Their <stdio.h> Equivalents

<wchar.h> Function	*<stdio.h> Equivalent*
fgetwc	fgetc
fgetws	fgets
fputwc	fputc
fputws	fputs
fwide	–
getwc	getc
getwchar	getchar
putwc	putc
putwchar	putchar
ungetwc	ungetc

Unless otherwise indicated, you can assume that each <wchar.h> function listed in Table 25.12 behaves like the corresponding <stdio.h> function. However, one minor difference is common to most of these functions. To indicate an error or end-of-file condition, some <stdio.h> character I/O functions return EOF. The equivalent <wchar.h> functions return WEOF instead.

fgetwc
getwc
getwchar
fgetws

There's another twist that affects the wide-character input functions. A call of a function that reads a single character (`fgetwc`, `getwc`, and `getwchar`) may fail because the bytes found in the input stream don't form a valid wide character or there aren't enough bytes available. The result is an encoding error, which causes the function to store `EILSEQ` in `errno` and return `WEOF`. The `fgetws` function, which reads a string of wide characters, may also fail because of an encoding error, in which case it returns a null pointer.

fputwc
putwc
putwchar
fputws

Wide-character output functions may also encounter encoding errors. Functions that write a single character (`fputwc`, `putwc`, and `putwchar`) store `EILSEQ` in `errno` and return `WEOF` if an encoding error occurs. However, the `fputws` function, which writes a wide-character string, is different: it returns `EOF` (not `WEOF`) if an encoding error occurs.

fwide

The `fwide` function doesn't correspond to any C89 function. `fwide` is used to determine the current orientation of a stream and, if desired, attempt to set its orientation. The `mode` parameter determines the behavior of the function:

- `mode` > 0. Attempts to make the stream wide-oriented if it has no orientation.
- `mode` < 0. Attempts to make the stream byte-oriented if it has no orientation.
- `mode` = 0. The orientation is not changed.

`fwide` doesn't change the orientation if the stream already has one.

The value returned by `fwide` depends on the orientation of the stream *after* the call. The return value is positive if the stream has wide orientation, negative if it has byte orientation, and zero if it has no orientation.

General Wide-String Utilities

The `<wchar.h>` header provides a number of functions that perform operations on wide strings. These are wide-character versions of functions that belong to the `<stdlib.h>` and `<string.h>` headers.

Wide-String Numeric Conversion Functions

```
double wcstod(const wchar_t * restrict nptr,
              wchar_t ** restrict endptr);
float wcstof(const wchar_t * restrict nptr,
             wchar_t ** restrict endptr);
long double wcstold(const wchar_t * restrict nptr,
                    wchar_t ** restrict endptr);
long int wcstol(const wchar_t * restrict nptr,
                wchar_t ** restrict endptr,
                int base);
long long int wcstoll(const wchar_t * restrict nptr,
                      wchar_t ** restrict endptr,
                      int base);
```

```
unsigned long int wcstoul(
                        const wchar_t * restrict nptr,
                        wchar_t ** restrict endptr,
                        int base);
unsigned long long int wcstoull(
                        const wchar_t * restrict nptr,
                        wchar_t ** restrict endptr,
                        int base);
```

The functions in this group are wide-character versions of the numeric conversion functions found in `<stdlib.h>` and described in Section 26.2. The `<wchar.h>` functions have arguments of type `wchar_t *` and `wchar_t **` instead of `char *` and `char **`, but their behavior is mostly the same as the `<stdlib.h>` functions. Table 25.13 shows the correspondence between the `<stdlib.h>` functions and their wide-character counterparts.

Table 25.13 Wide-String Numeric Conversion Functions and Their `<stdlib.h>` Equivalents

<wchar.h> Function	*<stdlib.h> Equivalent*
wcstod	strtod
wcstof	strtof
wcstold	strtold
wcstol	strtol
wcstoll	strtoll
wcstoul	strtoul
wcstoull	strtoull

Wide-String Copying Functions

```
wchar_t *wcscpy(wchar_t * restrict s1,
                const wchar_t * restrict s2);
wchar_t *wcsncpy(wchar_t * restrict s1,
                 const wchar_t * restrict s2,
                 size_t n);
wchar_t *wmemcpy(wchar_t * restrict s1,
                 const wchar_t * restrict s2,
                 size_t n);
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2,
                  size_t n);
```

The functions in this group are wide-character versions of the string copying functions found in `<string.h>` and described in Section 23.6. The `<wchar.h>` functions have arguments of type `wchar_t *` instead of `char *`, but their behavior is mostly the same as the `<string.h>` functions. Table 25.14 shows the correspondence between the `<string.h>` functions and their wide-character counterparts.

Table 25.14 Wide-String Copying Functions and Their <string.h> Equivalents

<wchar.h> Function	<string.h> Equivalent
wcscpy	strcpy
wcsncpy	strncpy
wmemcpy	memcpy
wmemmove	memmove

Wide-String Concatenation Functions

```
wchar_t *wcscat(wchar_t * restrict s1,
                const wchar_t * restrict s2);
wchar_t *wcsncat(wchar_t * restrict s1,
                 const wchar_t * restrict s2,
                 size_t n);
```

The functions in this group are wide-character versions of the string concatenation functions found in <string.h> and described in Section 23.6. The <wchar.h> functions have arguments of type wchar_t * instead of char *, but their behavior is mostly the same as the <string.h> functions. Table 25.15 shows the correspondence between the <string.h> functions and their wide-character counterparts.

Table 25.15 Wide-String Concatenation Functions and Their <string.h> Equivalents

<wchar.h> Function	<string.h> Equivalent
wcscat	strcat
wcsncat	strncat

Wide-String Comparison Functions

```
int wcscmp(const wchar_t *s1, const wchar_t *s2);
int wcscoll(const wchar_t *s1, const wchar_t *s2);
int wcsncmp(const wchar_t *s1, const wchar_t *s2,
            size_t n);
size_t wcsxfrm(wchar_t * restrict s1,
               const wchar_t * restrict s2,
               size_t n);
int wmemcmp(const wchar_t * s1, const wchar_t * s2,
            size_t n);
```

The functions in this group are wide-character versions of the string comparison functions found in <string.h> and described in Section 23.6. The <wchar.h> functions have arguments of type wchar_t * instead of char *, but their behavior is mostly the same as the <string.h> functions. Table 25.16 shows the correspondence between the <string.h> functions and their wide-character counterparts.

Table 25.16 Wide-String Comparison Functions and Their <string.h> Equivalents

<wchar.h> Function	<string.h> Equivalent
wcscmp	strcmp
wcscoll	strcoll
wcsncmp	strncmp
wcsxfrm	strxfrm
wmemcmp	memcmp

Wide-String Search Functions

```
wchar_t *wcschr(const wchar_t *s, wchar_t c);
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcspbrk(const wchar_t *s1,
                 const wchar_t *s2);
wchar_t *wcsrchr(const wchar_t *s, wchar_t c);
size_t wcsspn(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcsstr(const wchar_t *s1,
                const wchar_t *s2);
wchar_t *wcstok(wchar_t * restrict s1,
                const wchar_t * restrict s2,
                wchar_t ** restrict ptr);
wchar_t *wmemchr(const wchar_t *s, wchar_t c,
                 size_t n);
```

The functions in this group are wide-character versions of the string search functions found in <string.h> and described in Section 23.6. The <wchar.h> functions have arguments of type wchar_t * and wchar_t ** instead of char * and char **, but their behavior is mostly the same as the <string.h> functions. Table 25.17 shows the correspondence between the <string.h> functions and their wide-character counterparts.

Table 25.17 Wide-String Search Functions and Their <string.h> Equivalents

<wchar.h> Function	<string.h> Equivalent
wcschr	strchr
wcscspn	strcspn
wcspbrk	strpbrk
wcsrchr	strrchr
wcsspn	strspn
wcsstr	strstr
wcstok	strtok
wmemchr	memchr

wcstok The wcstok function serves the same purpose as strtok, but is used somewhat differently, thanks to its third parameter. (strtok has only two parameters.) To understand how wcstok works, we'll first need to review the behavior of strtok.

We saw in Section 23.6 that `strtok` searches a string for a "token"—a sequence of characters that doesn't include certain delimiting characters. The call `strtok(s1, s2)` scans the `s1` string for a nonempty sequence of characters that are *not* in the `s2` string. `strtok` marks the end of the token by storing a null character in `s1` just after the last character in the token; it then returns a pointer to the first character in the token.

Later calls of `strtok` can find additional tokens in the same string. The call `strtok(NULL, s2)` continues the search begun by the previous `strtok` call. As before, `strtok` marks the end of the token with a null character, and then returns a pointer to the beginning of the token. The process can be repeated until `strtok` returns a null pointer, indicating that no token was found.

One problem with `strtok` is that it uses a static variable to keep track of a search, which makes it impossible to use `strtok` to conduct simultaneous searches on two or more strings. Thanks to its extra parameter, `wcstok` doesn't have this problem.

The first two parameters to `wcstok` are the same as for `strtok` (except that they point to wide strings, of course). The third parameter, `ptr`, will point to a variable of type `wchar_t *`. The function will save information in this variable that enables later calls of `wcstok` to continue scanning the same string (when the first argument is a null pointer). When the search is resumed by a subsequent call of `wcstok`, a pointer to the same variable should be supplied as the third argument; the value of this variable must not be changed between calls of `wcstok`.

To see how `wcstok` works, let's redo the example of Section 23.6. Assume that `str`, `p`, and `q` are declared as follows:

```
wchar_t str[] = L" April  28,1998";
wchar_t *p, *q;
```

Our initial call of `wcstok` will pass `str` as the first argument:

```
p = wcstok(str, L" \t", &q);
```

`p` now points to the first character in `April`, which is followed by a null wide character. Calling `wcstok` with a null pointer as its first argument and `&q` as the third argument causes it to resume the search from where it left off:

```
p = wcstok(NULL, L" \t,", &q);
```

After this call, `p` points to the first character in `28`, which is now terminated by a null wide character. A final call of `wcstok` locates the year:

```
p = wcstok(NULL, L" \t", &q);
```

`p` now points to the first character in `1998`.

Miscellaneous Functions

```
size_t wcslen(const wchar_t *s);
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
```

The functions in this group are wide-character versions of the miscellaneous string functions found in <string.h> and described in Section 23.6. The <wchar.h> functions have arguments of type wchar_t * instead of char *, but their behavior is mostly the same as the <string.h> functions. Table 25.18 shows the correspondence between the <string.h> functions and their wide-character counterparts.

Table 25.18 Miscellaneous Wide-String Functions and Their <string.h> Equivalents

<wchar.h> Function	*<string.h> Equivalent*
wcslen	strlen
wmemset	memset

Wide-Character Time-Conversion Functions

```
size_t wcsftime(wchar_t * restrict s, size_t maxsize,
                const wchar_t * restrict format,
                const struct tm * restrict timeptr);
```

wcsftime The wcsftime function is the wide-character version of strftime, which belongs to the <time.h> header and is described in Section 26.3.

