

16 Structures, Unions, and Enumerations

*Functions delay binding; data structures induce binding.
Moral: Structure data late in the programming process.*

This chapter introduces three new types: structures, unions, and enumerations. A structure is a collection of values (members), possibly of different types. A union is similar to a structure, except that its members share the same storage; as a result, a union can store one member at a time, but not all members simultaneously. An enumeration is an integer type whose values are named by the programmer.

Of these three types, structures are by far the most important, so I'll devote most of the chapter to them. Section 16.1 shows how to declare structure variables and perform basic operations on them. Section 16.2 then explains how to define structure types, which—among other things—allow us to write functions that accept structure arguments or return structures. Section 16.3 explores how arrays and structures can be nested. The last two sections are devoted to unions (Section 16.4) and enumerations (Section 16.5).

16.1 Structure Variables

The only data structure we've covered so far is the array. Arrays have two important properties. First, all elements of an array have the same type. Second, to select an array element, we specify its position (as an integer subscript).

The properties of a *structure* are quite different from those of an array. The elements of a structure (its *members*, in C parlance) aren't required to have the same type. Furthermore, the members of a structure have names; to select a particular member, we specify its name, not its position.

Structures may sound familiar, since most programming languages provide a similar feature. In some languages, structures are called *records*, and members are known as *fields*.

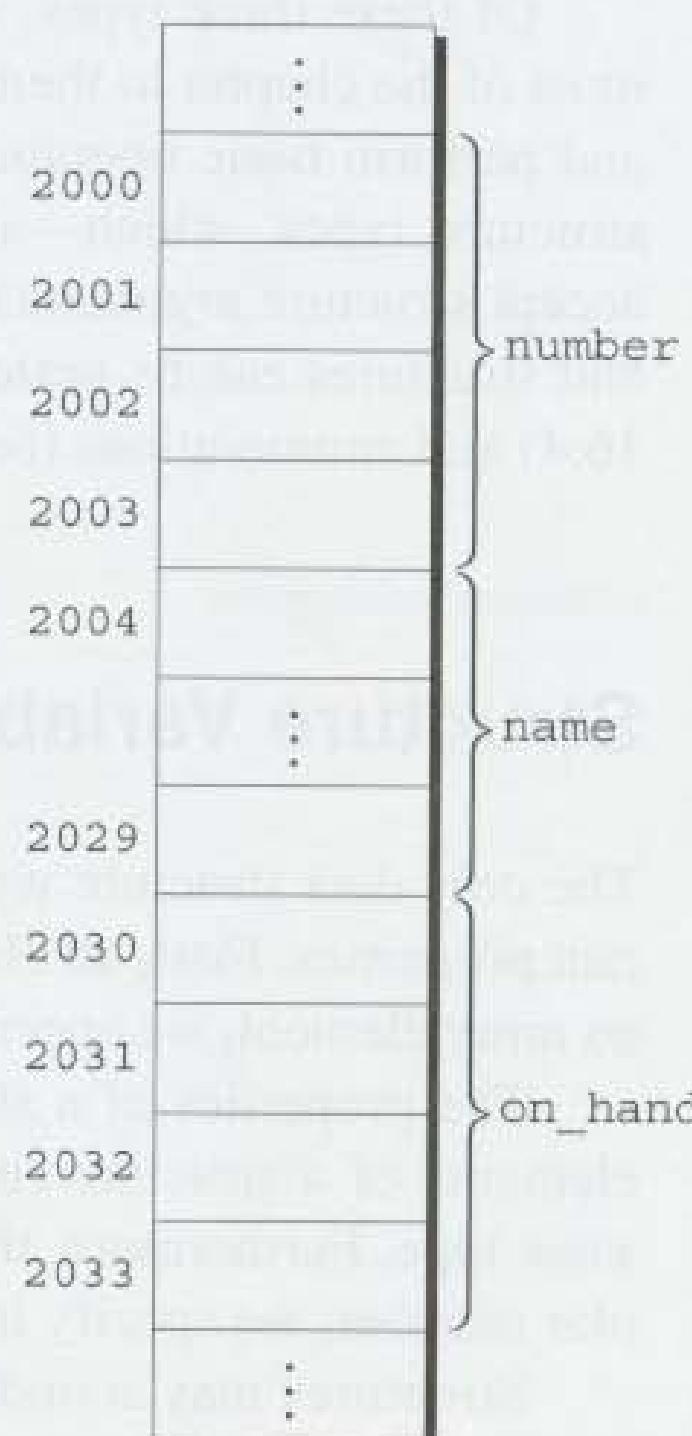
Declaring Structure Variables

When we need to store a collection of related data items, a structure is a logical choice. For example, suppose that we need to keep track of parts in a warehouse. The information that we'll need to store for each part might include a part number (an integer), a part name (a string of characters), and the number of parts on hand (an integer). To create variables that can store all three items of data, we might use a declaration such as the following:

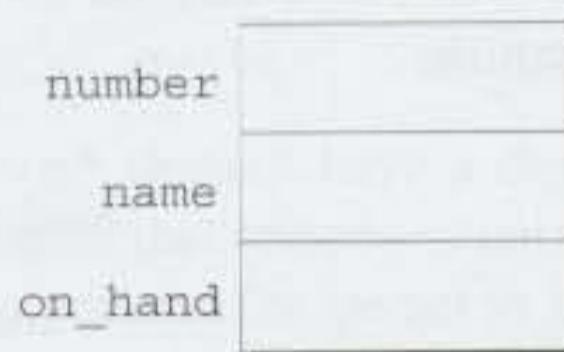
```
struct {
    int number;
    char name [NAME_LEN+1];
    int on_hand;
} part1, part2;
```

Each structure variable has three members: `number` (the part number), `name` (the name of the part), and `on_hand` (the quantity on hand). Notice that this declaration has the same form as other variable declarations in C: `struct { ... }` specifies a type, while `part1` and `part2` are variables of that type.

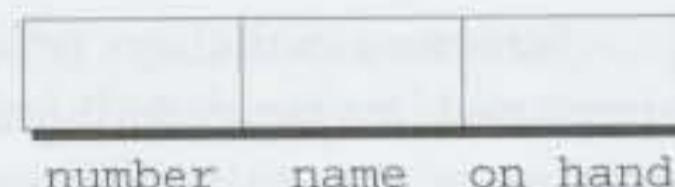
The members of a structure are stored in memory in the order in which they're declared. In order to show what the `part1` variable looks like in memory, let's assume that (1) `part1` is located at address 2000, (2) integers occupy four bytes, (3) `NAME_LEN` has the value 25, and (4) there are no gaps between the members. With these assumptions, `part1` will have the following appearance:



Usually it's not necessary to draw structures in such detail. I'll normally show them more abstractly, as a series of boxes:



I may sometimes draw the boxes horizontally instead of vertically:



Member values will go in the boxes later; for now, I've left them empty.

Each structure represents a new scope; any names declared in that scope won't conflict with other names in a program. (In C terminology, we say that each structure has a separate *name space* for its members.) For example, the following declarations can appear in the same program:

```

struct {
    int number;
    char name[NAME_LEN+1];
    int on_hand;
} part1, part2;

struct {
    char name[NAME_LEN+1];
    int number;
    char sex;
} employee1, employee2;
  
```

The `number` and `name` members in the `part1` and `part2` structures don't conflict with the `number` and `name` members in `employee1` and `employee2`.

Initializing Structure Variables

Like an array, a structure variable may be initialized at the time it's declared. To initialize a structure, we prepare a list of values to be stored in the structure and enclose it in braces:

```

struct {
    int number;
    char name[NAME_LEN+1];
    int on_hand;
} part1 = {528, "Disk drive", 10},
      part2 = {914, "Printer cable", 5};
  
```

The values in the initializer must appear in the same order as the members of the structure. In our example, the `number` member of `part1` will be 528, the `name` member will be "Disk drive", and so on. Here's how `part1` will look after initialization:

number	528
name	Disk drive
on_hand	10

Structure initializers follow rules similar to those for array initializers. Expressions used in a structure initializer must be constant; for example, we couldn't have used a variable to initialize `part1`'s `on_hand` member. (This restriction is relaxed in C99, as we'll see in Section 18.5.) An initializer can have fewer members than the structure it's initializing; as with arrays, any "leftover" members are given 0 as their initial value. In particular, the bytes in a leftover character array will be zero, making it represent the empty string.

C99

Designated Initializers

C99's designated initializers, which were discussed in Section 8.1 in the context of arrays, can also be used with structures. Consider the initializer for `part1` shown in the previous example:

```
{ 528, "Disk drive", 10 }
```

A designated initializer would look similar, but with each value labeled by the name of the member that it initializes:

```
{ .number = 528, .name = "Disk drive", .on_hand = 10 }
```

The combination of the period and the member name is called a *designator*. (Designators for array elements have a different form.)

Designated initializers have several advantages. For one, they're easier to read and check for correctness, because the reader can clearly see the correspondence between the members of the structure and the values listed in the initializer. Another is that the values in the initializer don't have to be placed in the same order that the members are listed in the structure. Our example initializer could be written as follows:

```
{ .on_hand = 10, .name = "Disk drive", .number = 528 }
```

Since the order doesn't matter, the programmer doesn't have to remember the order in which the members were originally declared. Moreover, the order of the members can be changed in the future without affecting designated initializers.

Not all values listed in a designated initializer need be prefixed by a designator. (This is true for arrays as well, as we saw in Section 8.1.) Consider the following example:

```
{ .number = 528, "Disk drive", .on_hand = 10 }
```

The value "Disk drive" doesn't have a designator, so the compiler assumes that it initializes the member that follows `number` in the structure. Any members that the initializer fails to account for are set to zero.

Operations on Structures

Since the most common array operation is subscripting—selecting an element by position—it's not surprising that the most common operation on a structure is selecting one of its members. Structure members are accessed by name, though, not by position.

To access a member within a structure, we write the name of the structure first, then a period, then the name of the member. For example, the following statements will display the values of `part1`'s members:

```
printf("Part number: %d\n", part1.number);
printf("Part name: %s\n", part1.name);
printf("Quantity on hand: %d\n", part1.on_hand);
```

Ivalues ▶ 4.2

The members of a structure are Ivalues, so they can appear on the left side of an assignment or as the operand in an increment or decrement expression:

```
part1.number = 258;           /* changes part1's part number */
part1.on_hand++;             /* increments part1's quantity on hand */
```

The period that we use to access a structure member is actually a C operator. It has the same precedence as the postfix `++` and `--` operators, so it takes precedence over nearly all other operators. Consider the following example:

```
scanf("%d", &part1.on_hand);
```

The expression `&part1.on_hand` contains two operators (`&` and `.`). The `.` operator takes precedence over the `&` operator, so `&` computes the address of `part1.on_hand`, as we wished.

The other major structure operation is assignment:

```
part2 = part1;
```

The effect of this statement is to copy `part1.number` into `part2.number`, `part1.name` into `part2.name`, and so on.

Since arrays can't be copied using the `=` operator, it comes as something of a surprise to discover that structures can. It's even more surprising when you consider that an array embedded within a structure is copied when the enclosing structure is copied. Some programmers exploit this property by creating "dummy" structures to enclose arrays that will be copied later:

table of operators ▶ Appendix A

```
struct { int a[10]; } a1, a2;
a1 = a2; /* legal, since a1 and a2 are structures */
```

The = operator can be used only with structures of *compatible* types. Two structures declared at the same time (as part1 and part2 were) are compatible. As we'll see in the next section, structures declared using the same "structure tag" or the same type name are also compatible.

Other than assignment, C provides no operations on entire structures. In particular, we can't use the == and != operators to test whether two structures are equal or not equal.

Q&A

16.2 Structure Types

Although the previous section showed how to declare structure *variables*, it failed to discuss an important issue: naming structure *types*. Suppose that a program needs to declare several structure variables with identical members. If all the variables can be declared at one time, there's no problem. But if we need to declare the variables at different points in the program, then life becomes more difficult. If we write

```
struct {
    int number;
    char name[NAME_LEN+1];
    int on_hand;
} part1;
```

in one place and

```
struct {
    int number;
    char name[NAME_LEN+1];
    int on_hand;
} part2;
```

in another, we'll quickly run into problems. Repeating the structure information will bloat the program. Changing the program later will be risky, since we can't easily guarantee that the declarations will remain consistent.

But those aren't the biggest problems. According to the rules of C, part1 and part2 don't have compatible types. As a result, part1 can't be assigned to part2, and vice versa. Also, since we don't have a name for the type of part1 or part2, we can't use them as arguments in function calls.

To avoid these difficulties, we need to be able to define a name that represents a *type* of structure, not a particular structure *variable*. As it turns out, C provides two ways to name structures: we can either declare a "structure tag" or use `typedef` to define a type name.

Q&A

Declaring a Structure Tag

A *structure tag* is a name used to identify a particular kind of structure. The following example declares a structure tag named `part`:

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

Notice the semicolon that follows the right brace—it must be present to terminate the declaration.



Accidentally omitting the semicolon at the end of a structure declaration can cause surprising errors. Consider the following example:

```
struct part {
    int number;
    char name[NAME_LEN+1];
    int on_hand;
}           /*** WRONG: semicolon missing ***/
```

```
f(void)
{
    ...
    return 0; /* error detected at this line */
}
```

The programmer failed to specify the return type of the function `f` (a bit of sloppy programming). Since the preceding structure declaration wasn't terminated properly, the compiler assumes that `f` returns a value of type `struct part`. The error won't be detected until the compiler reaches the first `return` statement in the function. The result: a cryptic error message.

Once we've created the `part` tag, we can use it to declare variables:

```
struct part part1, part2;
```

Unfortunately, we can't abbreviate this declaration by dropping the word `struct`:

```
part part1, part2; /*** WRONG ***/
```

`part` isn't a type name; without the word `struct`, it is meaningless.

Since structure tags aren't recognized unless preceded by the word `struct`, they don't conflict with other names used in a program. It would be perfectly legal (although more than a little confusing) to have a variable named `part`.

Incidentally, the declaration of a structure *tag* can be combined with the declaration of structure *variables*:

```
struct part {
    int number;
    char name[NAME_LEN+1];
    int on_hand;
} part1, part2;
```

Here, we've declared a structure tag named `part` (making it possible to use `part` later to declare more variables) as well as variables named `part1` and `part2`.

All structures declared to have type `struct part` are compatible with one another:

```
struct part part1 = {528, "Disk drive", 10};
struct part part2;

part2 = part1; /* legal; both parts have the same type */
```

Defining a Structure Type

As an alternative to declaring a structure tag, we can use `typedef` to define a genuine type name. For example, we could define a type named `Part` in the following way:

```
typedef struct {
    int number;
    char name[NAME_LEN+1];
    int on_hand;
} Part;
```

Note that the name of the type, `Part`, must come at the end, not after the word `struct`.

We can use `Part` in the same way as the built-in types. For example, we might use it to declare variables:

```
Part part1, part2;
```

Since `Part` is a `typedef` name, we're not allowed to write `struct Part`. All `Part` variables, regardless of where they're declared, are compatible.

When it comes time to name a structure, we can usually choose either to declare a structure tag or to use `typedef`. However, as we'll see later, declaring a structure tag is mandatory when the structure is to be used in a linked list. I'll use structure tags rather than `typedef` names in most of my examples.

Structures as Arguments and Return Values

Functions may have structures as arguments and return values. Let's look at two examples. Our first function, when given a `part` structure as its argument, prints the structure's members:

```
void print_part(struct part p)
{
    printf("Part number: %d\n", p.number);
```

Q&A

linked lists ► 17.5

```

    printf("Part name: %s\n", p.name);
    printf("Quantity on hand: %d\n", p.on_hand);
}

```

Here's how `print_part` might be called:

```
print_part(part1);
```

Our second function returns a `part` structure that it constructs from its arguments:

```

struct part build_part(int number, const char *name,
                      int on_hand)
{
    struct part p;

    p.number = number;
    strcpy(p.name, name);
    p.on_hand = on_hand;
    return p;
}

```

Notice that it's legal for `build_part`'s parameters to have names that match the members of the `part` structure, since the structure has its own name space. Here's how `build_part` might be called:

```
part1 = build_part(528, "Disk drive", 10);
```

Passing a structure to a function and returning a structure from a function both require making a copy of all members in the structure. As a result, these operations impose a fair amount of overhead on a program, especially if the structure is large. To avoid this overhead, it's sometimes advisable to pass a *pointer* to a structure instead of passing the structure itself. Similarly, we might have a function return a pointer to a structure instead of returning an actual structure. Section 17.5 gives examples of functions that have a pointer to a structure as an argument and/or return a pointer to a structure.

There are other reasons to avoid copying structures besides efficiency. For example, the `<stdio.h>` header defines a type named `FILE`, which is typically a structure. Each `FILE` structure stores information about the state of an open file and therefore must be unique in a program. Every function in `<stdio.h>` that opens a file returns a pointer to a `FILE` structure, and every function that performs an operation on an open file requires a `FILE` pointer as an argument.

On occasion, we may want to initialize a structure variable inside a function to match another structure, possibly supplied as a parameter to the function. In the following example, the initializer for `part2` is the parameter passed to the `f` function:

```

void f(struct part part1)
{
    struct part part2 = part1;
    ...
}

```

automatic storage duration ➤ 10.1

C permits initializers of this kind, provided that the structure we're initializing (part2, in this case) has automatic storage duration (it's local to a function and hasn't been declared `static`). The initializer can be any expression of the proper type, including a function call that returns a structure.

C99

Compound Literals

Section 9.3 introduced the C99 feature known as the *compound literal*. In that section, compound literals were used to create unnamed arrays, usually for the purpose of passing the array to a function. A compound literal can also be used to create a structure “on the fly,” without first storing it in a variable. The resulting structure can be passed as a parameter, returned by a function, or assigned to a variable. Let's look at a couple of examples.

First, we can use a compound literal to create a structure that will be passed to a function. For example, we could call the `print_part` function as follows:

```
print_part((struct part) {528, "Disk drive", 10});
```

The compound literal (shown in **bold**) creates a `part` structure containing the members 528, "Disk drive", and 10, in that order. This structure is then passed to `print_part`, which displays it.

Here's how a compound literal might be assigned to a variable:

```
part1 = (struct part) {528, "Disk drive", 10};
```

This statement resembles a declaration containing an initializer, but it's not the same—initializers can appear only in declarations, not in statements such as this one.

In general, a compound literal consists of a type name within parentheses, followed by a set of values enclosed by braces. In the case of a compound literal that represents a structure, the type name can be a structure tag preceded by the word `struct`—as in our examples—or a `typedef` name. A compound literal may contain designators, just like a designated initializer:

```
print_part((struct part) { .on_hand = 10,
                           .name = "Disk drive",
                           .number = 528});
```

A compound literal may fail to provide full initialization, in which case any uninitialized members default to zero.

16.3 Nested Arrays and Structures

Structures and arrays can be combined without restriction. Arrays may have structures as their elements, and structures may contain arrays and structures as members. We've already seen an example of an array nested inside a structure (the

name member of the part structure). Let's explore the other possibilities: structures whose members are structures and arrays whose elements are structures.

Nested Structures

Nesting one kind of structure inside another is often useful. For example, suppose that we've declared the following structure, which can store a person's first name, middle initial, and last name:

```
struct person_name {
    char first[FIRST_NAME_LEN+1];
    char middle_initial;
    char last[LAST_NAME_LEN+1];
};
```

We can use the person_name structure as part of a larger structure:

```
struct student {
    struct person_name name;
    int id, age;
    char sex;
} student1, student2;
```

Accessing student1's first name, middle initial, or last name requires two applications of the . operator:

```
strcpy(student1.name.first, "Fred");
```

One advantage of making name a structure (instead of having first, middle_initial, and last be members of the student structure) is that we can more easily treat names as units of data. For example, if we were to write a function that displays a name, we could pass it just one argument—a person_name structure—instead of three arguments:

```
display_name(student1.name);
```

Likewise, copying the information from a person_name structure to the name member of a student structure would take one assignment instead of three:

```
struct person_name new_name;
...
student1.name = new_name;
```

Arrays of Structures

One of the most common combinations of arrays and structures is an array whose elements are structures. An array of this kind can serve as a simple database. For example, the following array of part structures is capable of storing information about 100 parts:

```
struct part inventory[100];
```

To access one of the parts in the array, we'd use subscripting. To print the part stored in position *i*, for example, we could write

```
print_part(inventory[i]);
```

Accessing a member within a *part* structure requires a combination of subscripting and member selection. To assign 883 to the *number* member of *inventory[i]*, we could write

```
inventory[i].number = 883;
```

Accessing a single character in a part name requires subscripting (to select a particular part), followed by selection (to select the *name* member), followed by subscripting (to select a character within the part name). To change the name stored in *inventory[i]* to an empty string, we could write

```
inventory[i].name[0] = '\0';
```

Initializing an Array of Structures

Initializing an array of structures is done in much the same way as initializing a multidimensional array. Each structure has its own brace-enclosed initializer; the initializer for the array simply wraps another set of braces around the structure initializers.

One reason for initializing an array of structures is that we're planning to treat it as a database of information that won't change during program execution. For example, suppose that we're working on a program that will need access to the country codes used when making international telephone calls. First, we'll set up a structure that can store the name of a country along with its code:

```
struct dialing_code {
    char *country;
    int code;
};
```

Note that *country* is a pointer, not an array of characters. That could be a problem if we were planning to use *dialing_code* structures as variables, but we're not. When we initialize a *dialing_code* structure, *country* will end up pointing to a string literal.

Next, we'll declare an array of these structures and initialize it to contain the codes for some of the world's most populous nations:

```
const struct dialing_code country_codes[] =
{{"Argentina", 54}, {"Bangladesh", 880},
 {"Brazil", 55}, {"Burma (Myanmar)", 95},
 {"China", 86}, {"Colombia", 57},
 {"Congo, Dem. Rep. of", 243}, {"Egypt", 20},
 {"Ethiopia", 251}, {"France", 33},
 {"Germany", 49}, {"India", 91},
```

```

    {"Indonesia",           62}, {"Iran",          98},
    {"Italy",               39}, {"Japan",         81},
    {"Mexico",              52}, {"Nigeria",       234},
    {"Pakistan",             92}, {"Philippines",   63},
    {"Poland",                48}, {"Russia",        7},
    {"South Africa",        27}, {"South Korea",  82},
    {"Spain",                  34}, {"Sudan",         249},
    {"Thailand",                 66}, {"Turkey",        90},
    {"Ukraine",                380}, {"United Kingdom", 44},
    {"United States",            1}, {"Vietnam",       84} };

```

The inner braces around each structure value are optional. As a matter of style, however, I prefer not to omit them.

C99

Because arrays of structures (and structures containing arrays) are so common, C99's designated initializers allow an item to have more than one designator. Suppose that we want to initialize the `inventory` array to contain a single part. The part number is 528 and the quantity on hand is 10, but the name is to be left empty for now:

```

struct part inventory[100] =
{ [0].number = 528, [0].on_hand = 10, [0].name[0] = '\0' };

```

The first two items in the list use two designators (one to select array element 0—a part structure—and one to select a member within the structure). The last item uses three designators: one to select an array element, one to select the `name` member within that element, and one to select element 0 of `name`.