Extended Multibyte/Wide-Character Conversion Utilities

We'll now examine <wchar.h> functions that perform conversions between multibyte characters and wide characters. Five of these functions (mbrlen, mbrtowc, wcrtomb, mbsrtowcs, and wcsrtombs) correspond to the multibyte/wide-character and multibyte/wide-string conversion functions declared in <stdlib.h>. The <wchar.h> functions have an additional parameter, a pointer to a variable of type mbstate_t. This variable keeps track of the state of the conversion of a multibyte character sequence to a wide-character sequence (or vice versa), based on the current locale. As a result, the <wchar.h> functions are "restartable"; by passing a pointer to an mbstate_t variable modified by a previous function call, we can "restart" the function using the conversion state from that call. One advantage of this arrangement is that it allows two functions to share the same conversion state. For example, calls of mbrtowc and mbsrtowcs that are used to process a single multibyte character string could share an mbstate_t variable.

The conversion state stored in an mbstate_t variable consists of the current shift state plus the current position within a multibyte character. Setting the bytes of an mbstate_t variable to zero puts it in the initial conversion state, signifying that no multibyte character is yet in progress and that the initial shift state is in effect:

```
mbstate_t state;
...
memset(&state, '\0', sizeof(state));
```

Passing `&state` to one of the restartable functions causes the conversion to begin in the initial conversion state. Once an `mbstate_t` variable has been altered by one of these functions, it should not be used to convert a different multibyte character sequence, nor should it be used to perform a conversion in the opposite direction. Attempting to perform either action causes undefined behavior. Using the variable after a change in the `LC_CTYPE` category of a locale also causes undefined behavior.

Single-Byte/Wide-Character Conversion Functions

```
wint_t btowc(int c);
int wctob(wint_t c);
```

The functions in this group convert single-byte characters to wide characters and vice versa.

btowc The `btowc` function returns `WEOF` if `c` is equal to `EOF` or if `c` (when cast to `unsigned char`) isn't a valid single-byte character in the initial shift state. Otherwise, `btowc` returns the wide-character representation of `c`.

wctob The `wctob` function is the opposite of `btowc`. It returns `EOF` if `c` doesn't correspond to one multibyte character in the initial shift state. Otherwise, it returns the single-byte representation of `c`.

Conversion-State Functions

```
int mbsinit(const mbstate_t *ps);
```

mbsinit This group consists of a single function, `mbsinit`, which returns a nonzero value if `ps` is a null pointer or it points to an `mbstate_t` variable that describes an initial conversion state.

Restartable Multibyte/Wide-Character Conversion Functions

```
size_t mbrlen(const char * restrict s, size_t n,
              mbstate_t * restrict ps);
size_t mbrtowc(wchar_t * restrict pwc,
               const char * restrict s, size_t n,
               mbstate_t * restrict ps);
size_t wcrtomb(char * restrict s, wchar_t wc,
               mbstate_t * restrict ps);
```

The functions in this group are restartable versions of the `mblen`, `mbtowc`, and `wctomb` functions, which belong to `<stdlib.h>` and are discussed in Section 25.2. The newer `mbrlen`, `mbrtowc`, and `wcrtomb` functions differ from their `<stdlib.h>` counterparts in several ways:

- mbrlen, mbrtowc, and wcrtomb have an additional parameter named ps. When one of these functions is called, the corresponding argument should point to a variable of type mbstate_t; the function will store the state of the conversion in this variable. If the argument corresponding to ps is a null pointer, the function will use an internal variable to store the conversion state. (At the beginning of program execution, this variable is set to the initial conversion state.)
- When the s parameter is a null pointer, the older mblen, mbtowc, and wctomb functions return a nonzero value if multibyte character encodings have state-dependent encodings (and zero otherwise). The newer functions don't have this behavior.
- mbrlen, mbrtowc, and wcrtomb return a value of type size_t instead of int, the return type of the older functions.

mbrlen A call of mbrlen is equivalent to the call

```
mbrtowc(NULL, s, n, ps)
```

except that if ps is a null pointer, then the address of an internal variable is used instead.

mbrtowc If s is a null pointer, a call of mbrtowc is equivalent to the call

```
mbrtowc(NULL, "", 1, ps)
```

Otherwise, a call of mbrtowc examines up to n bytes pointed to by s to see if they complete a valid multibyte character. (Note that a multibyte character may already be in progress prior to the call, as tracked by the mbstate_t variable to which ps points.) If so, these bytes are converted into a wide character. The wide character is stored in the location pointed to by pwc as long as pwc isn't null. If this character is the null wide character, the mbstate_t variable used during the call is left in the initial conversion state.

mbrtowc has a variety of possible return values. It returns 0 if the conversion produces a null wide character. It returns a number between 1 and n if the conversion produces a wide character other than null, where the value returned is the number of bytes used to complete the multibyte character. It returns –2 if the n bytes pointed to by s aren't enough to complete a multibyte character (although the bytes themselves were valid). Finally, it returns –1 if an encoding error occurs (the function encounters bytes that don't form a valid multibyte character). In the last case, mbrtowc also stores EILSEQ in errno.

wcrtomb If s is a null pointer, a call of wcrtomb is equivalent to

```
wcrtomb(buf, L'\0', ps)
```

where buf is an internal buffer. Otherwise, wcrtomb converts wc from a wide character into a multibyte character, which it stores in the array pointed to by s. If wc is a null wide character, wcrtomb stores a null byte, preceded by a shift sequence if one is necessary to restore the initial shift state. In this case, the

mbstate_t variable used during the call is left in the initial conversion state. wcrtomb returns the number of bytes that it stores, including shift sequences. If wc isn't a valid wide character, the function returns –1 and stores EILSEQ in errno.

Restartable Multibyte/Wide-String Conversion Functions

```
size_t mbsrtowcs(wchar_t * restrict dst,
                 const char ** restrict src,
                 size_t len,
                 mbstate_t * restrict ps);
size_t wcsrtombs(char * restrict dst,
                 const wchar_t ** restrict src,
                 size_t len,
                 mbstate_t * restrict ps);
```

mbsrtowcs
wcsrtombs

The mbsrtowcs and wcsrtombs functions are restartable versions of mbstowcs and wcstombs, which belong to <stdlib.h> and are discussed in Section 25.2. mbsrtowcs and wcsrtombs are the same as their <stdlib.h> counterparts, except for the following differences:

- mbsrtowcs and wcsrtombs have an additional parameter named ps. When one of these functions is called, the corresponding argument should point to a variable of type mbstate_t; the function will store the state of the conversion in this variable. If the argument corresponding to ps is a null pointer, the function will use an internal variable to store the conversion state. (At the beginning of program execution, this variable is set to the initial conversion state.) Both functions update the state as the conversion proceeds. If the conversion stops because a null character is reached, the mbstate_t variable will be left in the initial conversion state.
- The src parameter, which represents the array containing characters to be converted (the source array), is a pointer to a pointer for mbsrtowcs and wcsrtombs. (In the older mbstowcs and wcstombs functions, the corresponding parameter was simply a pointer.) This change allows mbsrtowcs and wcsrtombs to keep track of where the conversion stopped. The pointer to which src points is set to null if the conversion stopped because a null character was reached. Otherwise, this pointer is set to point just past the last source character converted.
- The dst parameter may be a null pointer, in which case the converted characters aren't stored and the pointer to which src points isn't modified.
- When either function encounters an invalid character in the source array, it stores EILSEQ in errno (in addition to returning –1, as the older functions do).

25.6 The <wctype.h> Header (C99) Wide-Character Classification and Mapping Utilities

<ctype.h> header ➤23.5

The <wctype.h> header is the wide-character version of the <ctype.h> header. <ctype.h> provides two kinds of functions: character-classification functions (like isdigit, which tests whether a character is a digit) and character case-mapping functions (like toupper, which converts a lower-case letter to upper case). <wctype.h> provides similar functions for wide characters, although it differs from <ctype.h> in one important way: some of the functions in <wctype.h> are "extensible," meaning that they can perform custom character classification or case mapping.

<wctype.h> declares three types and a macro. The wint_t type and the WEOF macro were discussed in Section 25.5. The remaining types are wctype_t, whose values represent locale-specific character classifications, and wctrans_t, whose values represent locale-specific character mappings.

Most of the functions in <wctype.h> require a wint_t argument. The value of this argument must be a wide character (a wchar_t value) or WEOF. Passing any other argument causes undefined behavior.

The behavior of the functions in <wctype.h> is affected by the LC_CTYPE category of the current locale.

Wide-Character Classification Functions

```
int iswalnum(wint_t wc);
int iswalpha(wint_t wc);
int iswblank(wint_t wc);
int iswcntrl(wint_t wc);
int iswdigit(wint_t wc);
int iswgraph(wint_t wc);
int iswlower(wint_t wc);
int iswprint(wint_t wc);
int iswpunct(wint_t wc);
int iswspace(wint_t wc);
int iswupper(wint_t wc);
int iswxdigit(wint_t wc);
```

Each wide-character classification function returns a nonzero value if its argument has a particular property. Table 25.19 lists the property that each function tests.

The descriptions in Table 25.19 ignore some of the subtleties of wide characters. For example, the definition of iswgraph in the C99 standard states that it "tests for any wide character for which iswprint is true and iswspace is false,"

Table 25.19
Wide-Character Classification Functions

Function	*Test*
`iswalnum(wc)`	Is wc alphanumeric?
`iswalpha(wc)`	Is wc alphabetic?
`iswblank(wc)`	Is wc a blank?†
`iswcntrl(wc)`	Is wc a control character?
`iswdigit(wc)`	Is wc a decimal digit?
`iswgraph(wc)`	Is wc a printing character (other than a space)?
`iswlower(wc)`	Is wc a lower-case letter?
`iswprint(wc)`	Is wc a printing character (including a space)?
`iswpunct(wc)`	Is wc punctuation?
`iswspace(wc)`	Is wc a white-space character?
`iswupper(wc)`	Is wc an upper-case letter?
`iswxdigit(wc)`	Is wc a hexadecimal digit?

†The standard blank wide characters are space (`L' '`) and horizontal tab (`L'\t'`).

leaving open the possibility that more than one wide character is considered to be a "space." See Appendix D for more detailed descriptions of these functions.

In most cases, the wide-character classification functions are consistent with the corresponding functions in `<ctype.h>`: if a `<ctype.h>` function returns a nonzero value (indicating "true") for a particular character, then the corresponding `<wctype.h>` function will return true for the wide version of the same character. The only exception involves white-space wide characters (other than space) that are also printing characters, which may be classified differently by `iswgraph` and `iswpunct` than by `isgraph` and `ispunct`. For example, a character for which `isgraph` returns true may cause `iswgraph` to return false.

Extensible Wide-Character Classification Functions

```
int iswctype(wint_t wc, wctype_t desc);
wctype_t wctype(const char *property);
```

Each of the wide-character classification functions just discussed is able to test a single fixed condition. The `wctype` and `iswctype` functions—which are designed to be used together—make it possible to test for other conditions as well.

wctype

The `wctype` function is passed a string describing a class of wide characters; it returns a `wctype_t` value that represents this class. For example, the call

```
wctype("upper")
```

returns a `wctype_t` value representing the class of upper-case letters. The C99 standard requires that the following strings be allowed as arguments to `wctype`:

```
"alnum"  "alpha"  "blank"  "cntrl"  "digit"  "graph"
"lower"  "print"  "punct"  "space"  "upper"  "xdigit"
```

Additional strings may be provided by an implementation. Which strings are legal arguments to `wctype` at a given time depends on the `LC_CTYPE` category of the

current locale; the 12 strings listed above are legal in all locales. If `wctype` is passed a string that's not supported in the current locale, it returns zero.

iswctype

A call of the `iswctype` function requires two parameters: `wc` (a wide character) and `desc` (a value returned by `wctype`). `iswctype` returns a nonzero value if `wc` belongs to the class of characters corresponding to `desc`. For example, the call

```
iswctype(wc, wctype("alnum"))
```

is equivalent to

```
iswalnum(wc)
```

`wctype` and `iswctype` are most useful when the argument to `wctype` is a string other than the standard ones listed above.

Wide-Character Case-Mapping Functions

```
wint_t towlower(wint_t wc);
wint_t towupper(wint_t wc);
```

towlower
towupper

The `towlower` and `towupper` functions are the wide-character counterparts of `tolower` and `toupper`. For example, `towlower` returns the lower-case version of its argument, if the argument is an upper-case letter; otherwise, it returns the argument unchanged. As usual, there may be quirks when dealing with wide characters. For example, more than one lower-case version of a letter may exist in the current locale, in which case `towlower` is allowed to return any one of them.

Extensible Wide-Character Case-Mapping Functions

```
wint_t towctrans(wint_t wc, wctrans_t desc);
wctrans_t wctrans(const char *property);
```

The `wctrans` and `towctrans` functions are used together to support generalized wide-character mapping.

wctrans

The `wctrans` function is passed a string describing a character mapping; it returns a `wctrans_t` value that represents the mapping. For example, the call

```
wctrans("tolower")
```

returns a `wctrans_t` value representing the mapping of upper-case letters to lower case. The C99 standard requires that the strings `"tolower"` and `"toupper"` be allowed as arguments to `wctrans`. Additional strings may be provided by an implementation. Which strings are legal arguments to `wctrans` at a given time depends on the `LC_CTYPE` category of the current locale; `"tolower"` and `"toupper"` are legal in all locales. If `wctrans` is passed a string that's not supported in the current locale, it returns zero.

towctrans A call of the `towctrans` function requires two parameters: `wc` (a wide character) and `desc` (a value returned by `wctrans`). `towctrans` maps `wc` to another wide character based on the mapping specified by `desc`. For example, the call

```
towctrans(wc, wctrans("tolower"))
```

is equivalent to

```
towlower(wc)
```

`towctrans` is most useful in conjunction with implementation-defined mappings.

Q & A

Q: How long is the locale information string returned by `setlocale`? [p. 644]

A: There's no maximum length, which raises a question: how can we set aside space for the string if we don't know how long it will be? The answer, of course, is dynamic storage allocation. The following program fragment (based on a similar example in Harbison and Steele's *C: A Reference Manual*) shows how to determine the amount of memory needed, allocate the memory dynamically, and then copy the locale information into that memory:

```
char *temp, *old_locale;

temp = setlocale(LC_ALL, NULL);
if (temp == NULL) {
  /* locale information not available */
}
old_locale = malloc(strlen(temp) + 1);
if (old_locale == NULL) {
  /* memory allocation failed */
}
strcpy(old_locale, temp);
```

We can now switch to a different locale and then later restore the old locale:

```
setlocale(LC_ALL, "");          /* switches to native locale */
...
setlocale(LC_ALL, old_locale); /* restores old locale */
```

Q: Why does C provide both multibyte characters and wide characters? Wouldn't either one be enough by itself? [p. 648]

A: The two encodings serve different purposes. Multibyte characters are handy for input/output purposes, since I/O devices are often byte-oriented. Wide characters, on the other hand, are more convenient to work with inside a program, since every wide character occupies the same amount of space. Thus, a program might

read multibyte characters, convert them to wide characters for manipulation within the program, and then convert the wide characters back to multibyte form for output.

Q: Unicode and UCS seem to be pretty much the same. What's the difference between the two? [p. 650]

A: Both contain the same characters, and characters are represented by the same code points in both. Unicode is more than just a character set, though. For example, Unicode supports "bidirectional display order." Some languages, including Arabic and Hebrew, allow text to be written from right to left instead of left to right. Unicode is capable of specifying the display order of characters, allowing text to contain some characters that are to be displayed from left to right along with others that go from right to left.

Exercises

Section 25.1

1. Determine which locales are supported by your compiler.

Section 25.2

2. The Shift-JIS encoding for *kanji* requires either one or two bytes per character. If the first byte of a character is between `0x81` and `0x9f` or between `0xe0` and `0xef`, a second byte is required. (Any other byte is treated as a whole character.) The second byte must be between `0x40` and `0x7e` or between `0x80` and `0xfc`. (All ranges are inclusive.) For each of the following strings, give the value that the `mbcheck` function of Section 25.2 will return when passed that string as its argument, assuming that multibyte characters are encoded using Shift-JIS in the current locale.
   (a) `"\x05\x87\x80\x36\xed\xaa"`
   (b) `"\x20\xe4\x50\x88\x3f"`
   (c) `"\xde\xad\xbe\xef"`
   (d) `"\x8a\x60\x92\x74\x41"`

3. One of the useful properties of UTF-8 is that no sequence of bytes within a multibyte character can possibly represent another valid multibyte character. Does the Shift-JIS encoding for *kanji* (discussed in Exercise 2) have this property?

4. Give a C string literal that represents each of the following phrases. Assume that the characters à, è, é, ê, î, ô, û, and ü are represented by single-byte Latin-1 characters. (You'll need to look up the Latin-1 code points for these characters.) For example, the phrase *déjà vu* could be represented by the string `"d\xe9j\xe0 vu"`.
   (a) *Côte d'Azur*
   (b) *crème brûlée*
   (c) *crème fraîche*
   (d) *Fahrvergnügen*
   (e) *tête-à-tête*

5. Repeat Exercise 4, this time using the UTF-8 multibyte encoding. For example, the phrase *déjà vu* could be represented by the string `"d\xc3\xa9j\xc3\xa0 vu"`.

Section 25.3 W 6. Modify the following program fragment by replacing as many characters as possible by trigraphs.

```
while ((orig_char = getchar()) != EOF) {
  new_char = orig_char ^ KEY;
  if (isprint(orig_char) && isprint(new_char))
    putchar(new_char);
  else
    putchar(orig_char);
}
```

7. (C99) Modify the program fragment in Exercise 6 by replacing as many tokens as possible by digraphs and macros defined in `<iso646.h>`.

Programming Projects

W 1. Write a program that tests whether your compiler's `""` (native) locale is the same as its `"C"` locale.

2. Write a program that obtains the name of a locale from the command line and then displays the values stored in the corresponding `lconv` structure. For example, if the locale is `"fi_FI"` (Finland), the output of the program might look like this:

```
decimal_point = ","
thousands_sep = " "
grouping = 3
mon_decimal_point = ","
mon_thousands_sep = " "
mon_grouping = 3
positive_sign = ""
negative_sign = "-"
currency_symbol = "EUR"
frac_digits = 2
p_cs_precedes = 0
n_cs_precedes = 0
p_sep_by_space = 2
n_sep_by_space = 2
p_sign_posn = 1
n_sign_posn = 1
int_curr_symbol = "EUR "
int_frac_digits = 2
int_p_cs_precedes = 0
int_n_cs_precedes = 0
int_p_sep_by_space = 2
int_n_sep_by_space = 2
int_p_sign_posn = 1
int_n_sign_posn = 1
```

For readability, the characters in `grouping` and `mon_grouping` should be displayed as decimal numbers.

26 Miscellaneous Library Functions

It is the user who should parametrize procedures, not their creators.

<stdarg.h>, <stdlib.h>, and <time.h>—the only C89 headers that weren't covered in previous chapters—are unlike any others in the standard library. The <stdarg.h> header (Section 26.1) makes it possible to write functions with a variable number of arguments. <stdlib.h> (Section 26.2) is an assortment of functions that don't fit into one of the other headers. The <time.h> header (Section 26.3) allows programs to work with dates and times.

26.1 The <stdarg.h> Header: Variable Arguments

```
type va_arg(va_list ap, type);
void va_copy(va_list dest, va_list src);
void va_end(va_list ap);
void va_start(va_list ap, parmN);
```

Functions such as printf and scanf have an unusual property: they allow any number of arguments. The ability to handle a variable number of arguments isn't limited to library functions, as it turns out. The <stdarg.h> header provides the tools we'll need to write our own functions with variable-length argument lists. <stdarg.h> declares one type (va_list) and defines several macros. In C89, there are three macros, named va_start, va_arg, and va_end, which can be thought of as functions with the prototypes shown above. C99 adds a function-like macro named va_copy.

C99

To see how these macros work, we'll use them to write a function named `max_int` that finds the maximum of *any* number of integer arguments. Here's how we might call the function:

```
max_int(3, 10, 30, 20)
```

The first argument specifies how many additional arguments will follow. This call of `max_int` will return 30 (the largest of the numbers 10, 30, and 20).

Here's the definition of the `max_int` function:

```
int max_int(int n, ...)   /* n must be at least 1 */
{
  va_list ap;
  int i, current, largest;

  va_start(ap, n);
  largest = va_arg(ap, int);

  for (i = 1; i < n; i++) {
    current = va_arg(ap, int);
    if (current > largest)
      largest = current;
  }

  va_end(ap);
  return largest;
}
```

The `...` symbol in the parameter list (known as an ***ellipsis***) indicates that the parameter `n` is followed by a variable number of additional parameters.

The body of `max_int` begins with the declaration of a variable of type `va_list`:

```
va_list ap;
```

Declaring such a variable is mandatory for `max_int` to be able to access the arguments that follow `n`.

va_start

The statement

```
va_start(ap, n);
```

indicates where the variable-length part of the argument list begins (in this case, after `n`). A function with a variable number of arguments must have at least one "normal" parameter; the ellipsis always goes at the end of the parameter list, after the last normal parameter.

va_arg

The statement

```
largest = va_arg(ap, int);
```

fetches `max_int`'s second argument (the one after `n`), assigns it to `largest`, and automatically advances to the next argument. The word `int` indicates that we expect `max_int`'s second argument to have `int` type. The statement

```
current = va_arg(ap, int);
```

fetches max_int's remaining arguments, one by one, as it is executed inside a loop.