PROGRAM Maintaining a Parts Database

To illustrate how nested arrays and structures are used in practice, we'll now develop a fairly long program that maintains a database of information about parts stored in a warehouse. The program is built around an array of structures, with each structure containing information—part number, name, and quantity—about one part. Our program will support the following operations:

- **Add a new part number, part name, and initial quantity on hand.** The program must print an error message if the part is already in the database or if the database is full.
- **Given a part number, print the name of the part and the current quantity on hand.** The program must print an error message if the part number isn't in the database.
- **Given a part number, change the quantity on hand.** The program must print an error message if the part number isn't in the database.
- **Print a table showing all information in the database.** Parts must be displayed in the order in which they were entered.
- **Terminate program execution.**

We'll use the codes `i` (insert), `s` (search), `u` (update), `p` (print), and `q` (quit) to represent these operations. A session with the program might look like this:

```

Enter operation code: i
Enter part number: 528
Enter part name: Disk drive
Enter quantity on hand: 10

Enter operation code: s
Enter part number: 528
Part name: Disk drive
Quantity on hand: 10

Enter operation code: s
Enter part number: 914
Part not found.

Enter operation code: i
Enter part number: 914
Enter part name: Printer cable
Enter quantity on hand: 5

Enter operation code: u
Enter part number: 528
Enter change in quantity on hand: -2

Enter operation code: s
Enter part number: 528
Part name: Disk drive
Quantity on hand: 8

Enter operation code: p
Part Number      Part Name          Quantity on Hand
      528           Disk drive            8
      914           Printer cable        5

Enter operation code: q

```

The program will store information about each part in a structure. We'll limit the size of the database to 100 parts, making it possible to store the structures in an array, which I'll call `inventory`. (If this limit proves to be too small, we can always change it later.) To keep track of the number of parts currently stored in the array, we'll use a variable named `num_parts`.

Since this program is menu-driven, it's fairly easy to sketch the main loop:

```

for (;;) {
    prompt user to enter operation code;
    read code;
    switch (code) {
        case 'i': perform insert operation; break;
        case 's': perform search operation; break;
        case 'u': perform update operation; break;
        case 'p': perform print operation; break;
    }
}

```

```

        case 'q': terminate program;
        default: print error message;
    }
}

```

It will be convenient to have separate functions perform the insert, search, update, and print operations. Since these functions will all need access to `inventory` and `num_parts`, we might want to make these variables external. As an alternative, we could declare the variables inside `main`, and then pass them to the functions as arguments. From a design standpoint, it's usually better to make variables local to a function rather than making them external (see Section 10.2 if you've forgotten why). In this program, however, putting `inventory` and `num_parts` inside `main` would merely complicate matters.

For reasons that I'll explain later, I've decided to split the program into three files: `inventory.c`, which contains the bulk of the program; `readline.h`, which contains the prototype for the `read_line` function; and `readline.c`, which contains the definition of `read_line`. We'll discuss the latter two files later in this section. For now, let's concentrate on `inventory.c`.

```

inventory.c /* Maintains a parts database (array version) */

#include <stdio.h>
#include "readline.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 = 0; /* number of parts currently stored */

int find_part(int number);
void insert(void);
void search(void);
void update(void);
void print(void);

*****  

* main: Prompts the user to enter an operation code, *
*       then calls a function to perform the requested   *
*       action. Repeats until the user enters the      *
*       command 'q'. Prints an error message if the user *
*       enters an illegal code.                         *
*****/  

int main(void)
{
    char code;

```

```

        for (;;) {
            printf("Enter operation code: ");
            scanf(" %c", &code);
            while (getchar() != '\n') /* skips to end of line */
                ;
            switch (code) {
                case 'i': insert();
                            break;
                case 's': search();
                            break;
                case 'u': update();
                            break;
                case 'p': print();
                            break;
                case 'q': return 0;
                default: printf("Illegal code\n");
            }
            printf("\n");
        }

/************************************************************************
 * find_part: Looks up a part number in the inventory *
 *             array. Returns the array index if the part *
 *             number is found; otherwise, returns -1. *
 ************************************************************************/
int find_part(int number)
{
    int i;

    for (i = 0; i < num_parts; i++)
        if (inventory[i].number == number)
            return i;
    return -1;
}

/************************************************************************
 * insert: Prompts the user for information about a new *
 *          part and then inserts the part into the *
 *          database. Prints an error message and returns *
 *          prematurely if the part already exists or the *
 *          database is full. *
 ************************************************************************/
void insert(void)
{
    int part_number;

    if (num_parts == MAX_PARTS) {
        printf("Database is full; can't add more parts.\n");
        return;
    }

    printf("Enter part number: ");
    scanf("%d", &part_number);
}

```

```
if (find_part(part_number) >= 0) {
    printf("Part already exists.\n");
    return;
}

inventory[num_parts].number = part_number;
printf("Enter part name: ");
read_line(inventory[num_parts].name, NAME_LEN);
printf("Enter quantity on hand: ");
scanf("%d", &inventory[num_parts].on_hand);
num_parts++;
}

/*****************
 * search: Prompts the user to enter a part number, then *
 *          looks up the part in the database. If the part *
 *          exists, prints the name and quantity on hand;   *
 *          if not, prints an error message.                 *
 *****************/
void search(void)
{
    int i, number;

    printf("Enter part number: ");
    scanf("%d", &number);
    i = find_part(number);
    if (i >= 0) {
        printf("Part name: %s\n", inventory[i].name);
        printf("Quantity on hand: %d\n", inventory[i].on_hand);
    } else
        printf("Part not found.\n");
}

/*****************
 * update: Prompts the user to enter a part number.      *
 *          Prints an error message if the part doesn't   *
 *          exist; otherwise, prompts the user to enter    *
 *          change in quantity on hand and updates the   *
 *          database.                                     *
 *****************/
void update(void)
{
    int i, number, change;

    printf("Enter part number: ");
    scanf("%d", &number);
    i = find_part(number);
    if (i >= 0) {
        printf("Enter change in quantity on hand: ");
        scanf("%d", &change);
        inventory[i].on_hand += change;
    } else
        printf("Part not found.\n");
}
```

```
*****
 * print: Prints a listing of all parts in the database,
 *         showing the part number, part name, and
 *         quantity on hand. Parts are printed in the
 *         order in which they were entered into the
 *         database.
*****
void print(void)
{
    int i;

    printf("Part Number      Part Name
           "Quantity on Hand\n");
    for (i = 0; i < num_parts; i++)
        printf("%7d      %-25s%11d\n", inventory[i].number,
               inventory[i].name, inventory[i].on_hand);
}
```

In the main function, the format string " %c" allows `scanf` to skip over white space before reading the operation code. The space in the format string is crucial; without it, `scanf` would sometimes read the new-line character that terminated a previous line of input.

The program contains one function, `find_part`, that isn't called from `main`. This "helper" function helps us avoid redundant code and simplify the more important functions. By calling `find_part`, the `insert`, `search`, and `update` functions can locate a part in the database (or simply determine if the part exists).

There's just one detail left: the `read_line` function, which the program uses to read the part name. Section 13.3 discussed the issues that are involved in writing such a function. Unfortunately, the version of `read_line` in that section won't work properly in the current program. Consider what happens when the user inserts a part:

```
Enter part number: 528
Enter part name: Disk drive
```

The user presses the Enter key after entering the part number and again after entering the part name, each time leaving an invisible new-line character that the program must read. For the sake of discussion, let's pretend that these characters are visible:

```
Enter part number: 528 
Enter part name: Disk drive 
```

When we call `scanf` to read the part number, it consumes the 5, 2, and 8, but leaves the character unread. If we try to read the part name using our original `read_line` function, it will encounter the character immediately and stop reading. This problem is common when numerical input is followed by character input. Our solution will be to write a version of `read_line` that skips white-

space characters before it begins storing characters. Not only will this solve the new-line problem, but it also allows us to avoid storing any blanks that precede the part name.

Since `read_line` is unrelated to the other functions in `inventory.c`, and since it's potentially reusable in other programs, I've decided to separate it from `inventory.c`. The prototype for `read_line` will go in the `readline.h` header file:

```
readline.h #ifndef READLINE_H
#define READLINE_H

/*********************  

 * read_line: Skips leading white-space characters, then *  

 *           reads the remainder of the input line and *  

 *           stores it in str. Truncates the line if its *  

 *           length exceeds n. Returns the number of *  

 *           characters stored.  

 ****/  

int read_line(char str[], int n);

#endif
```

We'll put the definition of `read_line` in the `readline.c` file:

```
readline.c #include <ctype.h>
#include <stdio.h>
#include "readline.h"

int read_line(char str[], int n)
{
    int ch, i = 0;

    while (isspace(ch = getchar()))
        ;
    while (ch != '\n' && ch != EOF) {
        if (i < n)
            str[i++] = ch;
        ch = getchar();
    }
    str[i] = '\0';
    return i;
}
```

The expression

```
isspace(ch = getchar())
```

isspace function ▶ 23.5

controls the first `while` statement. This expression calls `getchar` to read a character, stores the character into `ch`, and then uses the `isspace` function to test whether `ch` is a white-space character. If not, the loop terminates with `ch` containing a character that's not white space. Section 15.3 explains why `ch` has type `int` instead of `char` and why it's good to test for EOF.

16.4 Unions

A *union*, like a structure, consists of one or more members, possibly of different types. However, the compiler allocates only enough space for the largest of the members, which overlay each other within this space. As a result, assigning a new value to one member alters the values of the other members as well.

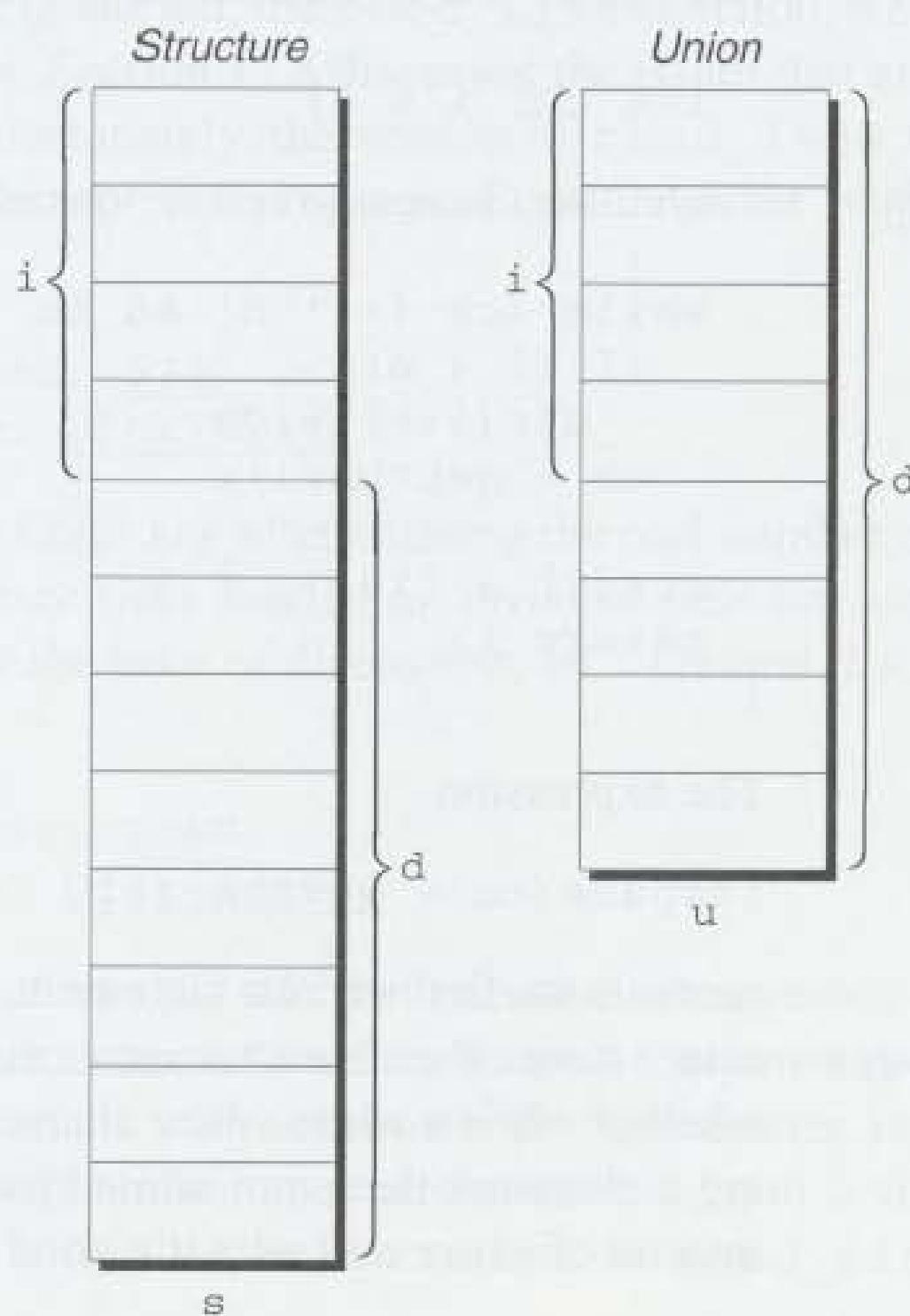
To illustrate the basic properties of unions, let's declare a union variable, *u*, with two members:

```
union {
    int i;
    double d;
} u;
```

Notice how the declaration of a union closely resembles a structure declaration:

```
struct {
    int i;
    double d;
} s;
```

In fact, the structure *s* and the union *u* differ in just one way: the members of *s* are stored at *different* addresses in memory, while the members of *u* are stored at the *same* address. Here's what *s* and *u* will look like in memory (assuming that *int* values require four bytes and *double* values take eight bytes):



In the `s` structure, `i` and `d` occupy different memory locations; the total size of `s` is 12 bytes. In the `u` union, `i` and `d` overlap (`i` is really the first four bytes of `d`), so `u` occupies only eight bytes. Also, `i` and `d` have the same address.

Members of a union are accessed in the same way as members of a structure. To store the number 82 in the `i` member of `u`, we would write

```
u.i = 82;
```

To store the value 74.8 in the `d` member, we would write

```
u.d = 74.8;
```

Since the compiler overlays storage for the members of a union, changing one member alters any value previously stored in any of the other members. Thus, if we store a value in `u.d`, any value previously stored in `u.i` will be lost. (If we examine the value of `u.i`, it will appear to be meaningless.) Similarly, changing `u.i` corrupts `u.d`. Because of this property, we can think of `u` as a place to store either `i` or `d`, not both. (The structure `s` allows us to store `i` and `d`.)

The properties of unions are almost identical to the properties of structures. We can declare union tags and union types in the same way we declare structure tags and types. Like structures, unions can be copied using the `=` operator, passed to functions, and returned by functions.

Unions can even be initialized in a manner similar to structures. However, only the first member of a union can be given an initial value. For example, we can initialize the `i` member of `u` to 0 in the following way:

```
union {
    int i;
    double d;
} u = {0};
```

Notice the presence of the braces, which are required. The expression inside the braces must be constant. (The rules are slightly different in C99, as we'll see in Section 18.5.)

C99

Designated initializers, a C99 feature that we've previously discussed in the context of arrays and structures, can also be used with unions. A designated initializer allows us to specify which member of a union should be initialized. For example, we can initialize the `d` member of `u` as follows:

```
union {
    int i;
    double d;
} u = {.d = 10.0};
```

Only one member can be initialized, but it doesn't have to be the first one.

There are several applications for unions. We'll discuss two of these now. Another application—viewing storage in different ways—is highly machine-dependent, so I'll postpone it until Section 20.3.

Using Unions to Save Space

We'll often use unions as a way to save space in structures. Suppose that we're designing a structure that will contain information about an item that's sold through a gift catalog. The catalog carries only three kinds of merchandise: books, mugs, and shirts. Each item has a stock number and a price, as well as other information that depends on the type of the item:

Books: Title, author, number of pages

Mugs: Design

Shirts: Design, colors available, sizes available

Our first design attempt might result in the following structure:

```
struct catalog_item {
    int stock_number;
    double price;
    int item_type;
    char title[TITLE_LEN+1];
    char author[AUTHOR_LEN+1];
    int num_pages;
    char design[DESIGN_LEN+1];
    int colors;
    int sizes;
};
```

The `item_type` member would have one of the values `BOOK`, `MUG`, or `SHIRT`. The `colors` and `sizes` members would store encoded combinations of colors and sizes.

Although this structure is perfectly usable, it wastes space, since only part of the information in the structure is common to all items in the catalog. If an item is a book, for example, there's no need to store design, colors, and sizes. By putting a union inside the `catalog_item` structure, we can reduce the space required by the structure. The members of the union will be structures, each containing the data that's needed for a particular kind of catalog item:

```
struct catalog_item {
    int stock_number;
    double price;
    int item_type;
    union {
        struct {
            char title[TITLE_LEN+1];
            char author[AUTHOR_LEN+1];
            int num_pages;
        } book;
        struct {
            char design[DESIGN_LEN+1];
        } mug;
```

```

struct {
    char design[DESIGN_LEN+1];
    int colors;
    int sizes;
} shirt;
} item;
};

```

Notice that the union (named `item`) is a member of the `catalog_item` structure, and the `book`, `mug`, and `shirt` structures are members of `item`. If `c` is a `catalog_item` structure that represents a book, we can print the book's title in the following way:

```
printf("%s", c.item.book.title);
```

As this example shows, accessing a union that's nested inside a structure can be awkward: to locate a book title, we had to specify the name of a structure (`c`), the name of the union member of the structure (`item`), the name of a structure member of the union (`book`), and then the name of a member of that structure (`title`).

We can use the `catalog_item` structure to illustrate an interesting aspect of unions. Normally, it's not a good idea to store a value into one member of a union and then access the data through a different member, because assigning to one member of a union causes the values of the other members to be undefined. However, the C standard mentions a special case: two or more of the members of the union are structures, and the structures begin with one or more matching members. (These members need to be in the same order and have compatible types, but need not have the same name.) If one of the structures is currently valid, then the matching members in the other structures will also be valid.

Consider the union embedded in the `catalog_item` structure. It contains three structures as members, two of which (`mug` and `shirt`) begin with a matching member (`design`). Now, suppose that we assign a value to one of the `design` members:

```
strcpy(c.item.mug.design, "Cats");
```

The `design` member in the other structure will be defined and have the same value:

```
printf("%s", c.item.shirt.design); /* prints "Cats" */
```

Using Unions to Build Mixed Data Structures

Unions have another important application: creating data structures that contain a mixture of data of different types. Let's say that we need an array whose elements are a mixture of `int` and `double` values. Since the elements of an array must be of the same type, it seems impossible to create such an array. Using unions, though, it's relatively easy. First, we define a union type whose members represent the different kinds of data to be stored in the array:

```
typedef union {
    int i;
    double d;
} Number;
```

Next, we create an array whose elements are `Number` values:

```
Number number_array[1000];
```

Each element of `number_array` is a `Number` union. A `Number` union can store either an `int` value or a `double` value, making it possible to store a mixture of `int` and `double` values in `number_array`. For example, suppose that we want element 0 of `number_array` to store 5, while element 1 stores 8.395. The following assignments will have the desired effect:

```
number_array[0].i = 5;
number_array[1].d = 8.395;
```

Adding a “Tag Field” to a Union

Unions suffer from a major problem: there’s no easy way to tell which member of a union was last changed and therefore contains a meaningful value. Consider the problem of writing a function that displays the value currently stored in a `Number` union. This function might have the following outline:

```
void print_number(Number n)
{
    if (n contains an integer)
        printf("%d", n.i);
    else
        printf("%g", n.d);
}
```

Unfortunately, there’s no way for `print_number` to determine whether `n` contains an integer or a floating-point number.

In order to keep track of this information, we can embed the union within a structure that has one other member: a “tag field” or “discriminant,” whose purpose is to remind us what’s currently stored in the union. In the `catalog_item` structure discussed earlier in this section, `item_type` served this purpose.

Let’s convert the `Number` type into a structure with an embedded union:

```
#define INT_KIND 0
#define DOUBLE_KIND 1

typedef struct {
    int kind; /* tag field */
    union {
        int i;
        double d;
    } u;
} Number;
```

`Number` has two members, `kind` and `u`. The value of `kind` will be either `INT_KIND` or `DOUBLE_KIND`.

Each time we assign a value to a member of `u`, we'll also change `kind` to remind us which member of `u` we modified. For example, if `n` is a `Number` variable, an assignment to the `i` member of `u` would have the following appearance:

```
n.kind = INT_KIND;
n.u.i = 82;
```

Notice that assigning to `i` requires that we first select the `u` member of `n`, then the `i` member of `u`.

When we need to retrieve the number stored in a `Number` variable, `kind` will tell us which member of the union was the last to be assigned a value. The `print_number` function can take advantage of this capability:

```
void print_number(Number n)
{
    if (n.kind == INT_KIND)
        printf("%d", n.u.i);
    else
        printf("%g", n.u.d);
}
```



It's the program's responsibility to change the tag field each time an assignment is made to a member of the union.

16.5 Enumerations

In many programs, we'll need variables that have only a small set of meaningful values. A Boolean variable, for example, should have only two possible values: "true" and "false." A variable that stores the suit of a playing card should have only four potential values: "clubs," "diamonds," "hearts," and "spades." The obvious way to deal with such a variable is to declare it as an integer and have a set of codes that represent the possible values of the variable:

```
int s; /* s will store a suit */
...
s = 2; /* 2 represents "hearts" */
```

Although this technique works, it leaves much to be desired. Someone reading the program can't tell that `s` has only four possible values, and the significance of 2 isn't immediately apparent.

Using macros to define a suit "type" and names for the various suits is a step in the right direction:

```
#define SUIT      int
#define CLUBS     0
#define DIAMONDS  1
#define HEARTS    2
#define SPADES    3
```

Our previous example now becomes easier to read:

```
SUIT s;
...
s = HEARTS;
```

This technique is an improvement, but it's still not the best solution. There's no indication to someone reading the program that the macros represent values of the same "type." If the number of possible values is more than a few, defining a separate macro for each will be tedious. Moreover, the names we've defined—CLUBS, DIAMONDS, HEARTS, and SPADES—will be removed by the preprocessor, so they won't be available during debugging.

C provides a special kind of type designed specifically for variables that have a small number of possible values. An *enumerated type* is a type whose values are listed ("enumerated") by the programmer, who must create a name (an *enumeration constant*) for each of the values. The following example enumerates the values (CLUBS, DIAMONDS, HEARTS, and SPADES) that can be assigned to the variables `s1` and `s2`:

```
enum {CLUBS, DIAMONDS, HEARTS, SPADES} s1, s2;
```

Although enumerations have little in common with structures and unions, they're declared in a similar way. Unlike the members of a structure or union, however, the names of enumeration constants must be different from other identifiers declared in the enclosing scope.

Enumeration constants are similar to constants created with the `#define` directive, but they're not equivalent. For one thing, enumeration constants are subject to C's scope rules: if an enumeration is declared inside a function, its constants won't be visible outside the function.

Enumeration Tags and Type Names

We'll often need to create names for enumerations, for the same reasons that we name structures and unions. As with structures and unions, there are two ways to name an enumeration: by declaring a tag or by using `typedef` to create a genuine type name.

Enumeration tags resemble structure and union tags. To define the tag `suit`, for example, we could write

```
enum suit {CLUBS, DIAMONDS, HEARTS, SPADES};
```

`suit` variables would be declared in the following way:

```
enum suit s1, s2;
```

As an alternative, we could use `typedef` to make `Suit` a type name:

```
typedef enum {CLUBS, DIAMONDS, HEARTS, SPADES} Suit;
Suit s1, s2;
```

In C89, using `typedef` to name an enumeration is an excellent way to create a Boolean type:

```
typedef enum {FALSE, TRUE} Bool;
```

C99 has a built-in Boolean type, of course, so there's no need for a C99 programmer to define a `Bool` type in this way.

Enumerations as Integers

Behind the scenes, C treats enumeration variables and constants as integers. By default, the compiler assigns the integers 0, 1, 2, ... to the constants in a particular enumeration. In our `suit` enumeration, for example, CLUBS, DIAMONDS, HEARTS, and SPADES represent 0, 1, 2, and 3, respectively.

We're free to choose different values for enumeration constants if we like. Let's say that we want CLUBS, DIAMONDS, HEARTS, and SPADES to stand for 1, 2, 3, and 4. We can specify these numbers when declaring the enumeration:

```
enum suit {CLUBS = 1, DIAMONDS = 2, HEARTS = 3, SPADES = 4};
```

The values of enumeration constants may be arbitrary integers, listed in no particular order:

```
enum dept {RESEARCH = 20, PRODUCTION = 10, SALES = 25};
```

It's even legal for two or more enumeration constants to have the same value.

When no value is specified for an enumeration constant, its value is one greater than the value of the previous constant. (The first enumeration constant has the value 0 by default.) In the following enumeration, BLACK has the value 0, LT_GRAY is 7, DK_GRAY is 8, and WHITE is 15:

```
enum EGA_colors {BLACK, LT_GRAY = 7, DK_GRAY, WHITE = 15};
```

Since enumeration values are nothing but thinly disguised integers, C allows us to mix them with ordinary integers:

```
int i;
enum {CLUBS, DIAMONDS, HEARTS, SPADES} s;

i = DIAMONDS;      /* i is now 1          */
s = 0;              /* s is now 0 (CLUBS)   */
s++;               /* s is now 1 (DIAMONDS) */
i = s + 2;          /* i is now 3          */
```

The compiler treats `s` as a variable of some integer type; CLUBS, DIAMONDS, HEARTS, and SPADES are just names for the integers 0, 1, 2, and 3.



Although it's convenient to be able to use an enumeration value as an integer, it's dangerous to use an integer as an enumeration value. For example, we might accidentally store the number 4—which doesn't correspond to any suit—into `s`.

Using Enumerations to Declare “Tag Fields”

Enumerations are perfect for solving a problem that we encountered in Section 16.4: determining which member of a union was the last to be assigned a value. In the `Number` structure, for example, we can make the `kind` member an enumeration instead of an `int`:

```
typedef struct {
    enum {INT_KIND, DOUBLE_KIND} kind;
    union {
        int i;
        double d;
    } u;
} Number;
```

The new structure is used in exactly the same way as the old one. The advantages are that we've done away with the `INT_KIND` and `DOUBLE_KIND` macros (they're now enumeration constants), and we've clarified the meaning of `kind`—it's now obvious that `kind` has only two possible values: `INT_KIND` and `DOUBLE_KIND`.

Q & A

Q: When I tried using the `sizeof` operator to determine the number of bytes in a structure, I got a number that was larger than the sizes of the members added together. How can this be?

A: Let's look at an example:

```
struct {
    char a;
    int b;
} s;
```

If `char` values occupy one byte and `int` values occupy four bytes, how large is `s`? The obvious answer—five bytes—may not be the correct one. Some computers require that the address of certain data items be a multiple of some number of bytes (typically two, four, or eight, depending on the item's type). To satisfy this requirement, a compiler will “align” the members of a structure by leaving “holes” (unused bytes) between adjacent members. If we assume that data items must

begin on a multiple of four bytes, the `a` member of the `s` structure will be followed by a three-byte hole. As a result, `sizeof(s)` will be 8.

By the way, a structure can have a hole at the end, as well as holes between members. For example, the structure

```
struct {
    int a;
    char b;
} s;
```

might have a three-byte hole after the `b` member.

Q: Can there be a “hole” at the beginning of a structure?

A: No. The C standard specifies that holes are allowed only *between* members or *after* the last member. One consequence is that a pointer to the first member of a structure is guaranteed to be the same as a pointer to the entire structure. (Note, however, that the two pointers won’t have the same type.)

Q: Why isn’t it legal to use the == operator to test whether two structures are equal? [p. 382]

A: This operation was left out of C because there’s no way to implement it that would be consistent with the language’s philosophy. Comparing structure members one by one would be too inefficient. Comparing all bytes in the structures would be better (many computers have special instructions that can perform such a comparison rapidly). If the structures contain holes, however, comparing bytes could yield an incorrect answer; even if corresponding members have identical values, leftover data stored in the holes might be different. The problem could be solved by having the compiler ensure that holes always contain the same value (zero, say). Initializing holes would impose a performance penalty on all programs that use structures, however, so it’s not feasible.

Q: Why does C provide two ways to name structure types (tags and `typedef` names)? [p. 382]

A: C originally lacked `typedef`, so tags were the only technique available for naming structure types. When `typedef` was added, it was too late to remove tags. Besides, a tag is still necessary when a member of a structure points to a structure of the same type (see the `node` structure of Section 17.5).

Q: Can a structure have both a tag and a `typedef` name? [p. 384]

A: Yes. In fact, the tag and the `typedef` name can even be the same, although that’s not required:

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

Q: How can I share a structure type among several files in a program?

A: Put a declaration of the structure tag (or a `typedef`, if you prefer) in a header file, then include the header file where the structure is needed. To share the `part` structure, for example, we'd put the following lines in a header file:

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

Notice that we're declaring only the structure *tag*, not variables of this type.

Incidentally, a header file that contains a declaration of a structure tag or structure type may need protection against multiple inclusion. Declaring a tag or `typedef` name twice in the same file is an error. Similar remarks apply to unions and enumerations.

Q: If I include the declaration of the part structure into two different files, will part variables in one file be of the same type as part variables in the other file?

A: Technically, no. However, the C standard says that the `part` variables in one file have a type that's compatible with the type of the `part` variables in the other file. Variables with compatible types can be assigned to each other, so there's little practical difference between types being "compatible" and being "the same."

C99 The rules for structure compatibility in C89 and C99 are slightly different. In C89, structures defined in different files are compatible if their members have the same names and appear in the same order, with corresponding members having compatible types. C99 goes one step further: it requires that either both structures have the same tag or neither has a tag.

Similar compatibility rules apply to unions and enumerations (with the same difference between C89 and C99).

Q: Is it legal to have a pointer to a compound literal?

A: Yes. Consider the `print_part` function of Section 16.2. Currently, the parameter to this function is a `part` structure. The function would be more efficient if it were modified to accept a *pointer* to a `part` structure instead. Using the function to print a compound literal would then be done by prefixing the argument with the & (address) operator:

```
print_part(&(struct part) {528, "Disk drive", 10});
```

Q: Allowing a pointer to a compound literal would seem to make it possible to modify the literal. Is that the case?

C99 A: Yes. Compound literals are lvalues that can be modified, although doing so is rare.

Q: I saw a program in which the last constant in an enumeration was followed by a comma, like this:

```
enum gray_values {
    BLACK = 0,
    DARK_GRAY = 64,
    GRAY = 128,
    LIGHT_GRAY = 192,
};
```

Is this practice legal?

- A: This practice is indeed legal in C99 (and is supported by some pre-C99 compilers as well). Allowing a “trailing comma” makes enumerations easier to modify, because we can add a constant to the end of an enumeration without changing existing lines of code. For example, we might want to add WHITE to our enumeration:

```
enum gray_values {
    BLACK = 0,
    DARK_GRAY = 64,
    GRAY = 128,
    LIGHT_GRAY = 192,
    WHITE = 255,
};
```

The comma after the definition of LIGHT_GRAY makes it easy to add WHITE to the end of the list.

- C99 One reason for this change is that C89 allows trailing commas in initializers, so it seemed inconsistent not to allow the same flexibility in enumerations. Incidentally, C99 also allows trailing commas in compound literals.

Q: Can the values of an enumerated type be used as subscripts?

- A: Yes, indeed. They are integers and have—by default—values that start at 0 and count upward, so they make great subscripts. In C99, moreover, enumeration constants can be used as subscripts in designated initializers. Here’s an example:

```
enum weekdays {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY};
const char *daily_specials[] = {
    [MONDAY] = "Beef ravioli",
    [TUESDAY] = "BLTs",
    [WEDNESDAY] = "Pizza",
    [THURSDAY] = "Chicken fajitas",
    [FRIDAY] = "Macaroni and cheese"
};
```

Exercises

Section 16.1

- In the following declarations, the x and y structures have members named x and y:

```
struct { int x, y; } x;
struct { int x, y; } y;
```

Are these declarations legal on an individual basis? Could both declarations appear as shown in a program? Justify your answer.

- W 2. (a) Declare structure variables named `c1`, `c2`, and `c3`, each having members `real` and `imaginary` of type `double`.
 (b) Modify the declaration in part (a) so that `c1`'s members initially have the values 0.0 and 1.0, while `c2`'s members are 1.0 and 0.0 initially. (`c3` is not initialized.)
 (c) Write statements that copy the members of `c2` into `c1`. Can this be done in one statement, or does it require two?
 (d) Write statements that add the corresponding members of `c1` and `c2`, storing the result in `c3`.

Section 16.2

3. (a) Show how to declare a tag named `complex` for a structure with two members, `real` and `imaginary`, of type `double`.
 (b) Use the `complex` tag to declare variables named `c1`, `c2`, and `c3`.
 (c) Write a function named `make_complex` that stores its two arguments (both of type `double`) in a `complex` structure, then returns the structure.
 (d) Write a function named `add_complex` that adds the corresponding members of its arguments (both `complex` structures), then returns the result (another `complex` structure).
- W 4. Repeat Exercise 3, but this time using a *type* named `Complex`.
5. Write the following functions, assuming that the `date` structure contains three members: `month`, `day`, and `year` (all of type `int`).
 (a) `int day_of_year(struct date d);`
 Returns the day of the year (an integer between 1 and 366) that corresponds to the date `d`.
 (b) `int compare_dates(struct date d1, struct date d2);`
 Returns -1 if `d1` is an earlier date than `d2`, +1 if `d1` is a later date than `d2`, and 0 if `d1` and `d2` are the same.
6. Write the following function, assuming that the `time` structure contains three members: `hours`, `minutes`, and `seconds` (all of type `int`).
`struct time split_time(long total_seconds);`
`total_seconds` is a time represented as the number of seconds since midnight. The function returns a structure containing the equivalent time in hours (0–23), minutes (0–59), and seconds (0–59).
7. Assume that the `fraction` structure contains two members: `numerator` and `denominator` (both of type `int`). Write functions that perform the following operations on fractions:
 (a) Reduce the fraction `f` to lowest terms. *Hint:* To reduce a fraction to lowest terms, first compute the greatest common divisor (GCD) of the numerator and denominator. Then divide both the numerator and denominator by the GCD.
 (b) Add the fractions `f1` and `f2`.
 (c) Subtract the fraction `f2` from the fraction `f1`.
 (d) Multiply the fractions `f1` and `f2`.
 (e) Divide the fraction `f1` by the fraction `f2`.

The fractions `f`, `f1`, and `f2` will be arguments of type `struct fraction`; each function will return a value of type `struct fraction`. The fractions returned by the functions in parts (b)–(e) should be reduced to lowest terms. *Hint:* You may use the function from part (a) to help write the functions in parts (b)–(e).

8. Let `color` be the following structure:

```
struct color {
    int red;
    int green;
    int blue;
};
```

(a) Write a declaration for a `const` variable named `MAGENTA` of type `struct color` whose members have the values 255, 0, and 255, respectively.

(b) (C99) Repeat part (a), but use a designated initializer that doesn't specify the value of `green`, allowing it to default to 0.

9. Write the following functions. (The `color` structure is defined in Exercise 8.)

(a) `struct color make_color(int red, int green, int blue);`

Returns a `color` structure containing the specified red, green, and blue values. If any argument is less than zero, the corresponding member of the structure will contain zero instead. If any argument is greater than 255, the corresponding member of the structure will contain 255.

(b) `int getRed(struct color c);`

Returns the value of `c`'s `red` member.

(c) `bool equal_color(struct color color1, struct color color2);`

Returns `true` if the corresponding members of `color1` and `color2` are equal.

(d) `struct color brighter(struct color c);`

Returns a `color` structure that represents a brighter version of the color `c`. The structure is identical to `c`, except that each member has been divided by 0.7 (with the result truncated to an integer). However, there are three special cases: (1) If all members of `c` are zero, the function returns a color whose members all have the value 3. (2) If any member of `c` is greater than 0 but less than 3, it is replaced by 3 before the division by 0.7. (3) If dividing by 0.7 causes a member to exceed 255, it is reduced to 255.

(e) `struct color darker(struct color c);`

Returns a `color` structure that represents a darker version of the color `c`. The structure is identical to `c`, except that each member has been multiplied by 0.7 (with the result truncated to an integer).

Section 16.3

10. The following structures are designed to store information about objects on a graphics screen:

```
struct point { int x, y; };
struct rectangle { struct point upper_left, lower_right; };
```

A `point` structure stores the `x` and `y` coordinates of a point on the screen. A `rectangle` structure stores the coordinates of the upper left and lower right corners of a rectangle. Write functions that perform the following operations on a `rectangle` structure `r` passed as an argument:

(a) Compute the area of `r`.

(b) Compute the center of `r`, returning it as a `point` value. If either the `x` or `y` coordinate of the center isn't an integer, store its truncated value in the `point` structure.

(c) Move `r` by `x` units in the `x` direction and `y` units in the `y` direction, returning the modified version of `r`. (`x` and `y` are additional arguments to the function.)

(d) Determine whether a point `p` lies within `r`, returning `true` or `false`. (`p` is an additional argument of type `struct point`.)

Section 16.4 **W** 11. Suppose that `s` is the following structure:

```
struct {
    double a;
    union {
        char b[4];
        double c;
        int d;
    } e;
    char f[4];
} s;
```

If `char` values occupy one byte, `int` values occupy four bytes, and `double` values occupy eight bytes, how much space will a C compiler allocate for `s`? (Assume that the compiler leaves no “holes” between members.)

12. Suppose that `u` is the following union:

```
union {
    double a;
    struct {
        char b[4];
        double c;
        int d;
    } e;
    char f[4];
} u;
```

If `char` values occupy one byte, `int` values occupy four bytes, and `double` values occupy eight bytes, how much space will a C compiler allocate for `u`? (Assume that the compiler leaves no “holes” between members.)

13. Suppose that `s` is the following structure (`point` is a structure tag declared in Exercise 10):

```
struct shape {
    int shape_kind;          /* RECTANGLE or CIRCLE */
    struct point center;    /* coordinates of center */
    union {
        struct {
            int height, width;
        } rectangle;
        struct {
            int radius;
        } circle;
    } u;
} s;
```

If the value of `shape_kind` is `RECTANGLE`, the `height` and `width` members store the dimensions of a rectangle. If the value of `shape_kind` is `CIRCLE`, the `radius` member stores the radius of a circle. Indicate which of the following statements are legal, and show how to repair the ones that aren’t:

- `s.shape_kind = RECTANGLE;`
- `s.center.x = 10;`
- `s.height = 25;`
- `s.u.rectangle.width = 8;`
- `s.u.circle = 5;`
- `s.u.radius = 5;`

- W 14. Let `shape` be the structure tag declared in Exercise 13. Write functions that perform the following operations on a `shape` structure `s` passed as an argument:
- Compute the area of `s`.
 - Move `s` by `x` units in the `x` direction and `y` units in the `y` direction, returning the modified version of `s`. (`x` and `y` are additional arguments to the function.)
 - Scale `s` by a factor of `c` (a `double` value), returning the modified version of `s`. (`c` is an additional argument to the function.)

Section 16.5

- W 15. (a) Declare a tag for an enumeration whose values represent the seven days of the week.
 (b) Use `typedef` to define a name for the enumeration of part (a).
16. Which of the following statements about enumeration constants are true?
- An enumeration constant may represent any integer specified by the programmer.
 - Enumeration constants have exactly the same properties as constants created using `#define`.
 - Enumeration constants have the values 0, 1, 2, ... by default.
 - All constants in an enumeration must have different values.
 - Enumeration constants may be used as integers in expressions.

- W 17. Suppose that `b` and `i` are declared as follows:

```
enum { FALSE, TRUE } b;
int i;
```

Which of the following statements are legal? Which ones are “safe” (always yield a meaningful result)?

- `b = FALSE;`
 - `b = i;`
 - `b++;`
 - `i = b;`
 - `i = 2 * b + 1;`
18. (a) Each square of a chessboard can hold one piece—a pawn, knight, bishop, rook, queen, or king—or it may be empty. Each piece is either black or white. Define two enumerated types: `Piece`, which has seven possible values (one of which is “empty”), and `Color`, which has two.
- (b) Using the types from part (a), define a structure type named `Square` that can store both the type of a piece and its color.
- (c) Using the `Square` type from part (b), declare an 8×8 array named `board` that can store the entire contents of a chessboard.
- (d) Add an initializer to the declaration in part (c) so that `board`’s initial value corresponds to the usual arrangement of pieces at the start of a chess game. A square that’s not occupied by a piece should have an “empty” piece value and the color black.
19. Declare a structure with the following members whose tag is `pinball_machine`:
- `name` – a string of up to 40 characters
 - `year` – an integer (representing the year of manufacture)
 - `type` – an enumeration with the values `EM` (electromechanical) and `SS` (solid state)
 - `players` – an integer (representing the maximum number of players)
20. Suppose that the `direction` variable is declared in the following way:
- ```
enum { NORTH, SOUTH, EAST, WEST } direction;
```

Let `x` and `y` be `int` variables. Write a switch statement that tests the value of `direction`, incrementing `x` if `direction` is EAST, decrementing `x` if `direction` is WEST, incrementing `y` if `direction` is SOUTH, and decrementing `y` if `direction` is NORTH.

21. What are the integer values of the enumeration constants in each of the following declarations?
  - (a) `enum {NUL, SOH, STX, ETX};`
  - (b) `enum {VT = 11, FF, CR};`
  - (c) `enum {SO = 14, SI, DLE, CAN = 24, EM};`
  - (d) `enum {ENQ = 45, ACK, BEL, LF = 37, ETB, ESC};`
22. Let `chess_pieces` be the following enumeration:
 

```
enum chess_pieces {KING, QUEEN, ROOK, BISHOP, KNIGHT, PAWN};
```

  - (a) Write a declaration (including an initializer) for a constant array of integers named `piece_value` that stores the numbers 200, 9, 5, 3, 3, and 1, representing the value of each chess piece, from king to pawn. (The king's value is actually infinite, since "capturing" the king (checkmate) ends the game, but some chess-playing software assigns the king a large value such as 200.)
  - (b) (C99) Repeat part (a), but use a designated initializer to initialize the array. Use the enumeration constants in `chess_pieces` as subscripts in the designators. (*Hint:* See the last question in Q&A for an example.)

## Programming Projects

- W 1. Write a program that asks the user to enter an international dialing code and then looks it up in the `country_codes` array (see Section 16.3). If it finds the code, the program should display the name of the corresponding country; if not, the program should print an error message.
- 2. Modify the `inventory.c` program of Section 16.3 so that the `p` (print) operation displays the parts sorted by part number.
- W 3. Modify the `inventory.c` program of Section 16.3 by making `inventory` and `num_parts` local to the `main` function.
- 4. Modify the `inventory.c` program of Section 16.3 by adding a `price` member to the `part` structure. The `insert` function should ask the user for the price of a new item. The `search` and `print` functions should display the price. Add a new command that allows the user to change the price of a part.
- 5. Modify Programming Project 8 from Chapter 5 so that the times are stored in a single array. The elements of the array will be structures, each containing a departure time and the corresponding arrival time. (Each time will be an integer, representing the number of minutes since midnight.) The program will use a loop to search the array for the departure time closest to the time entered by the user.
- 6. Modify Programming Project 9 from Chapter 5 so that each date entered by the user is stored in a `date` structure (see Exercise 5). Incorporate the `compare_dates` function of Exercise 5 into your program.

# 17 Advanced Uses of Pointers

*One can only display complex information in the mind. Like seeing, movement or flow or alteration of view is more important than the static picture, no matter how lovely.*

In previous chapters, we've seen two important uses of pointers. Chapter 11 showed how using a pointer to a variable as a function argument allows the function to modify the variable. Chapter 12 showed how to process arrays by performing arithmetic on pointers to array elements. This chapter completes our coverage of pointers by examining two additional applications: dynamic storage allocation and pointers to functions.

Using dynamic storage allocation, a program can obtain blocks of memory as needed during execution. Section 17.1 explains the basics of dynamic storage allocation. Section 17.2 discusses dynamically allocated strings, which provide more flexibility than ordinary character arrays. Section 17.3 covers dynamic storage allocation for arrays in general. Section 17.4 deals with the issue of storage deallocation—releasing blocks of dynamically allocated memory when they're no longer needed.

Dynamically allocated structures play a big role in C programming, since they can be linked together to form lists, trees, and other highly flexible data structures. Section 17.5 focuses on linked lists, the most fundamental linked data structure. One of the issues that arises in this section—the concept of a “pointer to a pointer”—is important enough to warrant a section of its own (Section 17.6).

Section 17.7 introduces pointers to functions, a surprisingly useful concept. Some of C's most powerful library functions expect function pointers as arguments. We'll examine one of these functions, `qsort`, which is capable of sorting any array.

The last two sections discuss pointer-related features that first appeared in C99: restricted pointers (Section 17.8) and flexible array members (Section 17.9). These features are primarily of interest to advanced C programmers, so both sections can be safely be skipped by the beginner.

## 17.1 Dynamic Storage Allocation

variable-length arrays ▶ 8.3

C's data structures are normally fixed in size. For example, the number of elements in an array is fixed once the program has been compiled. (In C99, the length of a variable-length array is determined at run time, but it remains fixed for the rest of the array's lifetime.) Fixed-size data structures can be a problem, since we're forced to choose their sizes when writing a program; we can't change the sizes without modifying the program and compiling it again.

Consider the inventory program of Section 16.3, which allows the user to add parts to a database. The database is stored in an array of length 100. To enlarge the capacity of the database, we can increase the size of the array and recompile the program. But no matter how large we make the array, there's always the possibility that it will fill up. Fortunately, all is not lost. C supports *dynamic storage allocation*: the ability to allocate storage during program execution. Using dynamic storage allocation, we can design data structures that grow (and shrink) as needed.

Although it's available for all types of data, dynamic storage allocation is used most often for strings, arrays, and structures. Dynamically allocated structures are of particular interest, since we can link them together to form lists, trees, and other data structures.

### Memory Allocation Functions

<stdlib.h> header ▶ 26.2

To allocate storage dynamically, we'll need to call one of the three memory allocation functions declared in the `<stdlib.h>` header:

- `malloc`—Allocates a block of memory but doesn't initialize it.
- `calloc`—Allocates a block of memory and clears it.
- `realloc`—Resizes a previously allocated block of memory.

Of the three, `malloc` is the most used. It's more efficient than `calloc`, since it doesn't have to clear the memory block that it allocates.

When we call a memory allocation function to request a block of memory, the function has no idea what type of data we're planning to store in the block, so it can't return a pointer to an ordinary type such as `int` or `char`. Instead, the function returns a value of type `void *`. A `void *` value is a "generic" pointer—essentially, just a memory address.

### Null Pointers

When a memory allocation function is called, there's always a possibility that it won't be able to locate a block of memory large enough to satisfy our request. If

that should happen, the function will return a **null pointer**. A null pointer is a “pointer to nothing”—a special value that can be distinguished from all valid pointers. After we’ve stored the function’s return value in a pointer variable, we must test to see if it’s a null pointer.