Don't forget that va_arg always advances to the next argument after fetching the current one. Because of this property, we couldn't have written max_int's loop in the following way:

```
for (i = 1; i < n; i++)
  if (va_arg(ap, int) > largest)    /*** WRONG ***/
    largest = va_arg(ap, int);
```

---

va_end

The statement

```
va_end(ap);
```

is required to "clean up" before the function returns. (Or, instead of returning, the function might call va_start and traverse the argument list again.)

va_copy

The va_copy macro copies src (a va_list value) into dest (also a va_list). The usefulness of va_copy lies in the fact that multiple calls of va_arg may have been made using src before it's copied into dest, thus processing some of the arguments. Calling va_copy allows a function to remember where it is in the argument list so that it can later return to the same point to reexamine an argument (and possibly the arguments that follow it).

Each call of va_start or va_copy must be paired with a call of va_end, and the calls must appear in the same function. All calls of va_arg must appear between the call of va_start (or va_copy) and the matching call of va_end.

---

default argument promotions ➤9.3

When a function with a variable argument list is called, the compiler performs the default argument promotions on all arguments that match the ellipsis. In particular, char and short arguments are promoted to int, and float values are promoted to double. Consequently, it doesn't make sense to pass types such as char, short, or float to va_arg, since arguments—after promotion—will never have one of those types.

Calling a Function with a Variable Argument List

Calling a function with a variable argument list is an inherently risky proposition. As far back as Chapter 3, we saw how dangerous it can be to pass the wrong arguments to printf and scanf. Other functions with variable argument lists are equally sensitive. The primary difficulty is that a function with a variable argument list has no way to determine the number of arguments or their types. This information must be passed into the function and/or assumed by the function. max_int relies on the first argument to specify how many additional arguments follow; it

assumes that the arguments are of type `int`. Functions such as `printf` and `scanf` rely on the format string, which describes the number of additional arguments and the type of each.

Another problem has to do with passing `NULL` as an argument. `NULL` is usually defined to represent `0`. When `0` is passed to a function with a variable argument list, the compiler assumes that it represents an integer—there's no way it can tell that we want it to represent the null pointer. The solution is to add a cast, writing `(void *) NULL` or `(void *) 0` instead of `NULL`. (See the Q&A section at the end of Chapter 17 for more discussion of this point.)

The v...printf Functions

```
int vfprintf(FILE * restrict stream,
             const char * restrict format,
             va_list arg);                    from <stdio.h>
int vprintf(const char * restrict format,
            va_list arg);                     from <stdio.h>
int vsnprintf(char * restrict s, size_t n,
             const char * restrict format,
             va_list arg);                    from <stdio.h>
int vsprintf(char * restrict s,
             const char * restrict format,
             va_list arg);                    from <stdio.h>
```

vfprintf
vprintf
vsprintf

C99

The `vfprintf`, `vprintf`, and `vsprintf` functions (the "v...printf functions") belong to `<stdio.h>`. We're discussing them in this section because they're invariably used in conjunction with the macros in `<stdarg.h>`. C99 adds the `vsnprintf` function.

The v...printf functions are closely related to `fprintf`, `printf`, and `sprintf`. Unlike these functions, however, the v...printf functions have a fixed number of arguments. Each function's last argument is a `va_list` value, which implies that it will be called by a function with a variable argument list. In practice, the v...printf functions are used primarily for writing "wrapper" functions that accept a variable number of arguments, which are then passed to a v...printf function.

As an example, let's say that we're working on a program that needs to display error messages from time to time. We'd like each message to begin with a prefix of the form

```
** Error n:
```

where *n* is `1` for the first error message and increases by one for each subsequent error. To make it easier to produce error messages, we'll write a function named `errorf` that's similar to `printf`, but adds `** Error` *n*`:` to the beginning of

its output and always writes to stderr instead of stdout. We'll have errorf call vfprintf to do most of the actual output. Here's what errorf might look like:

```
int errorf(const char *format, ...)
{
  static int num_errors = 0;
  int n;
  va_list ap;

  num_errors++;
  fprintf(stderr, "** Error %d: ", num_errors);
  va_start(ap, format);
  n = vfprintf(stderr, format, ap);
  va_end(ap);
  fprintf(stderr, "\n");
  return n;
}
```

The wrapper function—errorf, in our example—is responsible for calling va_start prior to calling the v...printf function and for calling va_end after the v...printf function returns. The wrapper function is allowed to call va_arg one or more times before calling the v...printf function.

vsnprintf

The vsnprintf function was added to the C99 version of <stdio.h>. It corresponds to snprintf (discussed in Section 22.8), which is also a C99 function.

C99 The v...scanf Functions

```
int vfscanf(FILE * restrict stream,
            const char * restrict format,
            va_list arg);                    from <stdio.h>
int vscanf(const char * restrict format,
           va_list arg);                     from <stdio.h>
int vsscanf(const char * restrict s,
            const char * restrict format,
            va_list arg);                    from <stdio.h>
```

vfscanf
vscanf
vsscanf

C99 adds a set of "v...scanf functions" to the <stdio.h> header. vfscanf, vscanf, and vsscanf are equivalent to fscanf, scanf, and sscanf, respectively, except that they have a va_list parameter through which a variable argument list can be passed. Like the v...printf functions, each v...scanf function is designed to be called by a wrapper function that accepts a variable number of arguments, which it then passes to the v...scanf function. The wrapper function is responsible for calling va_start prior to calling the v...scanf function and for calling va_end after the v...scanf function returns.

26.2 The `<stdlib.h>` Header: General Utilities

`<stdlib.h>` serves as a catch-all for functions that don't fit into any of the other headers. The functions in `<stdlib.h>` fall into eight groups:

Numeric conversion functions
Pseudo-random sequence generation functions
Memory-management functions
Communication with the environment
Searching and sorting utilities
Integer arithmetic functions
Multibyte/wide-character conversion functions
Multibyte/wide-string conversion functions

We'll look at each group in turn, with three exceptions: the memory management functions, the multibyte/wide-character conversion functions, and the multibyte/wide-string conversion functions.

The memory-management functions (`malloc`, `calloc`, `realloc`, and `free`) permit a program to allocate a block of memory and then later release it or change its size. Chapter 17 describes all four functions in some detail.

The multibyte/wide-character conversion functions are used to convert a multibyte character to a wide character or vice-versa. The multibyte/wide-string conversion functions perform similar conversions between multibyte strings and wide strings. Both groups of functions are discussed in Section 25.2.

Numeric Conversion Functions

```
double atof(const char *nptr);

int atoi(const char *nptr);
long int atol(const char *nptr);
long long int atoll(const char *nptr);

double strtod(const char * restrict nptr,
              char ** restrict endptr);
float strtof(const char * restrict nptr,
             char ** restrict endptr);
long double strtold(const char * restrict nptr,
                    char ** restrict endptr);

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

```
long long int strtoll(const char * restrict nptr,
                      char ** restrict endptr,
                      int base);
unsigned long int strtoul(
  const char * restrict nptr,
  char ** restrict endptr, int base);
unsigned long long int strtoull(
  const char * restrict nptr,
  char ** restrict endptr, int base);
```

The numeric conversion functions (or "string conversion functions," as they're known in C89) convert strings containing numbers in character form to their equivalent numeric values. Three of these functions are fairly old, another three were added when the C89 standard was created, and five more were added in C99.

C99

All the numeric conversion functions—whether new or old—work in much the same way. Each function attempts to convert a string (pointed to by the `nptr` parameter) to a number. Each function skips white-space characters at the beginning of the string, treats subsequent characters as part of a number (possibly beginning with a plus or minus sign), and stops at the first character that can't be part of the number. In addition, each function returns zero if no conversion can be performed (the string is empty or the characters following any initial white space don't have the form the function is looking for).

atof
atoi
atol

The old functions (`atof`, `atoi`, and `atol`) convert a string to a `double`, `int`, or `long int` value, respectively. Unfortunately, these functions lack any way to indicate how much of the string was consumed during a conversion. Moreover, the functions have no way to indicate that a conversion was unsuccessful. (Some implementations of these functions may modify the `errno` variable when a conversion fails, but that's not guaranteed.)

errno variable ➤24.2

strtod
strtol
strtoul

The C89 functions (`strtod`, `strtol`, and `strtoul`) are more sophisticated. For one thing, they indicate where the conversion stopped by modifying the variable that `endptr` points to. (The second argument can be a null pointer if we're not interested in where the conversion ended.) To check whether a function was able to consume the entire string, we can just test whether this variable points to a null character. If no conversion could be performed, the variable that `endptr` points to is given the value of `nptr` (as long as `endptr` isn't a null pointer). What's more, `strtol` and `strtoul` have a `base` argument that specifies the base of the number being converted. All bases between 2 and 36 (inclusive) are supported.

Besides being more versatile than the old functions, `strtod`, `strtol`, and `strtoul` are better at detecting errors. Each function stores `ERANGE` in `errno` if a conversion produces a value that's outside the range of the function's return type. In addition, the `strtod` function returns plus or minus `HUGE_VAL`; the

HUGE_VAL macro ➤23.3

strtol and strtoul functions return the smallest or largest values of their respective return types. (strtol returns either LONG_MIN or LONG_MAX, and strtoul returns ULONG_MAX.)

<limits.h> macros ➤23.2

atoll
strtof
strtold
strtoll
strtoull

C99 adds the atoll, strtof, strtold, strtoll, and strtoull functions. atoll is the same as the atol function, except that it converts a string to a long long int value. strtof and strtold are the same as strtod, except that they convert a string to a float or long double value, respectively. strtoll is the same as strtol, except that it converts a string to a long long int value. strtoull is the same as strtoul, except that it converts a string to an unsigned long long int value. C99 also makes a small change to the floating-point numeric conversion functions: the string passed to strtod (as well as its newer cousins, strtof and strtold) may contain a hexadecimal floating-point number, infinity, or NaN.

Q&A

PROGRAM

Testing the Numeric Conversion Functions

The following program converts a string to numeric form by applying each of the six numeric conversion functions that exist in C89. After calling the strtod, strtol, and strtoul functions, the program also shows whether each conversion produced a valid result and whether it was able to consume the entire string. The program obtains the input string from the command line.

tnumconv.c