It’s the programmer’s responsibility to test the return value of any memory allocation function and take appropriate action if it’s a null pointer. The effect of attempting to access memory through a null pointer is undefined; the program may crash or behave unpredictably.

**Q&A**

The null pointer is represented by a macro named NULL, so we can test malloc’s return value in the following way:

```
p = malloc(10000);
if (p == NULL) {
 /* allocation failed; take appropriate action */
}
```

Some programmers combine the call of malloc with the NULL test:

```
if ((p = malloc(10000)) == NULL) {
 /* allocation failed; take appropriate action */
}
```

**C99**

The NULL macro is defined in six headers: `<locale.h>`, `<stddef.h>`, `<stdio.h>`, `<stdlib.h>`, `<string.h>`, and `<time.h>`. (The C99 header `<wchar.h>` also defines NULL.) As long as one of these headers is included in a program, the compiler will recognize NULL. A program that uses any of the memory allocation functions will include `<stdlib.h>`, of course, making NULL available.

In C, pointers test true or false in the same way as numbers. All non-null pointers test true; only null pointers are false. Thus, instead of writing

```
if (p == NULL) ...
```

we could write

```
if (!p) ...
```

and instead of writing

```
if (p != NULL) ...
```

we could write

```
if (p) ...
```

As a matter of style, I prefer the explicit comparison with NULL.

## 17.2 Dynamically Allocated Strings

Dynamic storage allocation is often useful for working with strings. Strings are stored in character arrays, and it can be hard to anticipate how long these arrays need to be. By allocating strings dynamically, we can postpone the decision until the program is running.

### Using `malloc` to Allocate Memory for a String

The `malloc` function has the following prototype:

```
void *malloc(size_t size);
```

size\_t type ▶ 7.6 `malloc` allocates a block of `size` bytes and returns a pointer to it. Note that `size` has type `size_t`, an unsigned integer type defined in the C library. Unless we're allocating a very large block of memory, we can just think of `size` as an ordinary integer.

Using `malloc` to allocate memory for a string is easy, because C guarantees that a `char` value requires exactly one byte of storage (`sizeof(char)` is 1, in other words). To allocate space for a string of `n` characters, we'd write

```
p = malloc(n + 1);
```

where `p` is a `char *` variable. (The argument is `n + 1` rather than `n` to allow room for the null character.) The generic pointer that `malloc` returns will be converted to `char *` when the assignment is performed; no cast is necessary. (In general, we can assign a `void *` value to a variable of any pointer type and vice versa.) Nevertheless, some programmers prefer to cast `malloc`'s return value:

```
p = (char *) malloc(n + 1);
```



When using `malloc` to allocate space for a string, don't forget to include room for the null character.

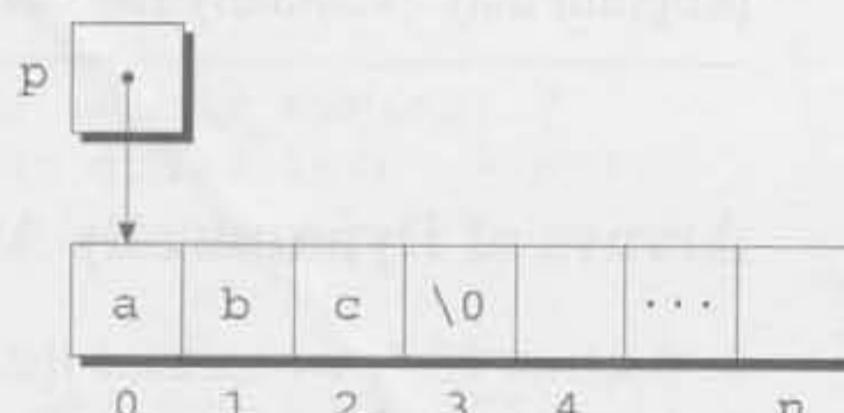
Memory allocated using `malloc` isn't cleared or initialized in any way, so `p` will point to an uninitialized array of `n + 1` characters:



Calling `strcpy` is one way to initialize this array:

```
strcpy(p, "abc");
```

The first four characters in the array will now be a, b, c, and \0:



## Using Dynamic Storage Allocation in String Functions

Dynamic storage allocation makes it possible to write functions that return a pointer to a “new” string—a string that didn’t exist before the function was called. Consider the problem of writing a function that concatenates two strings without changing either one. C’s standard library doesn’t include such a function (`strcat` isn’t quite what we want, since it modifies one of the strings passed to it), but we can easily write our own.

Our function will measure the lengths of the two strings to be concatenated, then call `malloc` to allocate just the right amount of space for the result. The function next copies the first string into the new space and then calls `strcat` to concatenate the second string.

```
char *concat(const char *s1, const char *s2)
{
 char *result;

 result = malloc(strlen(s1) + strlen(s2) + 1);
 if (result == NULL) {
 printf("Error: malloc failed in concat\n");
 exit(EXIT_FAILURE);
 }
 strcpy(result, s1);
 strcat(result, s2);
 return result;
}
```

If `malloc` returns a null pointer, `concat` prints an error message and terminates the program. That’s not always the right action to take; some programs need to recover from memory allocation failures and continue running.

Here’s how the `concat` function might be called:

```
p = concat("abc", "def");
```

After the call, `p` will point to the string "abcdef", which is stored in a dynamically allocated array. The array is seven characters long, including the null character at the end.



free function ▶ 17.4

Functions such as `concat` that dynamically allocate storage must be used with care. When the string that `concat` returns is no longer needed, we'll want to call the `free` function to release the space that the string occupies. If we don't, the program may eventually run out of memory.

## Arrays of Dynamically Allocated Strings

In Section 13.7, we tackled the problem of storing strings in an array. We found that storing strings as rows in a two-dimensional array of characters can waste space, so we tried setting up an array of pointers to string literals. The techniques of Section 13.7 work just as well if the elements of an array are pointers to dynamically allocated strings. To illustrate this point, let's rewrite the `remind.c` program of Section 13.5, which prints a one-month list of daily reminders.

### PROGRAM

## Printing a One-Month Reminder List (Revisited)

The original `remind.c` program stores the reminder strings in a two-dimensional array of characters, with each row of the array containing one string. After the program reads a day and its associated reminder, it searches the array to determine where the day belongs, using `strcmp` to do comparisons. It then uses `strcpy` to move all strings below that point down one position. Finally, the program copies the day into the array and calls `strcat` to append the reminder to the day.

In the new program (`remind2.c`), the array will be one-dimensional; its elements will be pointers to dynamically allocated strings. Switching to dynamically allocated strings in this program will have two primary advantages. First, we can use space more efficiently by allocating the exact number of characters needed to store a reminder, rather than storing the reminder in a fixed number of characters as the original program does. Second, we won't need to call `strcpy` to move existing reminder strings in order to make room for a new reminder. Instead, we'll merely move *pointers* to strings.

Here's the new program, with changes in **bold**. Switching from a two-dimensional array to an array of pointers turns out to be remarkably easy: we'll only need to change eight lines of the program.

```
remind2.c /* Prints a one-month reminder list (dynamic string version) */

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

#define MAX_REMIND 50 /* maximum number of reminders */
#define MSG_LEN 60 /* max length of reminder message */

int read_line(char str[], int n);
```

```
int main(void)
{
 char *reminders[MAX_REMIND];
 char day_str[3], msg_str[MSG_LEN+1];
 int day, i, j, num_remind = 0;

 for (;;) {
 if (num_remind == MAX_REMIND) {
 printf("-- No space left --\n");
 break;
 }

 printf("Enter day and reminder: ");
 scanf("%2d", &day);
 if (day == 0)
 break;
 sprintf(day_str, "%2d", day);
 read_line(msg_str, MSG_LEN);

 for (i = 0; i < num_remind; i++)
 if (strcmp(day_str, reminders[i]) < 0)
 break;
 for (j = num_remind; j > i; j--)
 reminders[j] = reminders[j-1];

 reminders[i] = malloc(2 + strlen(msg_str) + 1);
 if (reminders[i] == NULL) {
 printf("-- No space left --\n");
 break;
 }

 strcpy(reminders[i], day_str);
 strcat(reminders[i], msg_str);

 num_remind++;
 }

 printf("\nDay Reminder\n");
 for (i = 0; i < num_remind; i++)
 printf(" %s\n", reminders[i]);

 return 0;
}

int read_line(char str[], int n)
{
 int ch, i = 0;

 while ((ch = getchar()) != '\n')
 if (i < n)
 str[i++] = ch;
 str[i] = '\0';
 return i;
}
```

## 17.3 Dynamically Allocated Arrays

Dynamically allocated arrays have the same advantages as dynamically allocated strings (not surprisingly, since strings *are* arrays). When we’re writing a program, it’s often difficult to estimate the proper size for an array; it would be more convenient to wait until the program is run to decide how large the array should be. C solves this problem by allowing a program to allocate space for an array during execution, then access the array through a pointer to its first element. The close relationship between arrays and pointers, which we explored in Chapter 12, makes a dynamically allocated array just as easy to use as an ordinary array.

Although `malloc` can allocate space for an array, the `calloc` function is sometimes used instead, since it initializes the memory that it allocates. The `realloc` function allows us to make an array “grow” or “shrink” as needed.

### Using `malloc` to Allocate Storage for an Array

sizeof operator ▶ 7.6

We can use `malloc` to allocate space for an array in much the same way we used it to allocate space for a string. The primary difference is that the elements of an arbitrary array won’t necessarily be one byte long, as they are in a string. As a result, we’ll need to use the `sizeof` operator to calculate the amount of space required for each element.

Suppose we’re writing a program that needs an array of  $n$  integers, where  $n$  is to be computed during the execution of the program. We’ll first declare a pointer variable:

```
int *a;
```

Once the value of  $n$  is known, we’ll have the program call `malloc` to allocate space for the array:

```
a = malloc(n * sizeof(int));
```



Always use `sizeof` when calculating how much space is needed for an array. Failing to allocate enough memory can have severe consequences. Consider the following attempt to allocate space for an array of  $n$  integers:

```
a = malloc(n * 2);
```

If `int` values are larger than two bytes (as they are on most computers), `malloc` won’t allocate a large enough block of memory. When we later try to access elements of the array, the program may crash or behave erratically.

---

Once it points to a dynamically allocated block of memory, we can ignore the fact that `a` is a pointer and use it instead as an array name, thanks to the relation-

ship between arrays and pointers in C. For example, we could use the following loop to initialize the array that `a` points to:

```
for (i = 0; i < n; i++)
 a[i] = 0;
```

We also have the option of using pointer arithmetic instead of subscripting to access the elements of the array.

## The `calloc` Function

Although the `malloc` function can be used to allocate memory for an array, C provides an alternative—the `calloc` function—that's sometimes better. `calloc` has the following prototype in `<stdlib.h>`:

```
void *calloc(size_t nmemb, size_t size);
```

`calloc` allocates space for an array with `nmemb` elements, each of which is `size` bytes long; it returns a null pointer if the requested space isn't available. After allocating the memory, `calloc` initializes it by setting all bits to 0. For example, the following call of `calloc` allocates space for an array of `n` integers, which are all guaranteed to be zero initially:

```
a = calloc(n, sizeof(int));
```

Since `calloc` clears the memory that it allocates but `malloc` doesn't, we may occasionally want to use `calloc` to allocate space for an object other than an array. By calling `calloc` with 1 as its first argument, we can allocate space for a data item of any type:

```
struct point { int x, y; } *p;
p = calloc(1, sizeof(struct point));
```

After this statement has been executed, `p` will point to a structure whose `x` and `y` members have been set to zero.

## The `realloc` Function

Once we've allocated memory for an array, we may later find that it's too large or too small. The `realloc` function can resize the array to better suit our needs. The following prototype for `realloc` appears in `<stdlib.h>`:

```
void *realloc(void *ptr, size_t size);
```

When `realloc` is called, `ptr` must point to a memory block obtained by a previous call of `malloc`, `calloc`, or `realloc`. The `size` parameter represents the new size of the block, which may be larger or smaller than the original size. Although `realloc` doesn't require that `ptr` point to memory that's being used as an array, in practice it usually does.



Be sure that a pointer passed to `realloc` came from a previous call of `malloc`, `calloc`, or `realloc`. If it didn't, calling `realloc` causes undefined behavior.

The C standard spells out a number of rules concerning the behavior of `realloc`:

- When it expands a memory block, `realloc` doesn't initialize the bytes that are added to the block.
- If `realloc` can't enlarge the memory block as requested, it returns a null pointer; the data in the old memory block is unchanged.
- If `realloc` is called with a null pointer as its first argument, it behaves like `malloc`.
- If `realloc` is called with 0 as its second argument, it frees the memory block.

The C standard stops short of specifying exactly how `realloc` works. Still, we expect it to be reasonably efficient. When asked to reduce the size of a memory block, `realloc` should shrink the block "in place," without moving the data stored in the block. By the same token, `realloc` should always attempt to expand a memory block without moving it. If it's unable to enlarge the block (because the bytes following the block are already in use for some other purpose), `realloc` will allocate a new block elsewhere, then copy the contents of the old block into the new one.



Once `realloc` has returned, be sure to update all pointers to the memory block, since it's possible that `realloc` has moved the block elsewhere.

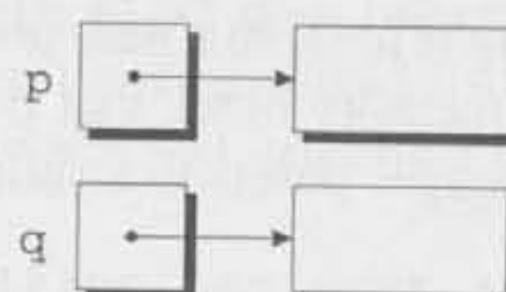
## 17.4 Deallocating Storage

`malloc` and the other memory allocation functions obtain memory blocks from a storage pool known as the *heap*. Calling these functions too often—or asking them for large blocks of memory—can exhaust the heap, causing the functions to return a null pointer.

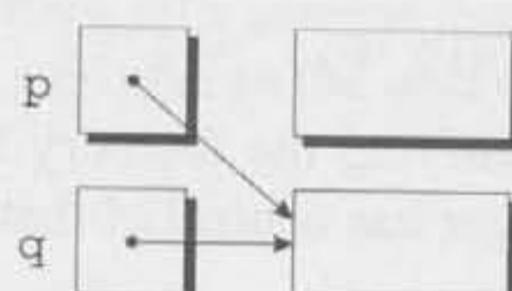
To make matters worse, a program may allocate blocks of memory and then lose track of them, thereby wasting space. Consider the following example:

```
p = malloc(...);
q = malloc(...);
p = q;
```

After the first two statements have been executed, p points to one memory block, while q points to another:



After q is assigned to p, both variables now point to the second memory block:



There are no pointers to the first block (shaded), so we'll never be able to use it again.

A block of memory that's no longer accessible to a program is said to be *garbage*. A program that leaves garbage behind has a *memory leak*. Some languages provide a *garbage collector* that automatically locates and recycles garbage, but C doesn't. Instead, each C program is responsible for recycling its own garbage by calling the `free` function to release unneeded memory.

## The `free` Function

The `free` function has the following prototype in `<stdlib.h>`:

```
void free(void *ptr);
```

Using `free` is easy; we simply pass it a pointer to a memory block that we no longer need:

```
p = malloc(...);
q = malloc(...);
free(p);
p = q;
```

Calling `free` releases the block of memory that p points to. This block is now available for reuse in subsequent calls of `malloc` or other memory allocation functions.



The argument to `free` must be a pointer that was previously returned by a memory allocation function. (The argument may also be a null pointer, in which case the call of `free` has no effect.) Passing `free` a pointer to any other object (such as a variable or array element) causes undefined behavior.

## The “Dangling Pointer” Problem

Although the `free` function allows us to reclaim memory that’s no longer needed, using it leads to a new problem: *dangling pointers*. The call `free(p)` deallocates the memory block that `p` points to, but doesn’t change `p` itself. If we forget that `p` no longer points to a valid memory block, chaos may ensue:

```
char *p = malloc(4);
...
free(p);
...
strcpy(p, "abc"); /*** WRONG ***/
```

Modifying the memory that `p` points to is a serious error, since our program no longer has control of that memory.



Attempting to access or modify a deallocated memory block causes undefined behavior. Trying to modify a deallocated memory block is likely to have disastrous consequences that may include a program crash.

Dangling pointers can be hard to spot, since several pointers may point to the same block of memory. When the block is freed, all the pointers are left dangling.

## 17.5 Linked Lists

Dynamic storage allocation is especially useful for building lists, trees, graphs, and other linked data structures. We’ll look at linked lists in this section; a discussion of other linked data structures is beyond the scope of this book. For more information, consult a book such as Robert Sedgewick’s *Algorithms in C, Parts 1–4: Fundamentals, Data Structures, Sorting, Searching*, Third Edition (Reading, Mass.: Addison-Wesley, 1998).

A *linked list* consists of a chain of structures (called *nodes*), with each node containing a pointer to the next node in the chain:



The last node in the list contains a null pointer, shown here as a diagonal line.

In previous chapters, we’ve used an array whenever we’ve needed to store a collection of data items; linked lists give us an alternative. A linked list is more flexible than an array: we can easily insert and delete nodes in a linked list, allowing the list to grow and shrink as needed. On the other hand, we lose the “random access” capability of an array. Any element of an array can be accessed in the same

amount of time; accessing a node in a linked list is fast if the node is close to the beginning of the list, slow if it's near the end.

This section describes how to set up a linked list in C. It also shows how to perform several common operations on linked lists: inserting a node at the beginning of a list, searching for a node, and deleting a node.

## Declaring a Node Type

To set up a linked list, the first thing we'll need is a structure that represents a single node in the list. For simplicity, let's assume that a node contains nothing but an integer (the node's data) plus a pointer to the next node in the list. Here's what our node structure will look like:

```
struct node {
 int value; /* data stored in the node */
 struct node *next; /* pointer to the next node */
};
```

Notice that the `next` member has type `struct node *`, which means that it can store a pointer to a node structure. There's nothing special about the name `node`, by the way; it's just an ordinary structure tag.

One aspect of the `node` structure deserves special mention. As Section 16.2 explained, we normally have the option of using either a tag or a `typedef` name to define a name for a particular kind of structure. However, when a structure has a member that points to the same kind of structure, as `node` does, we're required to use a structure tag. Without the `node` tag, we'd have no way to declare the type of `next`.

Now that we have the `node` structure declared, we'll need a way to keep track of where the list begins. In other words, we'll need a variable that always points to the first node in the list. Let's name the variable `first`:

```
struct node *first = NULL;
```

Setting `first` to `NULL` indicates that the list is initially empty.

## Creating a Node

As we construct a linked list, we'll want to create nodes one by one, adding each to the list. Creating a node requires three steps:

1. Allocate memory for the node.
2. Store data in the node.
3. Insert the node into the list.

We'll concentrate on the first two steps for now.

When we create a node, we'll need a variable that can point to the node temporarily, until it's been inserted into the list. Let's call this variable `new_node`:

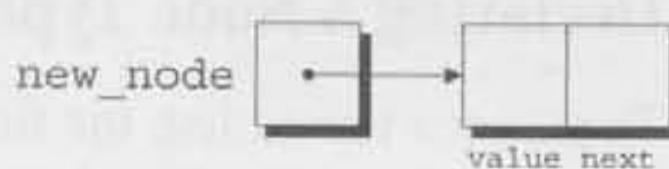
```
struct node *new_node;
```

**Q&A**

We'll use `malloc` to allocate memory for the new node, saving the return value in `new_node`:

```
new_node = malloc(sizeof(struct node));
```

`new_node` now points to a block of memory just large enough to hold a node structure:



Be careful to give `sizeof` the name of the *type* to be allocated, not the name of a *pointer* to that type:

```
new_node = malloc(sizeof(new_node)); /*** WRONG ***/
```

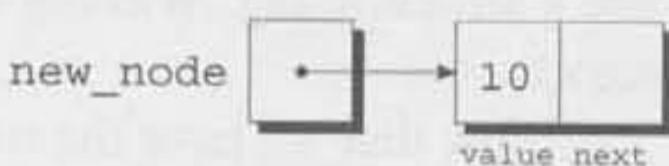
### Q&A

The program will still compile, but `malloc` will allocate only enough memory for a *pointer* to a node structure. The likely result is a crash later, when the program attempts to store data in the node that `new_node` is presumably pointing to.

Next, we'll store data in the `value` member of the new node:

```
(*new_node).value = 10;
```

Here's how the picture will look after this assignment:



To access the `value` member of the node, we've applied the indirection operator `*` (to reference the structure to which `new_node` points), then the selection operator `.` (to select a member of the structure). The parentheses around `*new_node` are mandatory because the `.` operator would otherwise take precedence over the `*` operator.

table of operators ➤ Appendix A

## The `->` Operator

Before we go on to the next step, inserting a new node into a list, let's take a moment to discuss a useful shortcut. Accessing a member of a structure using a pointer is so common that C provides a special operator just for this purpose. This operator, known as **right arrow selection**, is a minus sign followed by `>`. Using the `->` operator, we can write

```
new_node->value = 10;
```

instead of

```
(*new_node).value = 10;
```

The `->` operator is a combination of the `*` and `.` operators; it performs indirection on `new_node` to locate the structure that it points to, then selects the `value` member of the structure.

values ► 4.2

The `->` operator produces an lvalue, so we can use it wherever an ordinary variable would be allowed. We've just seen an example in which `new_node->value` appears on the left side of an assignment. It could just as easily appear in a call of `scanf`:

```
scanf("%d", &new_node->value);
```

Notice that the `&` operator is still required, even though `new_node` is a pointer. Without the `&`, we'd be passing `scanf` the *value* of `new_node->value`, which has type `int`.

## Inserting a Node at the Beginning of a Linked List

One of the advantages of a linked list is that nodes can be added at any point in the list: at the beginning, at the end, or anywhere in the middle. The beginning of a list is the easiest place to insert a node, however, so let's focus on that case.

If `new_node` is pointing to the node to be inserted, and `first` is pointing to the first node in the linked list, then we'll need two statements to insert the node into the list. First, we'll modify the new node's `next` member to point to the node that was previously at the beginning of the list:

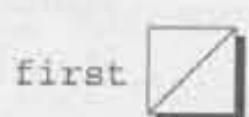
```
new_node->next = first;
```

Second, we'll make `first` point to the new node:

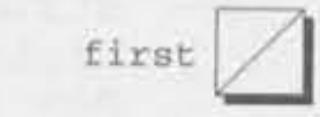
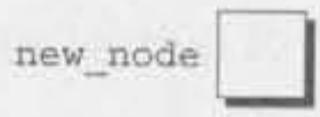
```
first = new_node;
```

Will these statements work if the list is empty when we insert a node? Yes, fortunately. To make sure this is true, let's trace the process of inserting two nodes into an empty list. We'll insert a node containing the number 10 first, followed by a node containing 20. In the figures that follow, null pointers are shown as diagonal lines.

```
first = NULL;
```



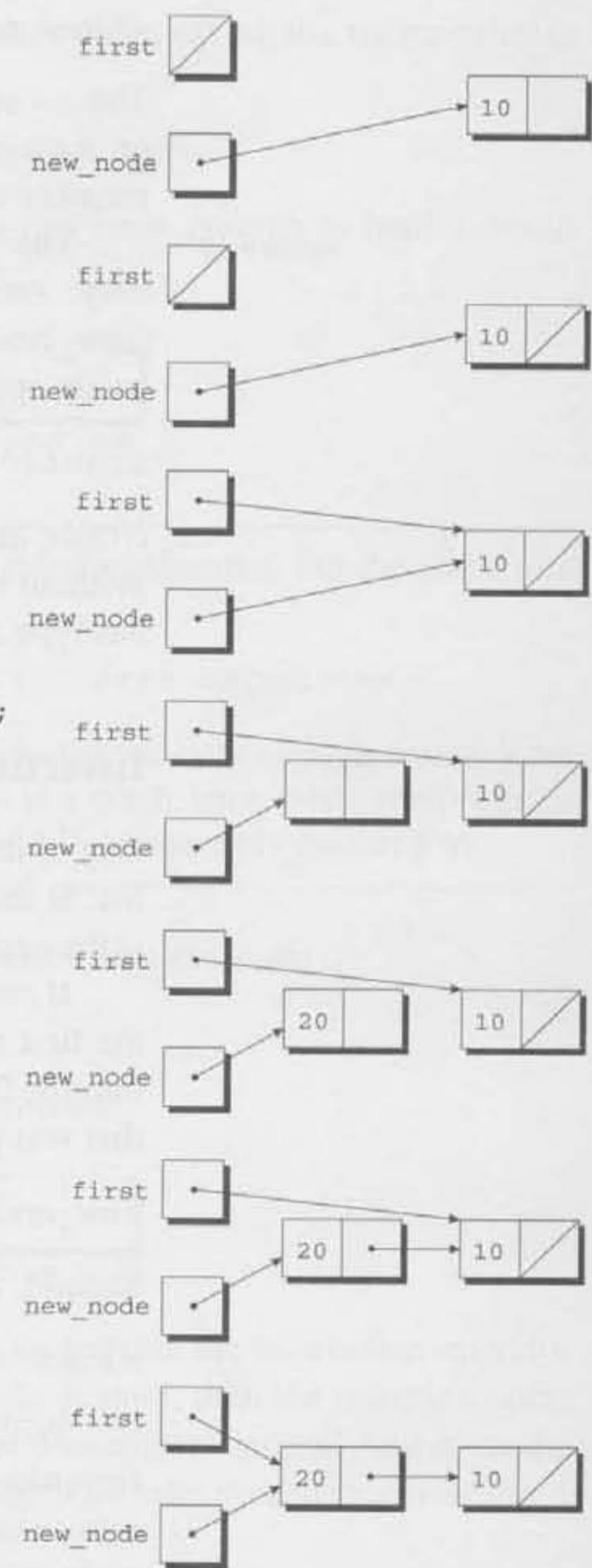
```
new_node = malloc(sizeof(struct node));
```



```

new_node->value = 10;
new_node->next = first;
first = new_node;
new_node = malloc(sizeof(struct node));

```



Inserting a node into a linked list is such a common operation that we'll probably want to write a function for that purpose. Let's name the function `add_to_list`. It will have two parameters: `list` (a pointer to the first node in the old list) and `n` (the integer to be stored in the new node).

```

struct node *add_to_list(struct node *list, int n)
{
 struct node *new_node;

 new_node = malloc(sizeof(struct node));
 if (new_node == NULL) {
 printf("Error: malloc failed in add_to_list\n");
 exit(EXIT_FAILURE);
 }
}

```

```

 new_node->value = n;
 new_node->next = list;
 return new_node;
}

```

Note that `add_to_list` doesn't modify the `list` pointer. Instead, it returns a pointer to the newly created node (now at the beginning of the list). When we call `add_to_list`, we'll need to store its return value into `first`:

```

first = add_to_list(first, 10);
first = add_to_list(first, 20);

```

These statements add nodes containing 10 and 20 to the list pointed to by `first`. Getting `add_to_list` to update `first` directly, rather than return a new value for `first`, turns out to be tricky. We'll return to this issue in Section 17.6.

The following function uses `add_to_list` to create a linked list containing numbers entered by the user:

```

struct node *read_numbers(void)
{
 struct node *first = NULL;
 int n;

 printf("Enter a series of integers (0 to terminate): ");
 for (;;) {
 scanf("%d", &n);
 if (n == 0)
 return first;
 first = add_to_list(first, n);
 }
}

```

The numbers will be in reverse order within the list, since `first` always points to the node containing the last number entered.

## Searching a Linked List

Once we've created a linked list, we may need to search it for a particular piece of data. Although a `while` loop can be used to search a list, the `for` statement is often superior. We're accustomed to using the `for` statement when writing loops that involve counting, but its flexibility makes the `for` statement suitable for other tasks as well, including operations on linked lists. Here's the customary way to visit the nodes in a linked list, using a pointer variable `p` to keep track of the "current" node:

**idiom** `for (p = first; p != NULL; p = p->next)`

...

The assignment

```
p = p->next
```

advances the `p` pointer from one node to the next. An assignment of this form is invariably used in C when writing a loop that traverses a linked list.

Let's write a function named `search_list` that searches a list (pointed to by the parameter `list`) for an integer `n`. If it finds `n`, `search_list` will return a pointer to the node containing `n`; otherwise, it will return a null pointer. Our first version of `search_list` relies on the "list-traversal" idiom:

```
struct node *search_list(struct node *list, int n)
{
 struct node *p;

 for (p = list; p != NULL; p = p->next)
 if (p->value == n)
 return p;
 return NULL;
}
```

Of course, there are many other ways to write `search_list`. One alternative would be to eliminate the `p` variable, instead using `list` itself to keep track of the current node:

```
struct node *search_list(struct node *list, int n)
{
 for (; list != NULL; list = list->next)
 if (list->value == n)
 return list;
 return NULL;
}
```

Since `list` is a copy of the original `list` pointer, there's no harm in changing it within the function.

Another alternative is to combine the `list->value == n` test with the `list != NULL` test:

```
struct node *search_list(struct node *list, int n)
{
 for (; list != NULL && list->value != n; list = list->next)
 ;
 return list;
}
```

Since `list` is `NULL` if we reach the end of the list, returning `list` is correct even if we don't find `n`. This version of `search_list` might be a bit clearer if we used a `while` statement:

```
struct node *search_list(struct node *list, int n)
{
 while (list != NULL && list->value != n)
 list = list->next;
 return list;
}
```

## Deleting a Node from a Linked List

A big advantage of storing data in a linked list is that we can easily delete nodes that we no longer need. Deleting a node, like creating a node, involves three steps:

1. Locate the node to be deleted.
2. Alter the previous node so that it “bypasses” the deleted node.
3. Call `free` to reclaim the space occupied by the deleted node.

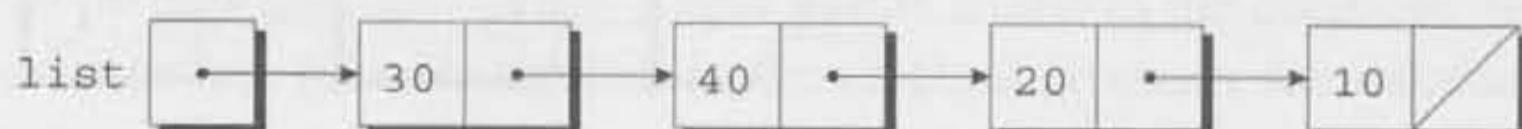
Step 1 is harder than it looks. If we search the list in the obvious way, we'll end up with a pointer to the node to be deleted. Unfortunately, we won't be able to perform step 2, which requires changing the *previous* node.

There are various solutions to this problem. We'll use the “trailing pointer” technique: as we search the list in step 1, we'll keep a pointer to the previous node (`prev`) as well as a pointer to the current node (`cur`). If `list` points to the list to be searched and `n` is the integer to be deleted, the following loop implements step 1:

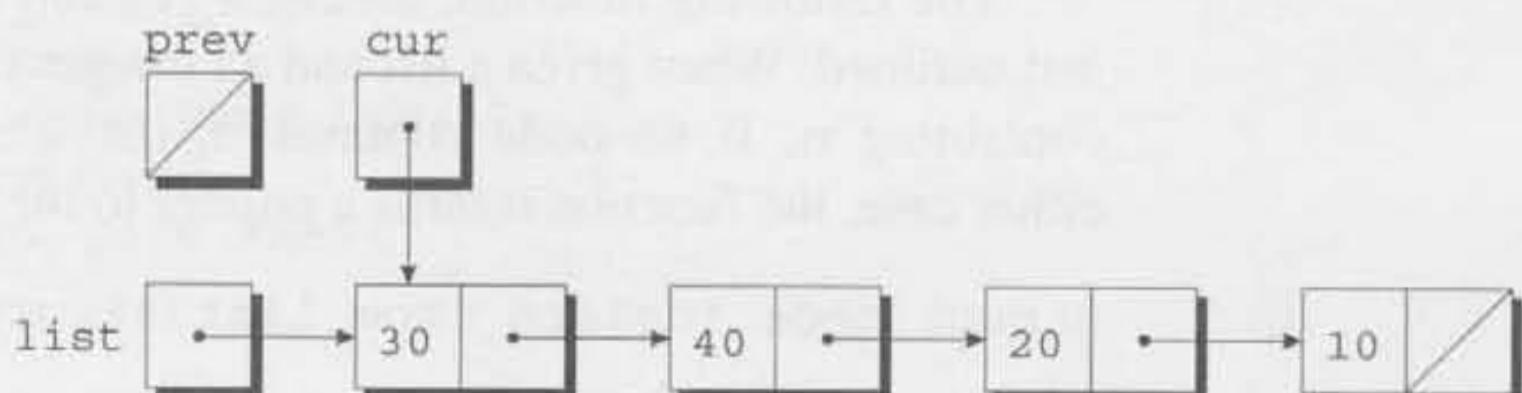
```
for (cur = list, prev = NULL;
 cur != NULL && cur->value != n;
 prev = cur, cur = cur->next)
;
```

Here we see the power of C's `for` statement. This rather exotic example, with its empty body and liberal use of the comma operator, performs all the actions needed to search for `n`. When the loop terminates, `cur` points to the node to be deleted, while `prev` points to the previous node (if there is one).

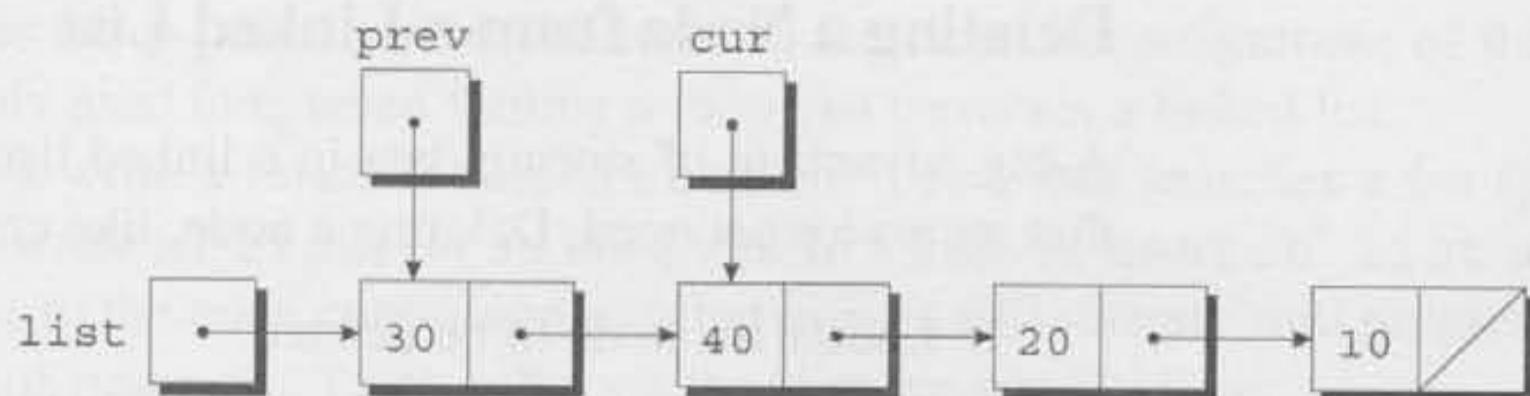
To see how this loop works, let's assume that `list` points to a list containing 30, 40, 20, and 10, in that order:



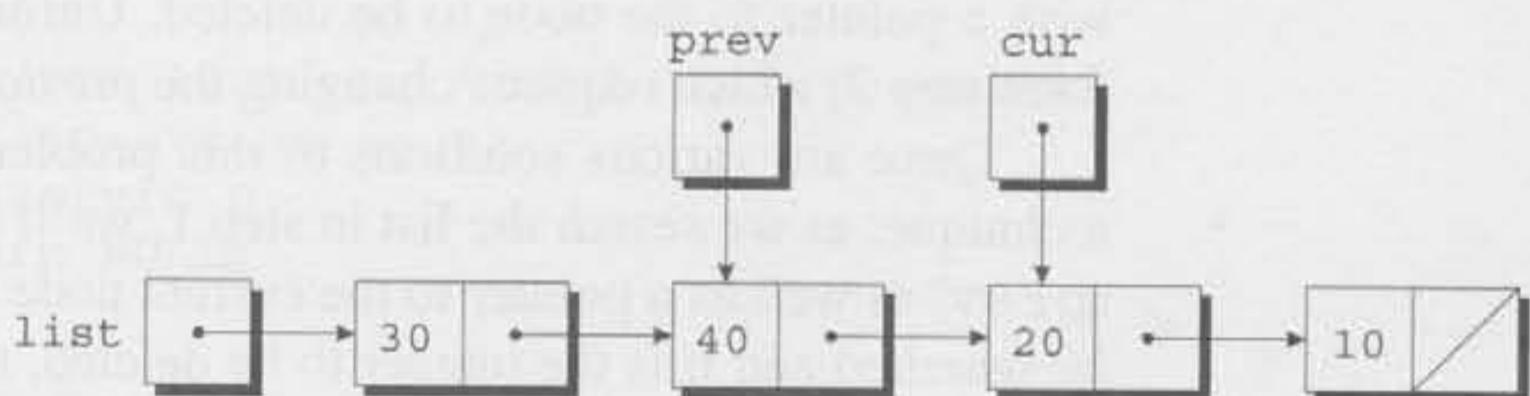
Let's say that `n` is 20, so our goal is to delete the third node in the list. After `cur = list, prev = NULL` has been executed, `cur` points to the first node in the list:



The test `cur != NULL && cur->value != n` is true, since `cur` is pointing to a node and the node doesn't contain 20. After `prev = cur, cur = cur->next` has been executed, we begin to see how the `prev` pointer will trail behind `cur`:



Again, the test `cur != NULL && cur->value != n` is true, so `prev = cur`, `cur = cur->next` is executed once more:

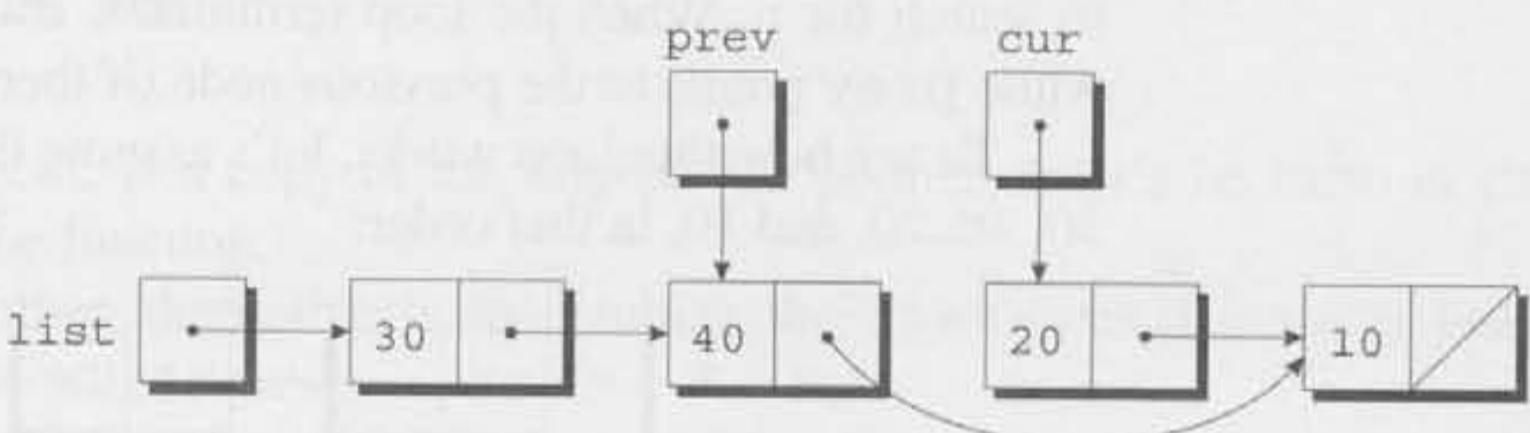


Since `cur` now points to the node containing 20, the condition `cur->value != n` is false and the loop terminates.

Next, we'll perform the bypass required by step 2. The statement

```
prev->next = cur->next;
```

makes the pointer in the previous node point to the node *after* the current node:



We're now ready for step 3, releasing the memory occupied by the current node:

```
free(cur);
```

The following function, `delete_from_list`, uses the strategy that we've just outlined. When given a list and an integer `n`, the function deletes the first node containing `n`. If no node contains `n`, `delete_from_list` does nothing. In either case, the function returns a pointer to the list.

```
struct node *delete_from_list(struct node *list, int n)
{
 struct node *cur, *prev;

 for (cur = list, prev = NULL;
 cur != NULL && cur->value != n;
 prev = cur, cur = cur->next)
 ;
```

```

 if (cur == NULL)
 return list; /* n was not found */
 if (prev == NULL)
 list = list->next; /* n is in the first node */
 else
 prev->next = cur->next; /* n is in some other node */
 free(cur);
 return list;
}

```

Deleting the first node in the list is a special case. The `prev == NULL` test checks for this case, which requires a different bypass step.

## Ordered Lists

When the nodes of a list are kept in order—sorted by the data stored inside the nodes—we say that the list is *ordered*. Inserting a node into an ordered list is more difficult (the node won’t always be put at the beginning of the list), but searching is faster (we can stop looking after reaching the point at which the desired node would have been located). The following program illustrates both the increased difficulty of inserting a node and the faster search.

### PROGRAM Maintaining a Parts Database (Revisited)

Let’s redo the parts database program of Section 16.3, this time storing the database in a linked list. Using a linked list instead of an array has two major advantages: (1) We don’t need to put a preset limit on the size of the database; it can grow until there’s no more memory to store parts. (2) We can easily keep the database sorted by part number—when a new part is added to the database, we simply insert it in its proper place in the list. In the original program, the database wasn’t sorted.

In the new program, the `part` structure will contain an additional member (a pointer to the next node in the linked list), and the variable `inventory` will be a pointer to the first node in the list:

```

struct part {
 int number;
 char name[NAME_LEN+1];
 int on_hand;
 struct part *next;
};

struct part *inventory = NULL; /* points to first part */

```

Most of the functions in the new program will closely resemble their counterparts in the original program. The `find_part` and `insert` functions will be more complex, however, since we’ll keep the nodes in the `inventory` list sorted by part number.

In the original program, `find_part` returns an index into the `inventory` array. In the new program, `find_part` will return a pointer to the node that contains the desired part number. If it doesn't find the part number, `find_part` will return a null pointer. Since the `inventory` list is sorted by part number, the new version of `find_part` can save time by stopping its search when it finds a node containing a part number that's greater than or equal to the desired part number. `find_part`'s search loop will have the form

```
for (p = inventory;
 p != NULL && number > p->number;
 p = p->next)
;
```

The loop will terminate when `p` becomes `NULL` (indicating that the part number wasn't found) or when `number > p->number` is false (indicating that the part number we're looking for is less than or equal to a number already stored in a node). In the latter case, we still don't know whether or not the desired number is actually in the list, so we'll need another test:

```
if (p != NULL && number == p->number)
 return p;
```

The original version of `insert` stores a new part in the next available array element. The new version must determine where the new part belongs in the list and insert it there. We'll also have `insert` check whether the part number is already present in the list. `insert` can accomplish both tasks by using a loop similar to the one in `find_part`:

```
for (cur = inventory, prev = NULL;
 cur != NULL && new_node->number > cur->number;
 prev = cur, cur = cur->next)
;
```

This loop relies on two pointers: `cur`, which points to the current node, and `prev`, which points to the previous node. Once the loop terminates, `insert` will check whether `cur` isn't `NULL` and `new_node->number` equals `cur->number`; if so, the part number is already in the list. Otherwise `insert` will insert a new node between the nodes pointed to by `prev` and `cur`, using a strategy similar to the one we employed for deleting a node. (This strategy works even if the new part number is larger than any in the list; in that case, `cur` will be `NULL` but `prev` will point to the last node in the list.)