```
/* Tests C89 numeric conversion functions */

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

#define CHK_VALID  printf("    %s            %s\n",          \
                          errno != ERANGE ? "Yes" : "No ", \
                          *ptr == '\0' ? "Yes" : "No")

int main(int argc, char *argv[])
{
  char *ptr;

  if (argc != 2) {
    printf("usage: tnumconv string\n");
    exit(EXIT_FAILURE);
  }

  printf("Function   Return Value\n");
  printf("--------   -------------\n");
  printf("atof       %g\n", atof(argv[1]));
  printf("atoi       %d\n", atoi(argv[1]));
  printf("atol       %ld\n\n", atol(argv[1]));

  printf("Function   Return Value   Valid?   "
         "String Consumed?\n"
```

```
        "--------   ------------   ------   "
        "----------------\n");

    errno = 0;
    printf("strtod     %-12g", strtod(argv[1], &ptr));
    CHK_VALID;

    errno = 0;
    printf("strtol     %-12ld", strtol(argv[1], &ptr, 10));
    CHK_VALID;

    errno = 0;
    printf("strtoul    %-12lu", strtoul(argv[1], &ptr, 10));
    CHK_VALID;

    return 0;
}
```

If 3000000000 is the command-line argument, the output of the program might have the following appearance:

```
Function   Return Value
--------   ------------
atof       3e+09
atoi       2147483647
atol       2147483647

Function   Return Value   Valid?   String Consumed?
--------   ------------   ------   ----------------
strtod     3e+09           Yes           Yes
strtol     2147483647      No            Yes
strtoul    3000000000      Yes           Yes
```

On many machines, the number 3000000000 is too large to represent as a long integer, although it's valid as an unsigned long integer. The `atoi` and `atol` functions had no way to indicate that the number represented by their argument was out of range. In the output shown, they returned 2147483647 (the largest long integer), but the C standard doesn't guarantee this behavior. The `strtoul` function performed the conversion correctly; `strtol` returned 2147483647 (the standard requires it to return the largest long integer) and stored `ERANGE` in `errno`.

If `123.456` is the command-line argument, the output will be

```
Function   Return Value
--------   ------------
atof       123.456
atoi       123
atol       123

Function   Return Value   Valid?   String Consumed?
--------   ------------   ------   ----------------
strtod     123.456         Yes           Yes
strtol     123             Yes           No
strtoul    123             Yes           No
```

All six functions treated this string as a valid number, although the integer functions stopped at the decimal point. The `strtol` and `strtoul` functions were able to indicate that they didn't completely consume the string.

If `foo` is the command-line argument, the output will be

```
Function    Return Value
--------    ------------
atof        0
atoi        0
atol        0

Function    Return Value    Valid?    String Consumed?
--------    ------------    ------    ----------------
strtod      0                Yes            No
strtol      0                Yes            No
strtoul     0                Yes            No
```

All the functions looked at the letter `f` and immediately returned zero. The `str...` functions didn't change `errno`, but we can tell that something went wrong from the fact that the functions didn't consume the string.

Pseudo-Random Sequence Generation Functions

```
int rand(void);
void srand(unsigned int seed);
```

The `rand` and `srand` functions support the generation of pseudo-random numbers. These functions are useful in simulation programs and game-playing programs (to simulate a dice roll or the deal in a card game, for example).

rand

Each time it's called, `rand` returns a number between 0 and `RAND_MAX` (a macro defined in `<stdlib.h>`). The numbers returned by `rand` aren't actually random; they're generated from a "seed" value. To the casual observer, however, `rand` appears to produce an unrelated sequence of numbers.

srand

Calling `srand` supplies the seed value for `rand`. If `rand` is called prior to `srand`, the seed value is assumed to be 1. Each seed value determines a particular sequence of pseudo-random numbers; `srand` allows us to select which sequence we want.

A program that always uses the same seed value will always get the same sequence of numbers from `rand`. This property can sometimes be useful: the program behaves the same way each time it's run, making testing easier. However, we usually want `rand` to produce a *different* sequence each time the program is run. (A poker-playing program that always deals the same cards isn't likely to be popular.) The easiest way to "randomize" the seed values is to call the `time` function, which returns a number that encodes the current date and time. Passing `time`'s return value to `srand` makes the behavior of `rand` vary from one run to the next. See the `guess.c` and `guess2.c` programs (Section 10.2) for examples of this technique.

time function ➤26.3

PROGRAM **Testing the Pseudo-Random Sequence Generation Functions**

The following program displays the first five values returned by the `rand` function, then allows the user to choose a new seed value. The process repeats until the user enters zero as the seed.

trand.c

```
/* Tests the pseudo-random sequence generation functions */

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

int main(void)
{
  int i, seed;

  printf("This program displays the first five values of "
         "rand.\n");

  for (;;) {
    for (i = 0; i < 5; i++)
      printf("%d ", rand());
    printf("\n\n");
    printf("Enter new seed value (0 to terminate): ");
    scanf("%d", &seed);
    if (seed == 0)
      break;
    srand(seed);
  }

  return 0;
}
```

Here's how a session with the program might look:

```
This program displays the first five values of rand.
1804289383 846930886 1681692777 1714636915 1957747793

Enter new seed value (0 to terminate): 100
677741240 611911301 516687479 1039653884 807009856

Enter new seed value (0 to terminate): 1
1804289383 846930886 1681692777 1714636915 1957747793

Enter new seed value (0 to terminate): 0
```

There are many ways to write the `rand` function, so there's no guarantee that every version of `rand` will generate the numbers shown here. Note that choosing 1 as the seed gives the same sequence of numbers as not specifying the seed at all.

Communication with the Environment

```
void abort(void);
int atexit(void (*func)(void));
```

```
void exit(int status);
void _Exit(int status);
char *getenv(const char *name);
int system(const char *string);
```

The functions in this group provide a simple interface to the operating system, allowing programs to (1) terminate, either normally or abnormally, and return a status code to the operating system, (2) fetch information from the user's environment, and (3) execute operating system commands. One of the functions, `_Exit`, is a C99 addition.

C99

exit

Performing the call `exit(n)` anywhere in a program is normally equivalent to executing the statement `return n;` in `main`: the program terminates, and *n* is returned to the operating system as a status code. `<stdlib.h>` defines the macros `EXIT_FAILURE` and `EXIT_SUCCESS`, which can be used as arguments to `exit`. The only other portable argument to `exit` is `0`, which has the same meaning as `EXIT_SUCCESS`. Returning status codes other than these is legal but not necessarily portable to all operating systems.

Q&A

atexit

When a program terminates, it usually performs a few final actions behind the scenes, including flushing output buffers that contain unwritten data, closing open streams, and deleting temporary files. We may have other "clean-up" actions that we'd like a program to perform at termination. The `atexit` function allows us to "register" a function to be called upon program termination. To register a function named `cleanup`, for example, we could call `atexit` as follows:

```
atexit(cleanup);
```

When we pass a function pointer to `atexit`, it stores the pointer away for future reference. If the program later terminates normally (via a call of `exit` or a `return` statement in the `main` function), any function registered with `atexit` will be called automatically. (If two or more functions have been registered, they're called in the reverse of the order in which they were registered.)

_Exit

The `_Exit` function is similar to `exit`. However, `_Exit` doesn't call functions that have been registered with `atexit`, nor does it call any signal handlers previously passed to the `signal` function. Also, `_Exit` doesn't necessarily flush output buffers, close open streams, or delete temporary files—whether these actions are performed is implementation-defined.

`signal` function ➤24.3

abort

`abort` is also similar to `exit`, but calling it causes abnormal program termination. Functions registered with `atexit` aren't called. Depending on the implementation, it may be the case that output buffers containing unwritten data aren't flushed, streams aren't closed, and temporary files aren't deleted. `abort` returns an implementation-defined status code indicating unsuccessful termination.

Q&A

getenv

Many operating systems provide an "environment": a set of strings that describe the user's characteristics. These strings typically include the path to be searched when the user runs a program, the type of the user's terminal (in the case of a multi-user system), and so on. For example, a UNIX search path might look

something like this:

```
PATH=/usr/local/bin:/bin:/usr/bin:.
```

getenv provides access to any string in the user's environment. To find the current value of the PATH string, for example, we could write

```
char *p = getenv("PATH");
```

p now points to the string "/usr/local/bin:/bin:/usr/bin:.". Be careful with getenv: it returns a pointer to a statically allocated string that may be changed by a later call of the function.

system The system function allows a C program to run another program (possibly an operating system command). The argument to system is a string containing a command, similar to one that we'd enter at the operating system prompt. For example, suppose that we're writing a program that needs a listing of the files in the current directory. A UNIX program would call system in the following way:

```
system("ls >myfiles");
```

This call invokes the UNIX command ls and asks it to write a listing of the current directory into the file named myfiles.

The return value of system is implementation-defined. system typically returns the termination status code from the program that we asked it to run; testing this value allows us to check whether the program worked properly. Calling system with a null pointer has a special meaning: the function returns a nonzero value if a command processor is available.

Searching and Sorting Utilities

```
void *bsearch(const void *key, const void *base,
              size_t nmemb, size_t size,
              int (*compar)(const void *,
                            const void *));
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

bsearch The bsearch function searches a sorted array for a particular value (the "key"). When bsearch is called, the key parameter points to the key, base points to the array, nmemb is the number of elements in the array, size is the size of each element (in bytes), and compar is a pointer to a comparison function. The comparison function is similar to the one required by qsort: when passed pointers to the key and an array element (in that order), the function must return a negative, zero, or positive integer depending on whether the key is less than, equal to, or greater than the array element. bsearch returns a pointer to an element that matches the key; if it doesn't find a match, bsearch returns a null pointer.

Although the C standard doesn't require it to, `bsearch` normally uses the binary search algorithm to search the array. `bsearch` first compares the key with the element in the middle of the array; if there's a match, the function returns. If the key is smaller than the middle element, `bsearch` limits its search to the first half of the array; if the key is larger, `bsearch` searches only the last half of the array. `bsearch` repeats this strategy until it finds the key or runs out of elements to search. Thanks to this technique, `bsearch` is quite fast—searching an array of 1000 elements requires only 10 comparisons at most; searching an array of 1,000,000 elements requires no more than 20 comparisons.

qsort

Section 17.7 discusses the `qsort` function, which can sort any array. `bsearch` works only for sorted arrays, but we can always use `qsort` to sort an array prior to asking `bsearch` to search it.

PROGRAM

Determining Air Mileage

Our next program computes the air mileage from New York City to various international cities. The program first asks the user to enter a city name, then displays the mileage to that city:

```
Enter city name: Shanghai
Shanghai is 7371 miles from New York City.
```

The program will store city/mileage pairs in an array. By using `bsearch` to search the array for a city name, the program can easily find the corresponding mileage. (Mileages are from *Infoplease.com*.)

airmiles.c

```
/* Determines air mileage from New York to other cities */

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

struct city_info {
  char *city;
  int miles;
};

int compare_cities(const void *key_ptr,
                   const void *element_ptr);

int main(void)
{
  char city_name[81];
  struct city_info *ptr;
  const struct city_info mileage[] =
    {{"Berlin",        3965}, {"Buenos Aires", 5297},
     {"Cairo",         5602}, {"Calcutta",     7918},
     {"Cape Town",     7764}, {"Caracas",      2132},
     {"Chicago",        713}, {"Hong Kong",    8054},
     {"Honolulu",      4964}, {"Istanbul",     4975},
```

```
{"Lisbon",        3364}, {"London",        3458},
{"Los Angeles",   2451}, {"Manila",        8498},
{"Mexico City",   2094}, {"Montreal",       320},
{"Moscow",        4665}, {"Paris",         3624},
{"Rio de Janeiro", 4817}, {"Rome",         4281},
{"San Francisco", 2571}, {"Shanghai",      7371},
{"Stockholm",     3924}, {"Sydney",        9933},
{"Tokyo",         6740}, {"Warsaw",        4344},
{"Washington",     205}};

  printf("Enter city name: ");
  scanf("%80[^\n]", city_name);
  ptr = bsearch(city_name, mileage,
                sizeof(mileage) / sizeof(mileage[0]),
                sizeof(mileage[0]), compare_cities);
  if (ptr != NULL)
    printf("%s is %d miles from New York City.\n",
           city_name, ptr->miles);
  else
    printf("%s wasn't found.\n", city_name);

  return 0;
}

int compare_cities(const void *key_ptr,
                   const void *element_ptr)
{
  return strcmp((char *) key_ptr,
                ((struct city_info *) element_ptr)->city);
}
```

Integer Arithmetic Functions

```
int abs(int j);
long int labs(long int j);
long long int llabs(long long int j);

div_t div(int numer, int denom);
ldiv_t ldiv(long int numer, long int denom);
lldiv_t lldiv(long long int numer,
              long long int denom);
```

abs
labs

The `abs` function returns the absolute value of an `int` value; the `labs` function returns the absolute value of a `long int` value.

div

The `div` function divides its first argument by its second, returning a `div_t` value. `div_t` is a structure that contains both a quotient member (named `quot`) and a remainder member (`rem`). For example, if `ans` is a `div_t` variable, we could write

```
ans = div(5, 2);
printf("Quotient: %d Remainder: %d\n", ans.quot, ans.rem);
```

ldiv

The `ldiv` function is similar but works with long integers; it returns an `ldiv_t` structure, which also has `quot` and `rem` members. (The `div_t` and `ldiv_t` types are declared in `<stdlib.h>`.)

Q&A

llabs
lldiv

C99 provides two additional functions. The `llabs` function returns the absolute value of a `long long int` value. `lldiv` is similar to `div` and `ldiv`, except that it divides two `long long int` values and returns an `lldiv_t` structure. (The `lldiv_t` type was also added in C99.)

C99

26.3 The `<time.h>` Header: Date and Time

The `<time.h>` header provides functions for determining the time (including the date), performing arithmetic on time values, and formatting times for display. Before we explore these functions, however, we need to discuss how times are stored. `<time.h>` provides three types, each of which represents a different way to store a time:

- `clock_t`: A time value measured in "clock ticks."
- `time_t`: A compact, encoded time and date (a ***calendar time***).
- `struct tm`: A time that has been divided into seconds, minutes, hours, and so on. A value of type `struct tm` is often called a ***broken-down time.*** Table 26.1 shows the members of the `tm` structure. All members are of type `int`.

Table 26.1 Members of the `tm` Structure

Name	*Description*	*Minimum Value*	*Maximum Value*
`tm_sec`	Seconds after the minute	0	61†
`tm_min`	Minutes after the hour	0	59
`tm_hour`	Hours since midnight	0	23
`tm_mday`	Day of the month	1	31
`tm_mon`	Months since January	0	11
`tm_year`	Years since 1900	0	–
`tm_wday`	Days since Sunday	0	6
`tm_yday`	Days since January 1	0	365
`tm_isdst`	Daylight Saving Time flag	††	††

†Allows for two extra "leap seconds." In C99, the maximum value is 60.
††Positive if Daylight Saving Time is in effect, zero if it's not in effect, and negative if this information is unknown.

These types are used for different purposes. A `clock_t` value is good only for representing a time duration; `time_t` and `struct tm` values can store an entire date and time. `time_t` values are tightly encoded, so they occupy little space. `struct tm` values require much more space, but they're often easier to work with. The C standard states that `clock_t` and `time_t` must be "arithmetic types," but leaves it at that. We don't even know if `clock_t` and `time_t` values are stored as integers or floating-point numbers.

We're now ready to look at the functions in `<time.h>`, which fall into two groups: time manipulation functions and time conversion functions.

Time Manipulation Functions

```
clock_t clock(void);
double difftime(time_t time1, time_t time0);
time_t mktime(struct tm *timeptr);
time_t time(time_t *timer);
```

clock The `clock` function returns a `clock_t` value representing the processor time used by the program since execution began. To convert this value to seconds, we can divide it by `CLOCKS_PER_SEC`, a macro defined in `<time.h>`.

When `clock` is used to determine how long a program has been running, it's customary to call it twice: once at the beginning of `main` and once just before the program terminates:

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

int main(void)
{
  clock_t start_clock = clock();
  ...
  printf("Processor time used: %g sec.\n",
         (clock() - start_clock) / (double) CLOCKS_PER_SEC);
  return 0;
}
```

The reason for the initial call of `clock` is that the program will use some processor time before it reaches `main`, thanks to hidden "start-up" code. Calling `clock` at the beginning of `main` determines how much time the start-up code requires so that we can subtract it later.

The C89 standard says only that `clock_t` is an arithmetic type; the type of `CLOCKS_PER_SEC` is unspecified. As a result, the type of the expression

```
(clock() - start_clock) / CLOCKS_PER_SEC
```

may vary from one implementation to another, making it difficult to display using `printf`. To solve the problem, our example converts `CLOCKS_PER_SEC` to `double`, forcing the entire expression to have type `double`. In C99, the type of `CLOCKS_PER_SEC` is specified to be `clock_t`, but `clock_t` is still an implementation-defined type.

C99

time The `time` function returns the current calendar time. If its argument isn't a null pointer, `time` also stores the calendar time in the object that the argument points to. `time`'s ability to return a time in two different ways is an historical quirk, but it gives us the option of writing either

```
cur_time = time(NULL);
```

or

```
time(&cur_time);
```

where `cur_time` is a variable of type `time_t`.

difftime

The `difftime` function returns the difference between `time0` (the earlier time) and `time1`, measured in seconds. Thus, to compute the actual running time of a program (not the processor time), we could use the following code:

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

int main(void)
{
  time_t start_time = time(NULL);
  ...
  printf("Running time: %g sec.\n",
         difftime(time(NULL), start_time));
  return 0;
}
```

mktime

The `mktime` function converts a broken-down time (stored in the structure that its argument points to) into a calendar time, which it then returns. As a side effect, `mktime` adjusts the members of the structure according to the following rules:

- `mktime` changes any members whose values aren't within their legal ranges (see Table 26.1). Those alterations may in turn require changes to other members. If `tm_sec` is too large, for example, `mktime` reduces it to the proper range (0–59), adding the extra minutes to `tm_min`. If `tm_min` is now too large, `mktime` reduces it and adds the extra hours to `tm_hour`. If necessary, the process will continue to the `tm_mday`, `tm_mon`, and `tm_year` members.
- After adjusting the other members of the structure (if necessary), `mktime` sets `tm_wday` (day of the week) and `tm_yday` (day of the year) to their correct values. There's never any need to initialize the values of `tm_wday` and `tm_yday` before calling `mktime`; it ignores the original values of these members.

`mktime`'s ability to adjust the members of a `tm` structure makes it useful for time-related arithmetic. As a example, let's use `mktime` to answer the following question: If the 2012 Olympics begin on July 27 and end 16 days later, what is the ending date? We'll start by storing July 27, 2012 in a `tm` structure:

```
struct tm t;

t.tm_mday = 27;
t.tm_mon = 6;       /* July */
t.tm_year = 112;    /* 2012 */
```

We'll also initialize the other members of the structure (except `tm_wday` and `tm_yday`) to ensure that they don't contain undefined values that could affect the answer:

```
t.tm_sec = 0;
t.tm_min = 0;
t.tm_hour = 0;
t.tm_isdst = -1;
```

Next, we'll add 16 to the `tm_mday` member:

```
t.tm_mday += 16;
```

That leaves 43 in `tm_mday`, which is out of range for that member. Calling `mktime` will bring the members of the structure back into their proper ranges:

```
mktime(&t);
```

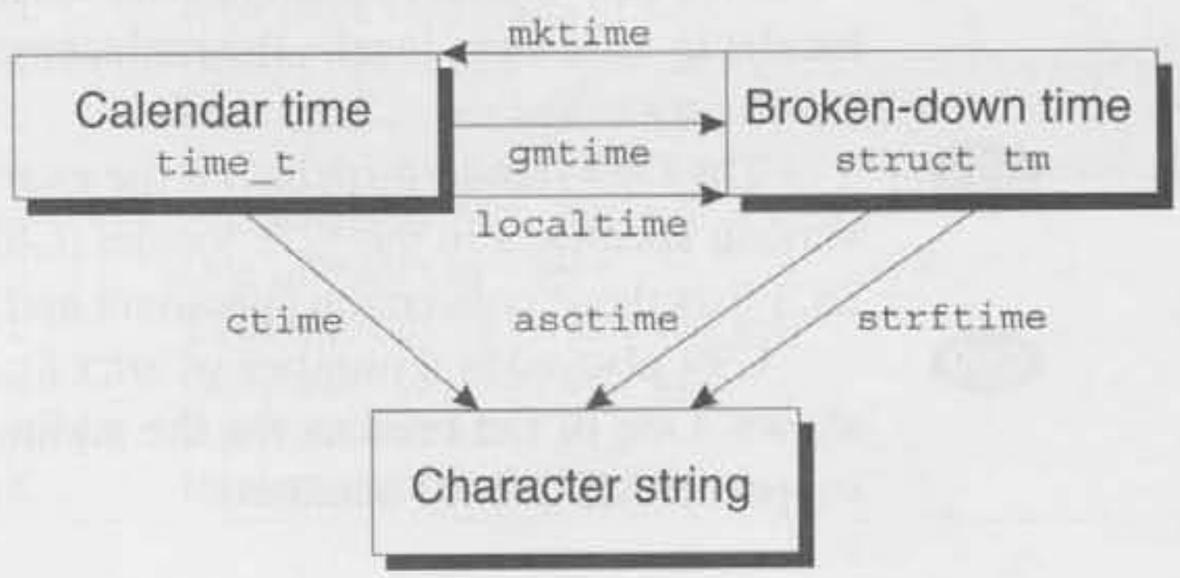
We'll discard `mktime`'s return value, since we're interested only in the function's effect on `t`. The relevant members of `t` now have the following values:

Member	*Value*	*Meaning*
`tm_mday`	12	12
`tm_mon`	7	August
`tm_year`	112	2012
`tm_wday`	0	Sunday
`tm_yday`	224	225th day of the year

Time Conversion Functions

```
char *asctime(const struct tm *timeptr);
char *ctime(const time_t *timer);
struct tm *gmtime(const time_t *timer);
struct tm *localtime(const time_t *timer);
size_t strftime(char * restrict s, size_t maxsize,
                const char * restrict format,
                const struct tm * restrict timeptr);
```

The time conversion functions make it possible to convert calendar times to broken-down times. They can also convert times (calendar or broken-down) to string form. The following figure shows how these functions are related:



The figure includes the `mktime` function, which the C standard classifies as a "manipulation" function rather than a "conversion" function.

gmtime
localtime
Q&A

The `gmtime` and `localtime` functions are similar. When passed a pointer to a calendar time, both return a pointer to a structure containing the equivalent broken-down time. `localtime` produces a local time, while `gmtime`'s return value is expressed in UTC (Coordinated Universal Time). The return value of `gmtime` and `localtime` points to a statically allocated structure that may be changed by a later call of either function.

asctime

The `asctime` (ASCII time) function returns a pointer to a null-terminated string of the form

```
Sun Jun  3 17:48:34 2007\n
```

constructed from the broken-down time pointed to by its argument.

ctime

The `ctime` function returns a pointer to a string describing a local time. If `cur_time` is a variable of type `time_t`, the call

```
ctime(&cur_time)
```

is equivalent to

```
asctime(localtime(&cur_time))
```

The return value of `asctime` and `ctime` points to a statically allocated string that may be changed by a later call of either function.

strftime

sprintf function ➤22.8

The `strftime` function, like the `asctime` function, converts a broken-down time to string form. Unlike `asctime`, however, it gives us a great deal of control over how the time is formatted. In fact, `strftime` resembles `sprintf` in that it writes characters into a string `s` (the first argument) according to a format string (the third argument). The format string may contain ordinary characters (which are copied into `s` unchanged) along with the conversion specifiers shown in Table 26.2 (which are replaced by the indicated strings). The last argument points to a `tm` structure, which is used as the source of date and time information. The second argument is a limit on the number of characters that can be stored in `s`.

locales ➤25.1

The `strftime` function, unlike the other functions in `<time.h>`, is sensitive to the current locale. Changing the `LC_TIME` category may affect the behavior of the conversion specifiers. The examples in Table 26.2 are strictly for the `"C"` locale; in a German locale, the replacement for `%A` might be `Dienstag` instead of `Tuesday`.

C99

The C99 standard spells out the exact replacement strings for some of the conversion specifiers in the `"C"` locale. (C89 didn't go into this level of detail.) Table 26.3 lists these conversion specifiers and the strings they're replaced by.

C99

C99 also adds a number of `strftime` conversion specifiers, as Table 26.2 shows. One of the reasons for the additional conversion specifiers is the desire to support the ISO 8601 standard.

Table 26.2
Conversion Specifiers for the `strftime` Function

Conversion	*Replacement*
`%a`	Abbreviated weekday name (e.g., `Sun`)
`%A`	Full weekday name (e.g., `Sunday`)
`%b`	Abbreviated month name (e.g., `Jun`)
`%B`	Full month name (e.g., `June`)
`%c`	Complete day and time (e.g., `Sun Jun 3 17:48:34 2007`)
`%C`†	Year divided by 100 and truncated to an integer (`00–99`)
`%d`	Day of month (`01–31`)
`%D`†	Equivalent to `%m/%d/%y`
`%e`†	Day of month (`1–31`); a single digit is preceded by a space
`%F`†	Equivalent to `%Y-%m-%d`
`%g`†	Last two digits of ISO 8601 week-based year (`00–99`)
`%G`†	ISO 8601 week-based year
`%h`†	Equivalent to `%b`
`%H`	Hour on 24-hour clock (`00–23`)
`%I`	Hour on 12-hour clock (`01–12`)
`%j`	Day of year (`001–366`)
`%m`	Month (`01–12`)
`%M`	Minute (`00–59`)
`%n`†	New-line character
`%p`	AM/PM designator (`AM` or `PM`)
`%r`†	12-hour clock time (e.g., `05:48:34 PM`)
`%R`†	Equivalent to `%H:%M`
`%S`	Second (`00–61`); maximum value in C99 is 60
`%t`†	Horizontal-tab character
`%T`†	Equivalent to `%H:%M:%S`
`%u`†	ISO 8601 weekday (`1–7`); Monday is 1
`%U`	Week number (`00–53`); first Sunday is beginning of week 1
`%V`†	ISO 8601 week number (`01–53`)
`%w`	Weekday (`0–6`); Sunday is 0
`%W`	Week number (`00–53`); first Monday is beginning of week 1
`%x`	Complete date (e.g., `06/03/07`)
`%X`	Complete time (e.g., `17:48:34`)
`%y`	Last two digits of year (`00–99`)
`%Y`	Year
`%z`†	Offset from UTC in ISO 8601 format (e.g., `-0530` or `+0200`)
`%Z`	Time zone name or abbreviation (e.g., `EST`)
`%%`	`%`

†C99 only

Table 26.3
Replacement Strings for `strftime` Conversion Specifiers in the `"C"` Locale

Conversion	*Replacement*
`%a`	First three characters of `%A`
`%A`	One of `"Sunday"`, `"Monday"`, ..., `"Saturday"`
`%b`	First three characters of `%B`
`%B`	One of `"January"`, `"February"`, ..., `"December"`
`%c`	Equivalent to `"%a %b %e %T %Y"`
`%p`	One of `"AM"` or `"PM"`
`%r`	Equivalent to `"%I:%M:%S %p"`
`%x`	Equivalent to `"%m/%d/%y"`
`%X`	Equivalent to `%T`
`%Z`	Implementation-defined

ISO 8601

ISO 8601 is an international standard that describes ways of representing dates and times. It was originally published in 1988 and later updated in 2000 and 2004. According to this standard, dates and times are entirely numeric (i.e., months are not represented by names) and hours are expressed using the 24-hour clock.

There are a number of ISO 8601 date and time formats, some of which are directly supported by `strftime` conversion specifiers in C99. The primary ISO 8601 date format (*YYYY–MM–DD*) and the primary time format (*hh*:*mm*:*ss*) correspond to the `%F` and `%T` conversion specifiers, respectively.

ISO 8601 has a system of numbering the weeks of a year; this system is supported by the `%g`, `%G`, and `%V` conversion specifiers. Weeks begin on Monday, and week 1 is the week containing the first Thursday of the year. Consequently, the first few days of January (as many as three) may belong to the last week of the previous year. For example, consider the calendar for January 2011:

January 2011

Mo	*Tu*	*We*	*Th*	*Fr*	*Sa*	*Su*
					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
31						

Year	*Week*
2010	52
2011	1
2011	2
2011	3
2011	4
2011	5

January 6 is the first Thursday of the year, so the week of January 3–9 is week 1. January 1 and January 2 belong to the last week (week 52) of the previous year. For these two dates, `strftime` will replace `%g` by `10`, `%G` by `2010`, and `%V` by `52`. Note that the last few days of December will sometimes belong to week 1 of the following year; this happens whenever December 29, 30, or 31 is a Monday.

The `%z` conversion specifier corresponds to the ISO 8601 time zone specification: *–hhmm* means that a time zone is *hh* hours and *mm* minutes behind UTC; the string *+hhmm* indicates the amount by which a time zone is ahead of UTC.

C99

C99 allows the use of an `E` or `O` character to modify the meaning of certain `strftime` conversion specifiers. Conversion specifiers that begin with an `E` or `O` modifier cause a replacement to be performed using an alternative format that depends on the current locale. If an alternative representation doesn't exist in the current locale, the modifier has no effect. (In the `"C"` locale, `E` and `O` are ignored.) Table 26.4 lists all conversion specifiers that are allowed to have `E` or `O` modifiers.

PROGRAM

Displaying the Date and Time

Let's say we need a program that displays the current date and time. The program's first step, of course, is to call the `time` function to obtain the calendar time. The

Table 26.4 E- and O-Modified Conversion Specifiers for the `strftime` Function (C99 only)

Conversion	*Replacement*
`%Ec`	Alternative date and time representation
`%EC`	Name of base year (period) in alternative representation
`%Ex`	Alternative date representation
`%EX`	Alternative time representation
`%Ey`	Offset from `%EC` (year only) in alternative representation
`%EY`	Full alternative year representation
`%Od`	Day of month, using alternative numeric symbols (filled with leading zeros or with leading spaces if there is no alternative symbol for zero)
`%Oe`	Day of month, using alternative numeric symbols (filled with leading spaces)
`%OH`	Hour on 24-hour clock, using alternative numeric symbols
`%OI`	Hour on 12-hour clock, using alternative numeric symbols
`%Om`	Month, using alternative numeric symbols
`%OM`	Minute, using alternative numeric symbols
`%OS`	Second, using alternative numeric symbols
`%Ou`	ISO 8601 weekday as a number in alternative representation, where Monday is 1
`%OU`	Week number, using alternative numeric symbols
`%OV`	ISO 8601 week number, using alternative numeric symbols
`%Ow`	Weekday as a number, using alternative numeric symbols
`%OW`	Week number, using alternative numeric symbols
`%Oy`	Last two digits of year, using alternative numeric symbols

second step is to convert the time to string form and print it. The easiest way to do the second step is to call `ctime`, which returns a pointer to a string containing a date and time, then pass this pointer to `puts` or `printf`.

So far, so good. But what if we want the program to display the date and time in a particular way? Let's assume that we need the following format, where `06` is the month and `03` is the day of the month:

```
06-03-2007   5:48p
```

The `ctime` function always uses the same format for the date and time, so it's no help. The `strftime` function is better; using it, we can almost achieve the appearance that we want. Unfortunately, `strftime` won't let us display a one-digit hour without a leading zero. Also, `strftime` uses `AM` and `PM` instead of `a` and `p`.

When `strftime` isn't good enough, we have another alternative: convert the calendar time to a broken-down time, then extract the relevant information from the `tm` structure and format it ourselves using `printf` or a similar function. We might even use `strftime` to do some of the formatting before having other functions complete the job.

The following program illustrates the options. It displays the current date and time in three formats: the one used by `ctime`, one close to what we want (created using `strftime`), and the desired format (created using `printf`). The `ctime` version is easy to do, the `strftime` version is a little harder, and the `printf` version is the most difficult.

datetime.c

```
/* Displays the current date and time in three formats */

#include <stdio.h>
#include <time.h>

int main(void)
{
  time_t current = time(NULL);
  struct tm *ptr;
  char date_time[21];
  int hour;
  char am_or_pm;

  /* Print date and time in default format */
  puts(ctime(&current));

  /* Print date and time, using strftime to format */
  strftime(date_time, sizeof(date_time),
           "%m-%d-%Y  %I:%M%p\n", localtime(&current));
  puts(date_time);

  /* Print date and time, using printf to format */
  ptr = localtime(&current);
  hour = ptr->tm_hour;
  if (hour <= 11)
    am_or_pm = 'a';
  else {
    hour -= 12;
    am_or_pm = 'p';
  }
  if (hour == 0)
    hour = 12;
  printf("%.2d-%.2d-%d  %2d:%.2d%c\n", ptr->tm_mon + 1,
         ptr->tm_mday, ptr->tm_year + 1900, hour,
         ptr->tm_min, am_or_pm);

  return 0;
}
```

The output of `datetime.c` will have the following appearance:

```
Sun Jun  3 17:48:34 2007

06-03-2007  05:48PM

06-03-2007   5:48p
```

Q & A

Q: Although `<stdlib.h>` provides a number of functions that convert strings to numbers, there don't appear to be any functions that convert numbers to strings. What gives?

A: Some C libraries supply functions with names like `itoa` that convert numbers to strings. Using these functions isn't a great idea, though: they aren't part of the C standard and won't be portable. The best way to perform this kind of conversion is to call a function such as `sprintf` that writes formatted output into a string:

sprintf function ➤22.8

```
char str[20];
int i;
...
sprintf(str, "%d", i);    /* writes i into the string str */
```

Not only is `sprintf` portable, but it also provides a great deal of control over the appearance of the number.

***Q: The description of the `strtod` function says that C99 allows the string argument to contain a hexadecimal floating-point number, infinity, or NaN. What is the format of these numbers? [p. 684]**

A: A hexadecimal floating-point number begins with `0x` or `0X`, followed by one or more hexadecimal digits (possibly including a decimal-point character), and then possibly a binary exponent. (See the Q&A at the end of Chapter 7 for a discussion of hexadecimal floating constants, which have a similar—but not identical—format.) Infinity has the form `INF` or `INFINITY`; any or all of the letters may be lower-case. NaN is represented by the string `NAN` (again ignoring case), possibly followed by a pair of parentheses. The parentheses may be empty or they may contain a series of characters, where each character is a letter, digit, or underscore. The characters may be used to specify some of the bits in the binary representation of the NaN value, but their exact meaning is implementation-defined. The same kind of character sequence—which the C99 standard calls an *n-char-sequence*—is also used in calls of the `nan` function.

nan function ➤23.4

***Q: You said that performing the call `exit(n)` anywhere in a program is *normally* equivalent to executing the statement `return n;` in `main`. When would it not be equivalent? [p. 688]**

A: There are two issues. First, when the `main` function returns, the lifetime of its local variables ends (assuming that they have automatic storage duration, as they will unless they're declared to be `static`), which isn't true if the `exit` function is called. A problem will occur if any action that takes place at program termination—such as calling a function previously registered using `atexit` or flushing an output stream buffer—requires access to one of these variables. In particular, a program might have called `setvbuf` and used one of `main`'s variables as a buffer. Thus, in rare cases a program may behave improperly if it attempts to return from `main` but work if it calls `exit` instead.

automatic storage duration ➤18.2

setvbuf function ➤22.2

C99

The other issue occurs only in C99, which makes it legal for `main` to have a return type other than `int` if an implementation explicitly allows the programmer to do so. In these circumstances, the call `exit(n)` isn't necessarily equivalent to executing `return n;` in `main`. In fact, the statement `return n;` may be illegal (if `main` is declared to return `void`, for example).

***Q: Is there a relationship between the `abort` function and `SIGABRT` signal? [p. 688]**

A: Yes. A call of `abort` actually raises the `SIGABRT` signal. If there's no handler for `SIGABRT`, the program terminates abnormally as described in Section 26.2. If a handler has been installed for `SIGABRT` (by a call of the `signal` function), the handler is called. If the handler returns, the program then terminates abnormally. However, if the handler *doesn't* return (it calls `longjmp`, for example), then the program doesn't terminate.

`signal` function ➤24.3

`longjmp` function ➤24.4

Q: Why do the `div` and `ldiv` functions exist? Can't we just use the / and % operators? [p. 692]

A: `div` and `ldiv` aren't quite the same as / and %. Recall from Section 4.1 that applying / and % to negative operands doesn't give a portable result in C89. If `i` or `j` is negative, whether the value of `i / j` is rounded up or down is implementation-defined, as is the sign of `i % j`. The answers computed by `div` and `ldiv`, on the other hand, don't depend on the implementation. The quotient is rounded toward zero; the remainder is computed according to the formula $n = q \times d + r$, where n is the original number, q is the quotient, d is the divisor, and r is the remainder. Here are a few examples:

n	d	q	r
7	3	2	1
–7	3	–2	–1
7	–3	–2	1
–7	–3	2	–1

C99 In C99, the / and % operators are guaranteed to produce the same result as `div` and `ldiv`.

Efficiency is the other reason that `div` and `ldiv` exist. Many machines have an instruction that can compute both the quotient and remainder, so calling `div` or `ldiv` may be faster than using the / and % operators separately.

Q: Where does the name of the `gmtime` function come from? [p. 696]

A: The name `gmtime` stands for Greenwich Mean Time (GMT), referring to the local (solar) time at the Royal Observatory in Greenwich, England. In 1884, GMT was adopted as an international reference time, with other time zones expressed as hours "behind GMT" or "ahead of GMT." In 1972, Coordinated Universal Time (UTC)—a system based on atomic clocks rather than solar observations—replaced GMT as the international time reference. By adding a "leap second" once every few years, UTC is kept synchronized with GMT to within 0.9 second, so for all but the most precise time measurements the two systems are identical.

Exercises

Section 26.1

1. Rewrite the `max_int` function so that, instead of passing the number of integers as the first argument, we must supply 0 as the last argument. *Hint:* `max_int` must have at least one

"normal" parameter, so you can't remove the parameter n. Instead, assume that it represents one of the numbers to be compared.

W 2. Write a simplified version of `printf` in which the only conversion specification is `%d`, and all arguments after the first are assumed to have `int` type. If the function encounters a `%` character that's not immediately followed by a `d` character, it should ignore both characters. The function should use calls of `putchar` to produce all output. You may assume that the format string doesn't contain escape sequences.

3. Extend the function of Exercise 2 so that it allows two conversion specifications: `%d` and `%s`. Each `%d` in the format string indicates an `int` argument, and each `%s` indicates a `char *` (string) argument.

4. Write a function named `display` that takes any number of arguments. The first argument must be an integer. The remaining arguments will be strings. The first argument specifies how many strings the call contains. The function will print the strings on a single line, with adjacent strings separated by one space. For example, the call

```
display(4, "Special", "Agent", "Dale", "Cooper");
```

will produce the following output:

```
Special Agent Dale Cooper
```

5. Write the following function:

```
char *vstrcat(const char *first, ...);
```

All arguments of `vstrcat` are assumed to be strings, except for the last argument, which must be a null pointer (cast to `char *` type). The function returns a pointer to a dynamically allocated string containing the concatenation of the arguments. `vstrcat` should return a null pointer if not enough memory is available. *Hint:* Have `vstrcat` go through the arguments twice: once to determine the amount of memory required for the returned string and once to copy the arguments into the string.

6. Write the following function:

```
char *max_pair(int num_pairs, ...);
```

The arguments of `max_pair` are assumed to be "pairs" of integers and strings; the value of `num_pairs` indicates how many pairs will follow. (A pair consists of an `int` argument followed by a `char *` argument). The function searches the integers to find the largest one; it then returns the string argument that follows it. Consider the following call:

```
max_pair(5, 180, "Seinfeld", 180, "I Love Lucy",
            39, "The Honeymooners", 210, "All in the Family",
            86, "The Sopranos")
```

The largest `int` argument is 210, so the function returns `"All in the Family"`, which follows it in the argument list.

Section 26.2

W 7. Explain the meaning of the following statement, assuming that `value` is a variable of type `long int` and `p` is a variable of type `char *`:

```
value = strtol(p, &p, 10);
```

8. Write a statement that randomly assigns one of the numbers 7, 11, 15, or 19 to the variable n.

W 9. Write a function that returns a random `double` value d in the range $0.0 \le d < 1.0$.

10. Convert the following calls of `atoi`, `atol`, and `atoll` into calls of `strtol`, `strtol`, and `strtoll`, respectively.

(a) `atoi(str)`
(b) `atol(str)`
(c) `atoll(str)`

11. Although the `bsearch` function is normally used with a sorted array, it will sometimes work correctly with an array that is only partially sorted. What condition must an array satisfy to guarantee that `bsearch` works properly for a particular key? *Hint:* The answer appears in the C standard.

Section 26.3

12. Write a function that, when passed a year, returns a `time_t` value representing 12:00 a.m. on the first day of that year.

13. Section 26.3 described some of the ISO 8601 date and time formats. Here are a few more:
    (a) Year followed by day of year: *YYYY–DDD*, where *DDD* is a number between 001 and 366
    (b) Year, week, and day of week: *YYYY*–W*ww*–*D*, where *ww* is a number between 01 and 53, and *D* is a digit between 1 through 7, beginning with Monday and ending with Sunday
    (c) Combined date and time: *YYYY–MM–DD*T*hh*:*mm*:*ss*

    Give `strftime` strings that correspond to each of these formats.

Programming Projects

W 1. (a) Write a program that calls the `rand` function 1000 times, printing the low-order bit of each value it returns (0 if the return value is even, 1 if it's odd). Do you see any patterns? (Often, the last few bits of `rand`'s return value aren't especially random.)

(b) How can we improve the randomness of `rand` for generating numbers within a small range?

2. Write a program that tests the `atexit` function. The program should have two functions (in addition to `main`), one of which prints `That's all,` and the other `folks!`. Use the `atexit` function to register both to be called at program termination. Make sure they're called in the proper order, so that we see the message `That's all, folks!` on the screen.

W 3. Write a program that uses the `clock` function to measure how long it takes `qsort` to sort an array of 1000 integers that are originally in reverse order. Run the program for arrays of 10000 and 100000 integers as well.

W 4. Write a program that prompts the user for a date (month, day, and year) and an integer `n`, then prints the date that's `n` days later.

5. Write a program that prompts the user to enter two dates, then prints the difference between them, measured in days. *Hint:* Use the `mktime` and `difftime` functions.

W 6. Write programs that display the current date and time in each of the following formats. Use `strftime` to do all or most of the formatting.
    (a) `Sunday, June 3, 2007  05:48p`
    (b) `Sun, 3 Jun 07  17:48`
    (c) `06/03/07  5:48:34 PM`