Here's the new program. Like the original program, this version requires the `read_line` function described in Section 16.3; I assume that `readline.h` contains a prototype for this function.

```
inventory2.c /* Maintains a parts database (linked list version) */

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

```

#define NAME_LEN 25

struct part {
 int number;
 char name[NAME_LEN+1];
 int on_hand;
 struct part *next;
};

struct part *inventory = NULL; /* points to first part */

struct part *find_part(int number);
void insert(void);
void search(void);
void update(void);
void print(void);

/******************
 * main: Prompts the user to enter an operation code,
 * then calls a function to perform the requested
 * action. Repeats until the user enters the
 * command 'q'. Prints an error message if the user
 * enters an illegal code.
*****************/
int main(void)
{
 char code;

 for (;;) {
 printf("Enter operation code: ");
 scanf(" %c", &code);
 while (getchar() != '\n') /* skips to end of line */
 ;
 switch (code) {
 case 'i': insert();
 break;
 case 's': search();
 break;
 case 'u': update();
 break;
 case 'p': print();
 break;
 case 'q': return 0;
 default: printf("Illegal code\n");
 }
 printf("\n");
 }
}

/******************
 * find_part: Looks up a part number in the inventory
 * list. Returns a pointer to the node
 * containing the part number; if the part
 * number is not found, returns NULL.
*****************/

```

```
struct part *find_part(int number)
{
 struct part *p;

 for (p = inventory;
 p != NULL && number > p->number;
 p = p->next)
 ;
 if (p != NULL && number == p->number)
 return p;
 return NULL;
}

/*****************
 * insert: Prompts the user for information about a new *
 * part and then inserts the part into the *
 * inventory list; the list remains sorted by *
 * part number. Prints an error message and *
 * returns prematurely if the part already exists *
 * or space could not be allocated for the part. *
 *****************/
void insert(void)
{
 struct part *cur, *prev, *new_node;

 new_node = malloc(sizeof(struct part));
 if (new_node == NULL) {
 printf("Database is full; can't add more parts.\n");
 return;
 }

 printf("Enter part number: ");
 scanf("%d", &new_node->number);

 for (cur = inventory, prev = NULL;
 cur != NULL && new_node->number > cur->number;
 prev = cur, cur = cur->next)
 ;
 if (cur != NULL && new_node->number == cur->number) {
 printf("Part already exists.\n");
 free(new_node);
 return;
 }

 printf("Enter part name: ");
 read_line(new_node->name, NAME_LEN);
 printf("Enter quantity on hand: ");
 scanf("%d", &new_node->on_hand);

 new_node->next = cur;
 if (prev == NULL)
 inventory = new_node;
 else
 prev->next = new_node;
}
```

```

 * search: Prompts the user to enter a part number, then *
 * looks up the part in the database. If the part *
 * exists, prints the name and quantity on hand; *
 * if not, prints an error message. *

void search(void)
{
 int number;
 struct part *p;

 printf("Enter part number: ");
 scanf("%d", &number);
 p = find_part(number);
 if (p != NULL) {
 printf("Part name: %s\n", p->name);
 printf("Quantity on hand: %d\n", p->on_hand);
 } else
 printf("Part not found.\n");
}

 * update: Prompts the user to enter a part number. *
 * Prints an error message if the part doesn't *
 * exist; otherwise, prompts the user to enter *
 * change in quantity on hand and updates the *
 * database. *

void update(void)
{
 int number, change;
 struct part *p;

 printf("Enter part number: ");
 scanf("%d", &number);
 p = find_part(number);
 if (p != NULL) {
 printf("Enter change in quantity on hand: ");
 scanf("%d", &change);
 p->on_hand += change;
 } else
 printf("Part not found.\n");
}

 * print: Prints a listing of all parts in the database, *
 * showing the part number, part name, and *
 * quantity on hand. Part numbers will appear in *
 * ascending order. *

void print(void)
{
 struct part *p;
```

```

 printf("Part Number Part Name
 "Quantity on Hand\n");
 for (p = inventory; p != NULL; p = p->next)
 printf("%7d %-25s%11d\n", p->number, p->name,
 p->on_hand);
}

```

Notice the use of `free` in the `insert` function. `insert` allocates memory for a part before checking to see if the part already exists. If it does, `insert` releases the space to avoid a memory leak.

---

## 17.6 Pointers to Pointers

In Section 13.7, we came across the notion of a *pointer to a pointer*. In that section, we used an array whose elements were of type `char *`; a pointer to one of the array elements itself had type `char **`. The concept of “pointers to pointers” also pops up frequently in the context of linked data structures. In particular, when an argument to a function is a pointer variable, we’ll sometimes want the function to be able to modify the variable by making it point somewhere else. Doing so requires the use of a pointer to a pointer.

Consider the `add_to_list` function of Section 17.5, which inserts a node at the beginning of a linked list. When we call `add_to_list`, we pass it a pointer to the first node in the original list; it then returns a pointer to the first node in the updated list:

```

struct node *add_to_list(struct node *list, int n)
{
 struct node *new_node;

 new_node = malloc(sizeof(struct node));
 if (new_node == NULL) {
 printf("Error: malloc failed in add_to_list\n");
 exit(EXIT_FAILURE);
 }
 new_node->value = n;
 new_node->next = list;
 return new_node;
}

```

Suppose that we modify the function so that it assigns `new_node` to `list` instead of returning `new_node`. In other words, let’s remove the `return` statement from `add_to_list` and replace it by

```
list = new_node;
```

Unfortunately, this idea doesn’t work. Suppose that we call `add_to_list` in the following way:

```
add_to_list(first, 10);
```

At the point of the call, `first` is copied into `list`. (Pointers, like all arguments, are passed by value.) The last line in the function changes the value of `list`, making it point to the new node. This assignment doesn't affect `first`, however.

Getting `add_to_list` to modify `first` is possible, but it requires passing `add_to_list` a *pointer* to `first`. Here's the correct version of the function:

```
void add_to_list(struct node **list, int n)
{
 struct node *new_node;

 new_node = malloc(sizeof(struct node));
 if (new_node == NULL) {
 printf("Error: malloc failed in add_to_list\n");
 exit(EXIT_FAILURE);
 }
 new_node->value = n;
 new_node->next = *list;
 *list = new_node;
}
```

When we call the new version of `add_to_list`, the first argument will be the address of `first`:

```
add_to_list(&first, 10);
```

Since `list` is assigned the address of `first`, we can use `*list` as an alias for `first`. In particular, assigning `new_node` to `*list` will modify `first`.

## 17.7 Pointers to Functions

We've seen that pointers may point to various kinds of data, including variables, array elements, and dynamically allocated blocks of memory. But C doesn't require that pointers point only to *data*; it's also possible to have pointers to *functions*. Pointers to functions aren't as odd as you might think. After all, functions occupy memory locations, so every function has an address, just as each variable has an address.

### Function Pointers as Arguments

We can use function pointers in much the same way we use pointers to data. In particular, passing a function pointer as an argument is fairly common in C. Suppose that we're writing a function named `integrate` that integrates a mathematical function `f` between points `a` and `b`. We'd like to make `integrate` as general as possible by passing it `f` as an argument. To achieve this effect in C, we'll declare `f` to be a pointer to a function. Assuming that we want to integrate functions that have

a double parameter and return a double result, the prototype for `integrate` will look like this:

```
double integrate(double (*f)(double), double a, double b);
```

The parentheses around `*f` indicate that `f` is a pointer to a function, not a function that returns a pointer. It's also legal to declare `f` as though it were a function:

```
double integrate(double f(double), double a, double b);
```

From the compiler's standpoint, this prototype is identical to the previous one.

sin function ► 23.3 When we call `integrate`, we'll supply a function name as the first argument. For example, the following call will integrate the `sin` (sine) function from 0 to  $\pi/2$ :

```
result = integrate(sin, 0.0, PI / 2);
```

Notice that there are no parentheses after `sin`. When a function name isn't followed by parentheses, the C compiler produces a pointer to the function instead of generating code for a function call. In our example, we're not calling `sin`; instead, we're passing `integrate` a pointer to `sin`. If this seems confusing, think of how C handles arrays. If `a` is the name of an array, then `a[i]` represents one element of the array, while `a` by itself serves as a pointer to the array. In a similar way, if `f` is a function, C treats `f(x)` as a *call* of the function but `f` by itself as a *pointer* to the function.

Within the body of `integrate`, we can call the function that `f` points to:

```
y = (*f)(x);
```

`*f` represents the function that `f` points to; `x` is the argument to the call. Thus, during the execution of `integrate(sin, 0.0, PI / 2)`, each call of `*f` is actually a call of `sin`. As an alternative to `(*f)(x)`, C allows us to write `f(x)` to call the function that `f` points to. Although `f(x)` looks more natural, I'll stick with `(*f)(x)` as a reminder that `f` is a pointer to a function, not a function name.

## The `qsort` Function

Although it might seem that pointers to functions aren't relevant to the average programmer, that couldn't be further from the truth. In fact, some of the most useful functions in the C library require a function pointer as an argument. One of these is `qsort`, which belongs to the `<stdlib.h>` header. `qsort` is a general-purpose sorting function that's capable of sorting any array, based on any criteria that we choose.

Since the elements of the array that it sorts may be of any type—even a structure or union type—`qsort` must be told how to determine which of two array elements is “smaller.” We'll provide this information to `qsort` by writing a **comparison function**. When given two pointers `p` and `q` to array elements, the comparison function must return an integer that is *negative* if `*p` is “less than” `*q`,

### Q&A

*zero* if  $*p$  is “equal to”  $*q$ , and *positive* if  $*p$  is “greater than”  $*q$ . The terms “less than,” “equal to,” and “greater than” are in quotes because it’s our responsibility to determine how  $*p$  and  $*q$  are compared.

`qsort` has the following prototype:

```
void qsort(void *base, size_t nmemb, size_t size,
 int (*compar)(const void *, const void *));
```

`base` must point to the first element in the array. (If only a portion of the array is to be sorted, we’ll make `base` point to the first element in this portion.) In the simplest case, `base` is just the name of the array. `nmemb` is the number of elements to be sorted (not necessarily the number of elements in the array). `size` is the size of each array element, measured in bytes. `compar` is a pointer to the comparison function. When `qsort` is called, it sorts the array into ascending order, calling the comparison function whenever it needs to compare array elements.

To sort the `inventory` array of Section 16.3, we’d use the following call of `qsort`:

```
qsort(inventory, num_parts, sizeof(struct part), compare_parts);
```

Notice that the second argument is `num_parts`, not `MAX_PARTS`; we don’t want to sort the entire `inventory` array, just the portion in which parts are currently stored. The last argument, `compare_parts`, is a function that compares two `part` structures.

Writing the `compare_parts` function isn’t as easy as you might expect. `qsort` requires that its parameters have type `void *`, but we can’t access the members of a `part` structure through a `void *` pointer; we need a pointer of type `struct part *` instead. To solve the problem, we’ll have `compare_parts` assign its parameters, `p` and `q`, to variables of type `struct part *`, thereby converting them to the desired type. `compare_parts` can now use these variables to access the members of the structures that `p` and `q` point to. Assuming that we want to sort the `inventory` array into ascending order by part number, here’s how the `compare_parts` function might look:

```
int compare_parts(const void *p, const void *q)
{
 const struct part *p1 = p;
 const struct part *q1 = q;

 if (p1->number < q1->number)
 return -1;
 else if (p1->number == q1->number)
 return 0;
 else
 return 1;
}
```

The declarations of `p1` and `q1` include the word `const` to avoid getting a warning from the compiler. Since `p` and `q` are `const` pointers (indicating that the objects

## Q&A

to which they point should not be modified), they should be assigned only to pointer variables that are also declared to be `const`.

Although this version of `compare_parts` works, most C programmers would write the function more concisely. First, notice that we can replace `p1` and `q1` by cast expressions:

```
int compare_parts(const void *p, const void *q)
{
 if (((struct part *) p)->number <
 ((struct part *) q)->number)
 return -1;
 else if (((struct part *) p)->number ==
 ((struct part *) q)->number))
 return 0;
 else
 return 1;
}
```

The parentheses around `((struct part *) p)` are necessary; without them, the compiler would try to cast `p->number` to type `struct part *`.

We can make `compare_parts` even shorter by removing the `if` statements:

```
int compare_parts(const void *p, const void *q)
{
 return ((struct part *) p)->number -
 ((struct part *) q)->number;
}
```

Subtracting `q`'s part number from `p`'s part number produces a negative result if `p` has a smaller part number, zero if the part numbers are equal, and a positive result if `p` has a larger part number. (Note that subtracting two integers is potentially risky because of the danger of overflow. I'm assuming that part numbers are positive integers, so that shouldn't happen here.)

To sort the `inventory` array by part name instead of part number, we'd use the following version of `compare_parts`:

```
int compare_parts(const void *p, const void *q)
{
 return strcmp(((struct part *) p)->name,
 ((struct part *) q)->name);
}
```

All `compare_parts` has to do is call `strcmp`, which conveniently returns a negative, zero, or positive result.

## Other Uses of Function Pointers

Although I've emphasized the usefulness of function pointers as arguments to other functions, that's not all they're good for. C treats pointers to functions just like pointers to data; we can store function pointers in variables or use them as ele-

ments of an array or as members of a structure or union. We can even write functions that return function pointers.

Here's an example of a variable that can store a pointer to a function:

```
void (*pf)(int);
```

`pf` can point to any function with an `int` parameter and a return type of `void`. If `f` is such a function, we can make `pf` point to `f` in the following way:

```
pf = f;
```

Notice that there's no ampersand preceding `f`. Once `pf` points to `f`, we can call `f` by writing either

```
(*pf)(i);
```

or

```
pf(i);
```

Arrays whose elements are function pointers have a surprising number of applications. For example, suppose that we're writing a program that displays a menu of commands for the user to choose from. We can write functions that implement these commands, then store pointers to the functions in an array:

```
void (*file_cmd[]) (void) = { new_cmd,
 open_cmd,
 close_cmd,
 close_all_cmd,
 save_cmd,
 save_as_cmd,
 save_all_cmd,
 print_cmd,
 exit_cmd
 };
```

If the user selects command `n`, where `n` falls between 0 and 8, we can subscript the `file_cmd` array and call the corresponding function:

```
(*file_cmd[n])(); /* or file_cmd[n](); */
```

Of course, we could get a similar effect with a `switch` statement. Using an array of function pointers gives us more flexibility, however, since the elements of the array can be changed as the program is running.

## PROGRAM Tabulating the Trigonometric Functions

The following program prints tables showing the values of the `cos`, `sin`, and `tan` functions (all three belong to `<math.h>`). The program is built around a function named `tabulate` that, when passed a function pointer `f`, prints a table showing the values of `f`.

```
tabulate.c /* Tabulates values of trigonometric functions */

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

void tabulate(double (*f)(double), double first,
 double last, double incr);

int main(void)
{
 double final, increment, initial;

 printf("Enter initial value: ");
 scanf("%lf", &initial);

 printf("Enter final value: ");
 scanf("%lf", &final);

 printf("Enter increment: ");
 scanf("%lf", &increment);

 printf("\n x cos(x)\n"
 " ----- ----- \n");
 tabulate(cos, initial, final, increment);

 printf("\n x sin(x)\n"
 " ----- ----- \n");
 tabulate(sin, initial, final, increment);

 printf("\n x tan(x)\n"
 " ----- ----- \n");
 tabulate(tan, initial, final, increment);

 return 0;
}

void tabulate(double (*f)(double), double first,
 double last, double incr)
{
 double x;
 int i, num_intervals;

 num_intervals = ceil((last - first) / incr);
 for (i = 0; i <= num_intervals; i++) {
 x = first + i * incr;
 printf("%10.5f %10.5f\n", x, (*f)(x));
 }
}
```

tabulate uses the ceil function, which also in <math.h>. When given an argument x of double type, ceil returns the smallest integer that's greater than or equal to x.

Here's what a session with `tabulate.c` might look like:

```
Enter initial value: 0
Enter final value: .5
Enter increment: .1
```

| x       | cos(x)  |
|---------|---------|
| 0.00000 | 1.00000 |
| 0.10000 | 0.99500 |
| 0.20000 | 0.98007 |
| 0.30000 | 0.95534 |
| 0.40000 | 0.92106 |
| 0.50000 | 0.87758 |

| x       | sin(x)  |
|---------|---------|
| 0.00000 | 0.00000 |
| 0.10000 | 0.09983 |
| 0.20000 | 0.19867 |
| 0.30000 | 0.29552 |
| 0.40000 | 0.38942 |
| 0.50000 | 0.47943 |

| x       | tan(x)  |
|---------|---------|
| 0.00000 | 0.00000 |
| 0.10000 | 0.10033 |
| 0.20000 | 0.20271 |
| 0.30000 | 0.30934 |
| 0.40000 | 0.42279 |
| 0.50000 | 0.54630 |

## 17.8 Restricted Pointers (C99)

This section and the next discuss two of C99's pointer-related features. Both are primarily of interest to advanced C programmers; most readers will want to skip these sections.

In C99, the keyword `restrict` may appear in the declaration of a pointer:

```
int * restrict p;
```

A pointer that's been declared using `restrict` is called a **restricted pointer**. The intent is that if `p` points to an object that is later modified, then that object is not accessed in any way other than through `p`. (Alternative ways to access the object include having another pointer to the same object or having `p` point to a named variable.) Having more than one way to access an object is often called **aliasing**.

Let's look at an example of the kind of behavior that restricted pointers are supposed to discourage. Suppose that `p` and `q` have been declared as follows:

```
int * restrict p;
int * restrict q;
```

Now suppose that `p` is made to point to a dynamically allocated block of memory:

```
p = malloc(sizeof(int));
```

(A similar situation would arise if `p` were assigned the address of a variable or an array element.) Normally it would be legal to copy `p` into `q` and then modify the integer through `q`:

```
q = p;
q = 0; / causes undefined behavior */
```

Because `p` is a restricted pointer, however, the effect of executing the statement `*q = 0;` is undefined. By making `p` and `q` point to the same object, we caused `*p` and `*q` to be aliases.

extern storage class ▶ 18.2

blocks ▶ 10.3

file scope ▶ 10.2

<string.h> header ▶ 23.6

If a restricted pointer `p` is declared as a local variable without the `extern` storage class, `restrict` applies only to `p` when the block in which `p` is declared is being executed. (Note that the body of a function is a block.) `restrict` can be used with function parameters of pointer type, in which case it applies only when the function is executing. When `restrict` is applied to a pointer variable with file scope, however, the restriction lasts for the entire execution of the program.

The exact rules for using `restrict` are rather complex; see the C99 standard for details. There are even situations in which an alias created from a restricted pointer is legal. For example, a restricted pointer `p` can be legally copied into another restricted pointer variable `q`, provided that `p` is local to a function and `q` is defined inside a block nested within the function's body.

To illustrate the use of `restrict`, let's look at the `memcpy` and `memmove` functions, which belong to the `<string.h>` header. `memcpy` has the following prototype in C99:

```
void *memcpy(void * restrict s1, const void * restrict s2,
 size_t n);
```

`memcpy` is similar to `strcpy`, except that it copies bytes from one object to another (`strcpy` copies characters from one string into another). `s2` points to the data to be copied, `s1` points to the destination of the copy, and `n` is the number of bytes to be copied. The use of `restrict` with both `s1` and `s2` indicates that the source of the copy and the destination shouldn't overlap. (It doesn't guarantee that they don't overlap, however.)

In contrast, `restrict` doesn't appear in the prototype for `memmove`:

```
void *memmove(void *s1, const void *s2, size_t n);
```

`memmove` does the same thing as `memcpy`: it copies bytes from one place to another. The difference is that `memmove` is guaranteed to work even if the source and destination overlap. For example, we could use `memmove` to shift the elements of an array by one position:

```
int a[100];
...
```

```
memmove(&a[0], &a[1], 99 * sizeof(int));
```

Prior to C99, there was no way to document the difference between `memcpy` and `memmove`. The prototypes for the two functions were nearly identical:

```
void *memcpy(void *s1, const void *s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
```

The use of `restrict` in the C99 version of `memcpy`'s prototype lets the programmer know that `s1` and `s2` should point to objects that don't overlap, or else the function isn't guaranteed to work.

Although using `restrict` in function prototypes is useful documentation, that's not the primary reason for its existence. `restrict` provides information to the compiler that may enable it to produce more efficient code—a process known as *optimization*. (The `register` storage class serves the same purpose.) Not every compiler attempts to optimize programs, however, and the ones that do normally allow the programmer to disable optimization. As a result, the C99 standard guarantees that `restrict` has no effect on the behavior of a program that conforms to the standard: if all uses of `restrict` are removed from such a program, it should behave the same.

Most programmers won't use `restrict` unless they're fine-tuning a program to achieve the best possible performance. Still, it's worth knowing about `restrict` because it appears in the C99 prototypes for a number of standard library functions.

## 17.9 Flexible Array Members (C99)

Every once in a while, we'll need to define a structure that contains an array of an unknown size. For example, we might want to store strings in a form that's different from the usual one. Normally, a string is an array of characters, with a null character marking the end. However, there are advantages to storing strings in other ways. One alternative is to store the length of the string along with the string's characters (but with no null character). The length and the characters could be stored in a structure such as this one:

```
struct vstring {
 int len;
 char chars[N];
};
```

Here `N` is a macro that represents the maximum length of a string. Using a fixed-length array such as this is undesirable, however, because it forces us to limit the length of the string, plus it wastes memory (since most strings won't need all `N` characters in the array).

C programmers have traditionally solved this problem by declaring the length of `chars` to be 1 (a dummy value) and then dynamically allocating each string:

```

struct vstring {
 int len;
 char chars[1];
};

...
struct vstring *str = malloc(sizeof(struct vstring) + n - 1);
str->len = n;

```

We're "cheating" by allocating more memory than the structure is declared to have (in this case, an extra  $n - 1$  characters), and then using the memory to store additional elements of the `chars` array. This technique has become so common over the years that it has a name: the "struct hack."

The struct hack isn't limited to character arrays; it has a variety of uses. Over time, it has become popular enough to be supported by many compilers. Some (including GCC) even allow the `chars` array to have zero length, which makes this trick a little more explicit. Unfortunately, the C89 standard doesn't guarantee that the struct hack will work, nor does it allow zero-length arrays.

In recognition of the struct hack's usefulness, C99 has a feature known as the **flexible array member** that serves the same purpose. When the last member of a structure is an array, its length may be omitted:

```

struct vstring {
 int len;
 char chars[]; /* flexible array member - C99 only */
};

```

The length of the `chars` array isn't determined until memory is allocated for a `vstring` structure, normally using a call of `malloc`:

```

struct vstring *str = malloc(sizeof(struct vstring) + n);
str->len = n;

```

In this example, `str` points to a `vstring` structure in which the `chars` array occupies  $n$  characters. The `sizeof` operator ignores the `chars` member when computing the size of the structure. (A flexible array member is unusual in that it takes up no space within a structure.)

A few special rules apply to a structure that contains a flexible array member. The flexible array member must appear last in the structure, and the structure must have at least one other member. Copying a structure that contains a flexible array member will copy the other members but not the flexible array itself.

A structure that contains a flexible array member is an **incomplete type**. An incomplete type is missing part of the information needed to determine how much memory it requires. Incomplete types, which are discussed further in one of the Q&A questions at the end of this chapter and in Section 19.3, are subject to various restrictions. In particular, an incomplete type (and hence a structure that contains a flexible array member) can't be a member of another structure or an element of an array. However, an array may contain pointers to structures that have a flexible array member; Programming Project 7 at the end of this chapter is built around such an array.

## Q & A

**Q: What does the NULL macro represent? [p. 415]**

A: NULL actually stands for 0. When we use 0 in a context where a pointer would be required, C compilers treat it as a null pointer instead of the integer 0. The NULL macro is provided merely to help avoid confusion. The assignment

```
p = 0;
```

could be assigning the value 0 to a numeric variable or assigning a null pointer to a pointer variable; we can't easily tell which. In contrast, the assignment

```
p = NULL;
```

makes it clear that p is a pointer.

**\*Q: In the header files that come with my compiler, NULL is defined as follows:**

```
#define NULL (void *) 0
```

**What's the advantage of casting 0 to void \*?**

A: This trick, which is allowed by the C standard, enables compilers to spot incorrect uses of the null pointer. For example, suppose that we try to assign NULL to an integer variable:

```
i = NULL;
```

If NULL is defined as 0, this assignment is perfectly legal. But if NULL is defined as (void \*) 0, the compiler can warn us that we're assigning a pointer to an integer variable.

Defining NULL as (void \*) 0 has a second, more important, advantage. Suppose that we call a function with a variable-length argument list and pass NULL as one of the arguments. If NULL is defined as 0, the compiler will incorrectly pass a zero integer value. (In an ordinary function call, NULL works fine because the compiler knows from the function's prototype that it expects a pointer. When a function has a variable-length argument list, however, the compiler lacks this knowledge.) If NULL is defined as (void \*) 0, the compiler will pass a null pointer.

To make matters even more confusing, some header files define NULL to be 0L (the long version of 0). This definition, like the definition of NULL as 0, is a holdover from C's earlier years, when pointers and integers were compatible. For most purposes, though, it really doesn't matter how NULL is defined; just think of it as a name for the null pointer.

**Q: Since 0 is used to represent the null pointer, I guess a null pointer is just an address with all zero bits, right?**

- A: Not necessarily. Each C compiler is allowed to represent null pointers in a different way, and not all compilers use a zero address. For example, some compilers use a nonexistent memory address for the null pointer; that way, attempting to access memory through a null pointer can be detected by the hardware.

How the null pointer is stored inside the computer shouldn't concern us; that's a detail for compiler experts to worry about. The important thing is that, when used in a pointer context, 0 is converted to the proper internal form by the compiler.

**Q: Is it acceptable to use `NULL` as a null character?**

- A: Definitely not. `NULL` is a macro that represents the null *pointer*, not the null *character*. Using `NULL` as a null character will work with some compilers, but not with all (since some define `NULL` as `(void *) 0`). In any event, using `NULL` as anything other than a pointer can lead to a great deal of confusion. If you want a name for the null character, define the following macro:

```
#define NUL '\0'
```

**\*Q: When my program terminates, I get the message “*Null pointer assignment*.” What does this mean?**

- A: This message, which is produced by programs compiled with some older DOS-based C compilers, indicates that the program has stored data in memory using a bad pointer (but not necessarily a null pointer). Unfortunately, the message isn't displayed until the program terminates, so there's no clue as to which statement caused the error. The “*Null pointer assignment*” message can be caused by a missing & in `scanf`:

```
scanf("%d", i); /* should have been scanf("%d", &i); */
```

Another possibility is an assignment involving a pointer that's uninitialized or null:

```
p = i; / p is uninitialized or null */
```

**\*Q: How does a program know that a “*null pointer assignment*” has occurred?**

- A: The message depends on the fact that, in the small and medium memory models, data is stored in a single segment, with addresses beginning at 0. The compiler leaves a “hole” at the beginning of the data segment—a small block of memory that's initialized to 0 but otherwise isn't used by the program. When the program terminates, it checks to see if any data in the “hole” area is nonzero. If so, it must have been altered through a bad pointer.

**Q: Is there any advantage to casting the return value of `malloc` or the other memory allocation functions? [p. 416]**

- A: Not usually. Casting the `void *` pointer that these functions return is unnecessary, since pointers of type `void *` are automatically converted to any pointer type upon assignment. The habit of casting the return value is a holdover from older versions of C, in which the memory allocation functions returned a `char *` value, making the cast necessary. Programs that are designed to be compiled as C++ code

may benefit from the cast, but that's about the only reason to do it.

In C89, there's actually a small advantage to *not* performing the cast. Suppose that we've forgotten to include the `<stdlib.h>` header in our program. When we call `malloc`, the compiler will assume that its return type is `int` (the default return value for any C function). If we don't cast the return value of `malloc`, a C89 compiler will produce an error (or at least a warning), since we're trying to assign an integer value to a pointer variable. On the other hand, if we cast the return value to a pointer, the program may compile, but likely won't run properly. With C99, this advantage disappears. Forgetting to include the `<stdlib.h>` header will cause an error when `malloc` is called, because C99 requires that a function be declared before it's called.

**Q:** **The `calloc` function initializes a memory block by setting its bits to zero. Does this mean that all data items in the block become zero? [p. 421]**

**A:** Usually, but not always. Setting an integer to zero bits always makes the integer zero. Setting a floating-point number to zero bits usually makes the number zero, but this isn't guaranteed—it depends on how floating-point numbers are stored. The story is the same for pointers; a pointer whose bits are zero isn't necessarily a null pointer.

**\*Q:** **I see how the structure tag mechanism allows a structure to contain a pointer to itself. But what if two structures each have a member that points to the other? [p. 425]**

**A:** Here's how we'd handle that situation:

```
struct s1; /* incomplete declaration of s1 */

struct s2 {
 ...
 struct s1 *p;
 ...
};

struct s1 {
 ...
 struct s2 *q;
 ...
};
```

incomplete types ▶ 19.3

The first declaration of `s1` creates an incomplete structure type, since we haven't specified the members of `s1`. The second declaration of `s1` "completes" the type by describing the members of the structure. Incomplete declarations of a structure type are permitted in C, although their uses are limited. Creating a pointer to such a type (as we did when declaring `p`) is one of these uses.

**Q:** **Calling `malloc` with the wrong argument—causing it to allocate too much memory or too little memory—seems to be a common error. Is there a safer way to use `malloc`? [p. 426]**

- A: Yes, there is. Some programmers use the following idiom when calling `malloc` to allocate memory for a single object:

```
p = malloc(sizeof(*p));
```

Since `sizeof(*p)` is the size of the object to which `p` will point, this statement guarantees that the correct amount of memory will be allocated. At first glance, this idiom looks fishy: it's likely that `p` is uninitialized, making the value of `*p` undefined. However, `sizeof` doesn't evaluate `*p`, it merely computes its size, so the idiom works even if `p` is uninitialized or contains a null pointer.

To allocate memory for an array with `n` elements, we can use a slightly modified version of the idiom:

```
p = malloc(n * sizeof(*p));
```

- Q: Why isn't the `qsort` function simply named `sort`? [p. 440]**

- A: The name `qsort` comes from the Quicksort algorithm published by C. A. R. Hoare in 1962 (and discussed in Section 9.6). Ironically, the C standard doesn't require that `qsort` use the Quicksort algorithm, although many versions of `qsort` do.

- Q: Isn't it necessary to cast `qsort`'s first argument to type `void *`, as in the following example? [p. 441]**

```
qsort((void *) inventory, num_parts, sizeof(struct part),
 compare_parts);
```

- A: No. A pointer of any type can be converted to `void *` automatically.

- \*Q: I want to use `qsort` to sort an array of integers, but I'm having trouble writing a comparison function. What's the secret?**

- A: Here's a version that works:

```
int compare_ints(const void *p, const void *q)
{
 return *(int *)p - *(int *)q;
}
```

Bizarre, eh? The expression `(int *)p` casts `p` to type `int *`, so `*(int *)p` would be the integer that `p` points to. A word of warning, though: Subtracting two integers may cause overflow. If the integers being sorted are completely arbitrary, it's safer to use `if` statements to compare `*(int *)p` with `*(int *)q`.

- \*Q: I needed to sort an array of strings, so I figured I'd just use `strcmp` as the comparison function. When I passed it to `qsort`, however, the compiler gave me a warning. I tried to fix the problem by embedding `strcmp` in a comparison function:**

```
int compare_strings(const void *p, const void *q)
{
 return strcmp(p, q);
}
```

**Now my program compiles, but `qsort` doesn't seem to sort the array. What am I doing wrong?**

- A: First, you can't pass `strcmp` itself to `qsort`, since `qsort` requires a comparison function with two `const void *` parameters. Your `compare_strings` function doesn't work because it incorrectly assumes that `p` and `q` are strings (`char *` pointers). In fact, `p` and `q` point to array elements containing `char *` pointers. To fix `compare_strings`, we'll cast `p` and `q` to type `char **`, then use the `*` operator to remove one level of indirection:

```
int compare_strings(const void *p, const void *q)
{
 return strcmp(*((char **)p), *((char **)q));
}
```

## Exercises

### Section 17.1

- Having to check the return value of `malloc` (or any other memory allocation function) each time we call it can be an annoyance. Write a function named `my_malloc` that serves as a “wrapper” for `malloc`. When we call `my_malloc` and ask it to allocate `n` bytes, it in turn calls `malloc`, tests to make sure that `malloc` doesn't return a null pointer, and then returns the pointer from `malloc`. Have `my_malloc` print an error message and terminate the program if `malloc` returns a null pointer.

### Section 17.2

- W 2. Write a function named `duplicate` that uses dynamic storage allocation to create a copy of a string. For example, the call

```
p = duplicate(str);
```

would allocate space for a string of the same length as `str`, copy the contents of `str` into the new string, and return a pointer to it. Have `duplicate` return a null pointer if the memory allocation fails.

### Section 17.3

- Write the following function:

```
int *create_array(int n, int initial_value);
```

The function should return a pointer to a dynamically allocated `int` array with `n` members, each of which is initialized to `initial_value`. The return value should be `NULL` if the array can't be allocated.

### Section 17.5

- Suppose that the following declarations are in effect:

```
struct point { int x, y; };
struct rectangle { struct point upper_left, lower_right; };
struct rectangle *p;
```

Assume that we want `p` to point to a `rectangle` structure whose upper left corner is at (10, 25) and whose lower right corner is at (20, 15). Write a series of statements that allocate such a structure and initialize it as indicated.

- W 5. Suppose that `f` and `p` are declared as follows:

```
struct {
 union {
 char a, b;
 int c;
 } d;
 int e[5];
} f, *p = &f;
```

Which of the following statements are legal?

- (a) `p->b = ' ';`
- (b) `p->e[3] = 10;`
- (c) `(*p).d.a = '*' ;`
- (d) `p->d->c = 20;`

6. Modify the `delete_from_list` function so that it uses only one pointer variable instead of two (`cur` and `prev`).

- W 7. The following loop is supposed to delete all nodes from a linked list and release the memory that they occupy. Unfortunately, the loop is incorrect. Explain what's wrong with it and show how to fix the bug.

```
for (p = first; p != NULL; p = p->next)
 free(p);
```

- W 8. Section 15.2 describes a file, `stack.c`, that provides functions for storing integers in a stack. In that section, the stack was implemented as an array. Modify `stack.c` so that a stack is now stored as a linked list. Replace the `contents` and `top` variables by a single variable that points to the first node in the list (the “top” of the stack). Write the functions in `stack.c` so that they use this pointer. Remove the `is_full` function, instead having `push` return either `true` (if memory was available to create a node) or `false` (if not).

9. True or false: If `x` is a structure and `a` is a member of that structure, then `(&x) ->a` is the same as `x.a`. Justify your answer.

10. Modify the `print_part` function of Section 16.2 so that its parameter is a *pointer* to a `part` structure. Use the `->` operator in your answer.

11. Write the following function:

```
int count_occurrences(struct node *list, int n);
```

The `list` parameter points to a linked list; the function should return the number of times that `n` appears in this list. Assume that the `node` structure is the one defined in Section 17.5.

12. Write the following function:

```
struct node *find_last(struct node *list, int n);
```

The `list` parameter points to a linked list. The function should return a pointer to the *last* node that contains `n`; it should return `NULL` if `n` doesn't appear in the list. Assume that the `node` structure is the one defined in Section 17.5.

13. The following function is supposed to insert a new node into its proper place in an ordered list, returning a pointer to the first node in the modified list. Unfortunately, the function

doesn't work correctly in all cases. Explain what's wrong with it and show how to fix it. Assume that the node structure is the one defined in Section 17.5.

```
struct node *insert_into_ordered_list(struct node *list,
 struct node *new_node)
{
 struct node *cur = list, *prev = NULL;
 while (cur->value <= new_node->value) {
 prev = cur;
 cur = cur->next;
 }
 prev->next = new_node;
 new_node->next = cur;
 return list;
}
```

### Section 17.6

14. Modify the `delete_from_list` function (Section 17.5) so that its first parameter has type `struct node **` (a pointer to a pointer to the first node in a list) and its return type is `void`. `delete_from_list` must modify its first argument to point to the list after the desired node has been deleted.

### Section 17.7

- W 15. Show the output of the following program and explain what it does.

```
#include <stdio.h>

int f1(int (*f)(int));
int f2(int i);

int main(void)
{
 printf("Answer: %d\n", f1(f2));
 return 0;
}

int f1(int (*f)(int))
{
 int n = 0;

 while ((*f)(n)) n++;
 return n;
}

int f2(int i)
{
 return i * i + i - 12;
}
```

16. Write the following function. The call `sum(g, i, j)` should return `g(i) + ... + g(j)`.

```
int sum(int (*f)(int), int start, int end);
```

- W 17. Let `a` be an array of 100 integers. Write a call of `qsort` that sorts only the *last* 50 elements in `a`. (You don't need to write the comparison function).
18. Modify the `compare_parts` function so that parts are sorted with their numbers in *descending* order.
19. Write a function that, when given a string as its argument, searches the following array of structures for a matching command name, then calls the function associated with that name.

```

struct {
 char *cmd_name;
 void (*cmd_pointer)(void);
} file_cmd[] =
{ {"new", new_cmd},
 {"open", open_cmd},
 {"close", close_cmd},
 {"close all", close_all_cmd},
 {"save", save_cmd},
 {"save as", save_as_cmd},
 {"save all", save_all_cmd},
 {"print", print_cmd},
 {"exit", exit_cmd}
};

```

## Programming Projects

- W 1. Modify the `inventory.c` program of Section 16.3 so that the `inventory` array is allocated dynamically and later reallocated when it fills up. Use `malloc` initially to allocate enough space for an array of 10 part structures. When the array has no more room for new parts, use `realloc` to double its size. Repeat the doubling step each time the array becomes full.
- W 2. Modify the `inventory.c` program of Section 16.3 so that the `p` (print) command calls `qsort` to sort the `inventory` array before it prints the parts.
- 3. Modify the `inventory2.c` program of Section 17.5 by adding an `e` (erase) command that allows the user to remove a part from the database.
- 4. Modify the `justify` program of Section 15.3 by rewriting the `line.c` file so that it stores the current line in a linked list. Each node in the list will store a single word. The `line` array will be replaced by a variable that points to the node containing the first word. This variable will store a null pointer whenever the line is empty.
- 5. Write a program that sorts a series of words entered by the user:

```

Enter word: foo
Enter word: bar
Enter word: baz
Enter word: quux
Enter word:

```

In sorted order: bar baz foo quux

Assume that each word is no more than 20 characters long. Stop reading when the user enters an empty word (i.e., presses Enter without entering a word). Store each word in a dynamically allocated string, using an array of pointers to keep track of the strings, as in the `remind2.c` program (Section 17.2). After all words have been read, sort the array (using any sorting technique) and then use a loop to print the words in sorted order. *Hint:* Use the `read_line` function to read each word, as in `remind2.c`.

- 6. Modify Programming Project 5 so that it uses `qsort` to sort the array of pointers.
- 7. (C99) Modify the `remind2.c` program of Section 17.2 so that each element of the `reminders` array is a pointer to a `vstring` structure (see Section 17.9) rather than a pointer to an ordinary string.

# 18 Declarations

*Making something variable is easy.  
Controlling duration of constancy is the trick.*

Declarations play a central role in C programming. By declaring variables and functions, we furnish vital information that the compiler will need in order to check a program for potential errors and translate it into object code.

Previous chapters have provided examples of declarations without going into full details; this chapter fills in the gaps. It explores the sophisticated options that can be used in declarations and reveals that variable declarations and function declarations have quite a bit in common. It also provides a firm grounding in the important concepts of storage duration, scope, and linkage.

Section 18.1 examines the syntax of declarations in their most general form, a topic that we've avoided up to this point. The next four sections focus on the items that appear in declarations: storage classes (Section 18.2), type qualifiers (Section 18.3), declarators (Section 18.4), and initializers (Section 18.5). Section 18.6 discusses the `inline` keyword, which can appear in C99 function declarations.

## 18.1 Declaration Syntax

Declarations furnish information to the compiler about the meaning of identifiers. When we write

```
int i;
```

we're informing the compiler that, in the current scope, the name `i` represents a variable of type `int`. The declaration

```
float f(float);
```

tells the compiler that `f` is a function that returns a `float` value and has one argument, also of type `float`.

In general, a declaration has the following appearance:

**declaration**

*declaration-specifiers declarators ;*

**Declaration specifiers** describe the properties of the variables or functions being declared. **Declarators** give their names and may provide additional information about their properties.

Declaration specifiers fall into three categories:

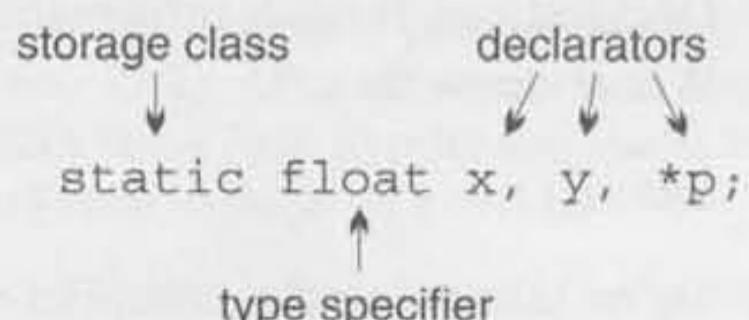
- **Storage classes.** There are four storage classes: `auto`, `static`, `extern`, and `register`. At most one storage class may appear in a declaration; if present, it should come first.
- **Type qualifiers.** In C89, there are only two type qualifiers: `const` and `volatile`. C99 has a third type qualifier, `restrict`. A declaration may contain zero or more type qualifiers.
- **Type specifiers.** The keywords `void`, `char`, `short`, `int`, `long`, `float`, `double`, `signed`, and `unsigned` are all type specifiers. These words may be combined as described in Chapter 7; the order in which they appear doesn't matter (`int unsigned long` is the same as `long unsigned int`). Type specifiers also include specifications of structures, unions, and enumerations (for example, `struct point { int x, y; }`, `struct { int x, y; }`, or `struct point`). Type names created using `typedef` are type specifiers as well.

C99

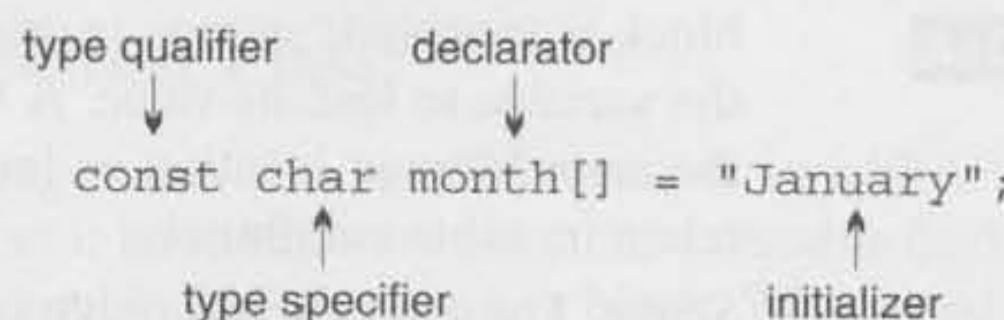
(C99 has a fourth kind of declaration specifier, the **function specifier**, which is used only in function declarations. This category has just one member, the keyword `inline`.) Type qualifiers and type specifiers should follow the storage class, but there are no other restrictions on their order. As a matter of style, I'll put type qualifiers before type specifiers.

Declarators include identifiers (names of simple variables), identifiers followed by `[]` (array names), identifiers preceded by `*` (pointer names), and identifiers followed by `()` (function names). Declarators are separated by commas. A declarator that represents a variable may be followed by an initializer.

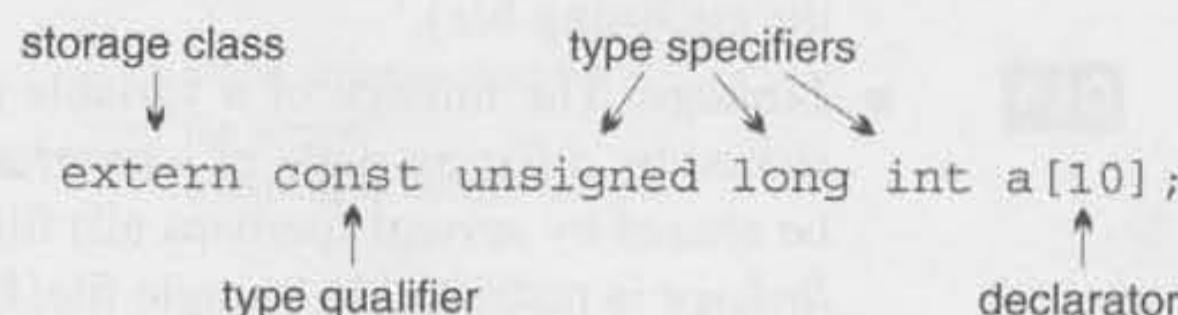
Let's look at a few examples that illustrate these rules. Here's a declaration with a storage class and three declarators:



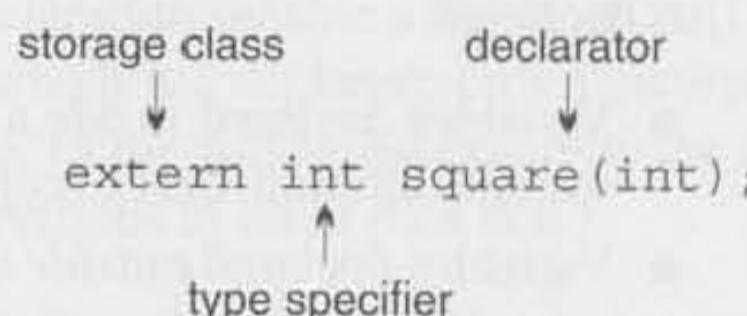
The following declaration has a type qualifier but no storage class. It also has an initializer:



The following declaration has both a storage class and a type qualifier. It also has three type specifiers; their order isn't important:



Function declarations, like variable declarations, may have a storage class, type qualifiers, and type specifiers. The following declaration has a storage class and a type specifier:



The next four sections cover storage classes, type qualifiers, declarators, and initializers in detail.

## 18.2 Storage Classes

Storage classes can be specified for variables and—to a lesser extent—functions and parameters. We'll concentrate on variables for now.

Recall from Section 10.3 that the term *block* refers to the body of a function (the part enclosed in braces) or a compound statement, possibly containing declarations. In C99, selection statements (`if` and `switch`) and iteration statements (`while`, `do`, and `for`)—along with the “inner” statements that they control—are considered to be blocks as well, although this is primarily a technicality.

**C99**

**Q&A**

### Properties of Variables

Every variable in a C program has three properties:

- **Storage duration.** The storage duration of a variable determines when memory is set aside for the variable and when that memory is released. Storage for a variable with *automatic storage duration* is allocated when the surrounding

**Q&A**

block is executed; storage is deallocated when the block terminates, causing the variable to lose its value. A variable with *static storage duration* stays at the same storage location as long as the program is running, allowing it to retain its value indefinitely.

- **Scope.** The scope of a variable is the portion of the program text in which the variable can be referenced. A variable can have either *block scope* (the variable is visible from its point of declaration to the end of the enclosing block) or *file scope* (the variable is visible from its point of declaration to the end of the enclosing file).

**Q&A**

- **Linkage.** The linkage of a variable determines the extent to which it can be shared by different parts of a program. A variable with *external linkage* may be shared by several (perhaps all) files in a program. A variable with *internal linkage* is restricted to a single file, but may be shared by the functions in that file. (If a variable with the same name appears in another file, it's treated as a different variable.) A variable with *no linkage* belongs to a single function and can't be shared at all.

The default storage duration, scope, and linkage of a variable depend on where it's declared:

- Variables declared *inside* a block (including a function body) have *automatic storage duration*, *block scope*, and *no linkage*.
- Variables declared *outside* any block, at the outermost level of a program, have *static storage duration*, *file scope*, and *external linkage*.

The following example shows the default properties of the variables *i* and *j*:

```
int i; // static storage duration, file scope, external linkage
void f(void)
{
 int j; // automatic storage duration, block scope, no linkage
}
```

For many variables, the default storage duration, scope, and linkage are satisfactory. When they aren't, we can alter these properties by specifying an explicit storage class: *auto*, *static*, *extern*, or *register*.

## The **auto** Storage Class

The *auto* storage class is legal only for variables that belong to a block. An *auto* variable has automatic storage duration (not surprisingly), block scope, and no linkage. The *auto* storage class is almost never specified explicitly, since it's the default for variables declared inside a block.

## The static Storage Class

The `static` storage class can be used with all variables, regardless of where they're declared, but it has a different effect on a variable declared outside a block than it does on a variable declared inside a block. When used *outside* a block, the word `static` specifies that a variable has internal linkage. When used *inside* a block, `static` changes the variable's storage duration from automatic to static. The following figure shows the effect of declaring `i` and `j` to be `static`:

```
static int i; static storage duration
 file scope
 internal linkage

void f(void)
{
 static int j; static storage duration
 block scope
 no linkage
}
```

When used in a declaration outside a block, `static` essentially hides a variable within the file in which it's declared; only functions that appear in the same file can see the variable. In the following example, the functions `f1` and `f2` both have access to `i`, but functions in other files don't:

```
static int i;

void f1(void)
{
 /* has access to i */
}

void f2(void)
{
 /* has access to i */
}
```

This use of `static` can help implement a technique known as information hiding.

A `static` variable declared within a block resides at the same storage location throughout program execution. Unlike automatic variables, which lose their values each time the program leaves the enclosing block, a `static` variable will retain its value indefinitely. `static` variables have some interesting properties:

- A `static` variable in a block is initialized only once, prior to program execution. An `auto` variable is initialized every time it comes into existence (provided, of course, that it has an initializer).
- Each time a function is called recursively, it gets a new set of `auto` variables. If it has a `static` variable, on the other hand, that variable is shared by all calls of the function.

- Although a function shouldn't return a pointer to an `auto` variable, there's nothing wrong with it returning a pointer to a `static` variable.

Declaring one of its variables to be `static` allows a function to retain information between calls in a “hidden” area that the rest of the program can't access. More often, however, we'll use `static` to make programs more efficient. Consider the following function:

```
char digit_to_hex_char(int digit)
{
 const char hex_chars[16] = "0123456789ABCDEF";
 return hex_chars[digit];
}
```

Each time the `digit_to_hex_char` function is called, the characters `0123456789ABCDEF` will be copied into the `hex_chars` array to initialize it. Now, let's make the array `static`:

```
char digit_to_hex_char(int digit)
{
 static const char hex_chars[16] = "0123456789ABCDEF";
 return hex_chars[digit];
}
```

Since `static` variables are initialized only once, we've improved the speed of `digit_to_hex_char`.

## The `extern` Storage Class

The `extern` storage class enables several source files to share the same variable. Section 15.2 covered the essentials of using `extern`, so I won't devote much space to it here. Recall that the declaration

```
extern int i;
```

informs the compiler that `i` is an `int` variable, but doesn't cause it to allocate memory for `i`. In C terminology, this declaration is not a *definition* of `i`; it merely informs the compiler that we need access to a variable that's defined elsewhere (perhaps later in the same file, or—more often—in another file). A variable can have many *declarations* in a program but should have only one *definition*.

There's one exception to the rule that an `extern` declaration of a variable isn't a definition. An `extern` declaration that initializes a variable serves as a definition of the variable. For example, the declaration

```
extern int i = 0;
```

is effectively the same as

```
int i = 0;
```

This rule prevents multiple `extern` declarations from initializing a variable in different ways.

### Q&A

A variable in an `extern` declaration always has static storage duration. The scope of the variable depends on the declaration's placement. If the declaration is inside a block, the variable has block scope; otherwise, it has file scope:

```
static storage duration
extern int i; file scope
 ? linkage

void f(void)
{
 static storage duration
 extern int j; block scope
 ? linkage
}
```

Determining the linkage of an `extern` variable is a bit harder. If the variable was declared `static` earlier in the file (outside of any function definition), then it has internal linkage. Otherwise (the normal case), the variable has external linkage.

## The `register` Storage Class

Using the `register` storage class in the declaration of a variable asks the compiler to store the variable in a register instead of keeping it in main memory like other variables. (A *register* is a storage area located in a computer's CPU. Data stored in a register can be accessed and updated faster than data stored in ordinary memory.) Specifying the storage class of a variable to be `register` is a request, not a command. The compiler is free to store a `register` variable in memory if it chooses.

The `register` storage class is legal only for variables declared in a block. A `register` variable has the same storage duration, scope, and linkage as an `auto` variable. However, a `register` variable lacks one property that an `auto` variable has: since registers don't have addresses, it's illegal to use the `&` operator to take the address of a `register` variable. This restriction applies even if the compiler has elected to store the variable in memory.

`register` is best used for variables that are accessed and/or updated frequently. For example, the loop control variable in a `for` statement is a good candidate for `register` treatment:

```
int sum_array(int a[], int n)
{
 register int i;
 int sum = 0;

 for (i = 0; i < n; i++)
 sum += a[i];
 return sum;
}
```

`register` isn't nearly as popular among C programmers as it once was. Today's compilers are much more sophisticated than early C compilers; many can determine automatically which variables would benefit the most from being kept in registers. Still, using `register` provides useful information that can help the compiler optimize the performance of a program. In particular, the compiler knows that a `register` variable can't have its address taken, and therefore can't be modified through a pointer. In this respect, the `register` keyword is related to C99's `restrict` keyword.

## The Storage Class of a Function

Function declarations (and definitions), like variable declarations, may include a storage class, but the only options are `extern` and `static`. The word `extern` at the beginning of a function declaration specifies that the function has external linkage, allowing it to be called from other files. `static` indicates internal linkage, limiting use of the function's name to the file in which it's defined. If no storage class is specified, the function is assumed to have external linkage.

Consider the following function declarations:

```
extern int f(int i);
static int g(int i);
int h(int i);
```

`f` has external linkage, `g` has internal linkage, and `h` (by default) has external linkage. Because it has internal linkage, `g` can't be called directly from outside the file in which it's defined. (Declaring `g` to be `static` doesn't completely prevent it from being called in another file; an indirect call via a function pointer is still possible.)

Declaring functions to be `extern` is like declaring variables to be `auto`—it serves no purpose. For that reason, I don't use `extern` in function declarations. Be aware, however, that some programmers use `extern` extensively, which certainly does no harm.

Declaring functions to be `static`, on the other hand, is quite useful. In fact, I recommend using `static` when declaring any function that isn't intended to be called from other files. The benefits of doing so include:

- **Easier maintenance.** Declaring a function `f` to be `static` guarantees that `f` isn't visible outside the file in which its definition appears. As a result, someone modifying the program later knows that changes to `f` won't affect functions in other files. (One exception: a function in another file that's passed a pointer to `f` might be affected by changes to `f`. Fortunately, that situation is easy to spot by examining the file in which `f` is defined, since the function that passes `f` must also be defined there.)
- **Reduced “name space pollution.”** Since functions declared `static` have internal linkage, their names can be reused in other files. Although we proba-

bly wouldn't deliberately reuse a function name for some other purpose, it can be hard to avoid in large programs. An excessive number of names with external linkage can result in what C programmers call "name space pollution": names in different files accidentally conflicting with each other. Using `static` helps prevent this problem.

Function parameters have the same properties as `auto` variables: automatic storage duration, block scope, and no linkage. The only storage class that can be specified for parameters is `register`.

## Summary

Now that we've covered the various storage classes, let's summarize what we know. The following program fragment shows all possible ways to include—or omit—storage classes in declarations of variables and parameters.

```
int a;
extern int b;
static int c;

void f(int d, register int e)
{
 auto int g;
 int h;
 static int i;
 extern int j;
 register int k;
}
```

Table 18.1 shows the properties of each variable and parameter in this example.

**Table 18.1**

Properties of Variables  
and Parameters

| Name | Storage Duration | Scope | Linkage  |
|------|------------------|-------|----------|
| a    | static           | file  | external |
| b    | static           | file  | †        |
| c    | static           | file  | internal |
| d    | automatic        | block | none     |
| e    | automatic        | block | none     |
| g    | automatic        | block | none     |
| h    | automatic        | block | none     |
| i    | static           | block | none     |
| j    | static           | block | †        |
| k    | automatic        | block | none     |

<sup>†</sup>The definitions of `b` and `j` aren't shown, so it's not possible to determine the linkage of these variables. In most cases, the variables will be defined in another file and will have external linkage.

Of the four storage classes, the most important are `static` and `extern`. `auto` has no effect, and modern compilers have made `register` less important.

## 18.3 Type Qualifiers

**C99**

restricted pointers ▶ 17.8

There are two type qualifiers: `const` and `volatile`. (C99 has a third type qualifier, `restrict`, which is used only with pointers.) Since the use of `volatile` is limited to low-level programming, I'll postpone discussing it until Section 20.3. `const` is used to declare objects that resemble variables but are “read-only”: a program may access the value of a `const` object, but can't change it. For example, the declaration

```
const int n = 10;
```

creates a `const` object named `n` whose value is 10. The declaration

```
const int tax_brackets[] = {750, 2250, 3750, 5250, 7000};
```

creates a `const` array named `tax_brackets`.

Declaring an object to be `const` has several advantages:

- It's a form of documentation: it alerts anyone reading the program to the read-only nature of the object.
- The compiler can check that the program doesn't inadvertently attempt to change the value of the object.
- When programs are written for certain types of applications (embedded systems, in particular), the compiler can use the word `const` to identify data to be stored in ROM (read-only memory).

At first glance, it might appear that `const` serves the same role as the `#define` directive, which we've used in previous chapters to create names for constants. There are significant differences between `#define` and `const`, however:

- We can use `#define` to create a name for a numerical, character, or string constant. `const` can be used to create read-only objects of *any* type, including arrays, pointers, structures, and unions.
- `const` objects are subject to the same scope rules as variables; constants created using `#define` aren't. In particular, we can't use `#define` to create a constant with block scope.
- The value of a `const` object, unlike the value of a macro, can be viewed in a debugger.
- Unlike macros, `const` objects can't be used in constant expressions. For example, we can't write

```
const int n = 10;
int a[n]; /*** WRONG ***/
```

**C99**

since array bounds must be constant expressions. (In C99, this example would

be legal if `a` has automatic storage duration—it would be treated as a variable-length array—but not if it has static storage duration.)

- It's legal to apply the address operator (`&`) to a `const` object, since it has an address. A macro doesn't have an address.

There are no absolute rules that dictate when to use `#define` and when to use `const`. I recommend using `#define` for constants that represent numbers or characters. That way, you'll be able to use the constants as array dimensions, in `switch` statements, and in other places where constant expressions are required.

## 18.4 Declarators

A declarator consists of an identifier (the name of the variable or function being declared), possibly preceded by the `*` symbol or followed by `[]` or `()`. By combining `*`, `[]`, and `()`, we can create declarators of mind-numbing complexity.

Before we look at the more complicated declarators, let's review the declarators that we've seen in previous chapters. In the simplest case, a declarator is just an identifier, like `i` in the following example:

```
int i;
```

Declarators may also contain the symbols `*`, `[]`, and `()`:

- A declarator that begins with `*` represents a pointer:

```
int *p;
```

- A declarator that ends with `[]` represents an array:

```
int a[10];
```

The brackets may be left empty if the array is a parameter, if it has an initializer, or if its storage class is `extern`:

```
extern int a[];
```

Since `a` is defined elsewhere in the program, the compiler doesn't need to know its length here. (In the case of a multidimensional array, only the first set of brackets can be empty.) C99 provides two additional options for what goes between the brackets in the declaration of an array parameter. One option is the keyword `static`, followed by an expression that specifies the array's minimum length. The other is the `*` symbol, which can be used in a function prototype to indicate a variable-length array argument. Section 9.3 discusses both C99 features.

- A declarator that ends with `()` represents a function:

```
int abs(int i);
void swap(int *a, int *b);
int find_largest(int a[], int n);
```

C99

C allows parameter names to be omitted in a function declaration:

```
int abs(int);
void swap(int *, int *);
int find_largest(int [], int);
```

The parentheses can even be left empty:

```
int abs();
void swap();
int find_largest();
```

The declarations in the last group specify the return types of the `abs`, `swap`, and `find_largest` functions, but provide no information about their arguments. Leaving the parentheses empty isn't the same as putting the word `void` between them, which indicates that there are no arguments. The empty-parentheses style of function declaration has largely disappeared. It's inferior to the prototype style introduced in C89, since it doesn't allow the compiler to check whether function calls have the right arguments.

If all declarators were as simple as these, C programming would be a snap. Unfortunately, declarators in actual programs often combine the `*`, `[]`, and `()` notations. We've seen examples of such combinations already. We know that

```
int *ap[10];
```

declares an array of 10 pointers to integers. We know that

```
float *fp(float);
```

declares a function that has a `float` argument and returns a pointer to a `float`. And, in Section 17.7, we learned that

```
void (*pf)(int);
```

declares a pointer to a function with an `int` argument and a `void` return type.

## Deciphering Complex Declarations

So far, we haven't had too much trouble understanding declarators. But what about declarators like the one in the following declaration?

```
int *(*x[10])(void);
```

This declarator combines `*`, `[]`, and `()`, so it's not obvious whether `x` is a pointer, an array, or a function.

Fortunately, there are two simple rules that will allow us to understand any declaration, no matter how convoluted:

- **Always read declarators from the inside out.** In other words, locate the identifier that's being declared, and start deciphering the declaration from there.

- When there's a choice, always favor [] and () over \*. If \* precedes the identifier and [] follows it, the identifier represents an array, not a pointer. Likewise, if \* precedes the identifier and () follows it, the identifier represents a function, not a pointer. (Of course, we can always use parentheses to override the normal priority of [] and () over \*.)

Let's apply these rules to our simple examples first. In the declaration

```
int *ap[10];
```

the identifier is ap. Since \* precedes ap and [] follows it, we give preference to [], so ap is an *array of pointers*. In the declaration

```
float *fp(float);
```

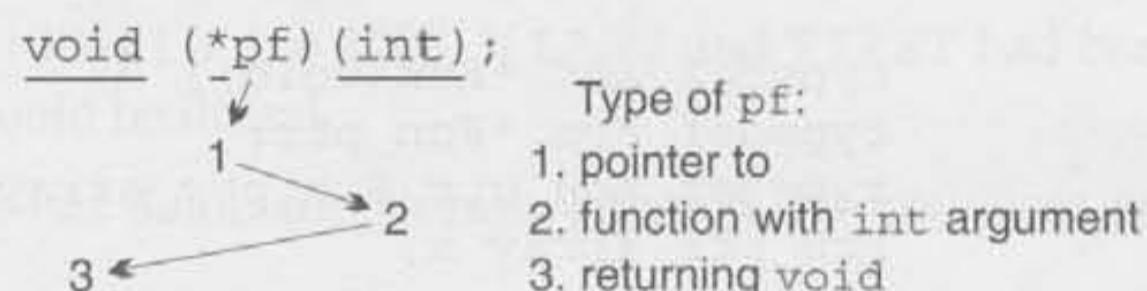
the identifier is fp. Since \* precedes fp and () follows it, we give preference to (), so fp is a *function that returns a pointer*.

The declaration

```
void (*pf)(int);
```

is a little trickier. Since \*pf is enclosed in parentheses, pf must be a pointer. But (\*pf) is followed by (int), so pf must point to a function with an int argument. The word void represents the return type of this function.

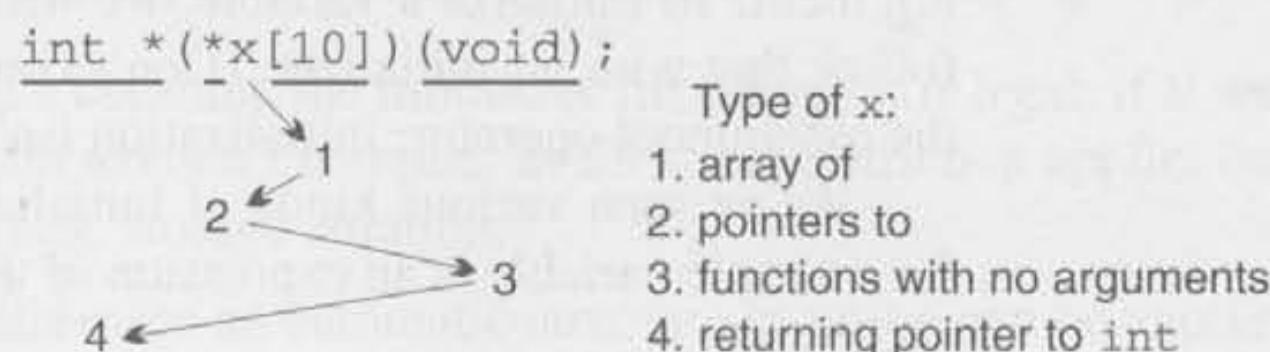
As the last example shows, understanding a complex declarator often involves zigzagging from one side of the identifier to the other:



Let's use this zigzagging technique to decipher the declaration given earlier:

```
int *(*x[10])(void);
```

First, we locate the identifier being declared (x). We see that x is preceded by \* and followed by [] ; since [] have priority over \*, we go right (x is an array). Next, we go left to find out the type of the elements in the array (pointers). Next, we go right to find out what kind of data the pointers point to (functions with no arguments). Finally, we go left to see what each function returns (a pointer to an int). Graphically, here's what the process looks like:



Mastering C declarations takes time and practice. The only good news is that there are certain things that can't be declared in C. Functions can't return arrays:

```
int f(int) [] ; /*** WRONG ***/
```

Functions can't return functions:

```
int g(int) (int) ; /*** WRONG ***/
```

Arrays of functions aren't possible, either:

```
int a[10] (int) ; /*** WRONG ***/
```

In each case, we can use pointers to get the desired effect. A function can't return an array, but it can return a *pointer* to an array. A function can't return a function, but it can return a *pointer* to a function. Arrays of functions aren't allowed, but an array may contain *pointers* to functions. (Section 17.7 has an example of such an array.)

## Using Type Definitions to Simplify Declarations

Some programmers use type definitions to help simplify complex declarations. Consider the declaration of *x* that we examined earlier in this section:

```
int *(*x[10]) (void) ;
```

To make *x*'s type easier to understand, we could use the following series of type definitions:

```
typedef int *Fcn(void) ;
typedef Fcn *Fcn_ptr;
typedef Fcn_ptr Fcn_ptr_array[10];
Fcn_ptr_array x;
```

If we read these lines in reverse order, we see that *x* has type *Fcn\_ptr\_array*, a *Fcn\_ptr\_array* is an array of *Fcn\_ptr* values, a *Fcn\_ptr* is a pointer to type *Fcn*, and a *Fcn* is a function that has no arguments and returns a pointer to an *int* value.

---

## 18.5 Initializers

For convenience, C allows us to specify initial values for variables as we're declaring them. To initialize a variable, we write the = symbol after its declarator, then follow that with an initializer. (Don't confuse the = symbol in a declaration with the assignment operator; initialization isn't the same as assignment.)

We've seen various kinds of initializers in previous chapters. The initializer for a simple variable is an expression of the same type as the variable:

```
int i = 5 / 2; /* i is initially 2 */
```

If the types don't match, C converts the initializer using the same rules as for conversion during assignment ➤ 7.4

```
int j = 5.5; /* converted to 5 */
```

The initializer for a pointer variable must be a pointer expression of the same type as the variable or of type `void *`:

```
int *p = &i;
```

The initializer for an array, structure, or union is usually a series of values enclosed in braces:

```
int a[5] = {1, 2, 3, 4, 5};
```

**C99**

In C99, brace-enclosed initializers can have other forms, thanks to designated initializers.

To complete our coverage of declarations, let's take a look at some additional rules that govern initializers:

- An initializer for a variable with static storage duration must be constant:

```
#define FIRST 1
#define LAST 100

static int i = LAST - FIRST + 1;
```

Since `LAST` and `FIRST` are macros, the compiler can compute the initial value of `i` ( $100 - 1 + 1 = 100$ ). If `LAST` and `FIRST` had been variables, the initializer would be illegal.

- If a variable has automatic storage duration, its initializer need not be constant:

```
int f(int n)
{
 int last = n - 1;
 ...
}
```

- A brace-enclosed initializer for an array, structure, or union must contain only constant expressions, never variables or function calls:

```
#define N 2

int powers[5] = {1, N, N * N, N * N * N, N * N * N * N};
```

**C99**

Since `N` is a constant, the initializer for `powers` is legal; if `N` were a variable, the program wouldn't compile. In C99, this restriction applies only if the variable has static storage duration.

- The initializer for an automatic structure or union can be another structure or union:

```
void g(struct part part1)
{
 struct part part2 = part1;
 ...
}
```

The initializer doesn't have to be a variable or parameter name, although it does need to be an expression of the proper type. For example, `part2`'s initializer could be `*p`, where `p` is of type `struct part *`, or `f(part1)`, where `f` is a function that returns a `part` structure.

## Uninitialized Variables

In previous chapters, we've implied that uninitialized variables have undefined values. That's not always true; the initial value of a variable depends on its storage duration:

- Variables with *automatic* storage duration have no default initial value. The initial value of an automatic variable can't be predicted and may be different each time the variable comes into existence.
- Variables with *static* storage duration have the value zero by default. Unlike memory allocated by `calloc`, which is simply set to zero bits, a static variable is correctly initialized based on its type: integer variables are initialized to 0, floating variables are initialized to 0.0, and pointer variables contain a null pointer.

calloc function ▶ 17.3

As a matter of style, it's better to provide initializers for static variables rather than rely on the fact that they're guaranteed to be zero. If a program accesses a variable that hasn't been initialized explicitly, someone reading the program later can't easily determine whether the variable is assumed to be zero or whether it's initialized by an assignment somewhere in the program.

---

## 18.6 Inline Functions (C99)

C99 function declarations have an additional option that doesn't exist in C89: they may contain the keyword `inline`. This keyword is a new breed of declaration specifier, distinct from storage classes, type qualifiers, and type specifiers. To understand the effect of `inline`, we'll need to visualize the machine instructions that are generated by a C compiler to handle the process of calling a function and returning from a function.

At the machine level, several instructions may need to be executed to prepare for the call, the call itself requires jumping to the first instruction in the function, and there may be additional instructions executed by the function itself as it begins to execute. If the function has arguments, they'll need to be copied (because C passes its arguments by value). Returning from a function requires a similar

amount of effort on both the part of the function that was called and the one that called it. The cumulative work required to call a function and later return from it is often referred to as “overhead,” since it’s extra work above and beyond what the function is really supposed to accomplish. Although the overhead of a function call slows the program by only a tiny amount, it may add up in certain situations, such as when a function is called millions or billions of times, when an older, slower processor is in use (as might be the case in an embedded system), or when a program has to meet very strict deadlines (as in a real-time system).

parameterized macros ➤ 14.3

In C89, the only way to avoid the overhead of a function call is to use a parameterized macro. Parameterized macros have certain drawbacks, though. C99 offers a better solution to this problem: create an *inline function*. The word “inline” suggests an implementation strategy in which the compiler replaces each call of the function by the machine instructions for the function. This technique avoids the usual overhead of a function call, although it may cause a minor increase in the size of the compiled program.

Declaring a function to be `inline` doesn’t actually force the compiler to “inline” the function, however. It merely suggests that the compiler should try to make calls of the function as fast as possible, perhaps by performing an inline expansion when the function is called. The compiler is free to ignore this suggestion. In this respect, `inline` is similar to the `register` and `restrict` keywords, which the compiler may use to improve the performance of a program but may also choose to ignore.

## Inline Definitions

An inline function has the keyword `inline` as one of its declaration specifiers:

```
inline double average(double a, double b)
{
 return (a + b) / 2;
}
```

Here’s where things get a bit complicated. `average` has external linkage, so other source files may contain calls of `average`. However, the definition of `average` isn’t considered to be an external definition by the compiler (it’s an *inline definition* instead), so attempting to call `average` from another file will be considered an error.

There are two ways to avoid this error. One option is to add the word `static` to the function definition:

```
static inline double average(double a, double b)
{
 return (a + b) / 2;
}
```

`average` now has internal linkage, so it can’t be called from other files. Other files may contain their own definitions of `average`, which might be the same as this definition or might be different.

The other option is to provide an external definition for `average` so that calls are permitted from other files. One way to do this is to write the `average` function a second time (without using `inline`) and put the second definition in a different source file. Doing so is legal, but it's not a good idea to have two versions of the same function, because we can't guarantee that they'll remain consistent when the program is modified.

Here's a better approach. First, we'll put the inline definition of `average` in a header file (let's name it `average.h`):

```
#ifndef AVERAGE_H
#define AVERAGE_H

inline double average(double a, double b)
{
 return (a + b) / 2;
}

#endif
```

Next, we'll create a matching source file, `average.c`:

```
#include "average.h"

extern double average(double a, double b);
```

Now, any file that needs to call the `average` function may simply include `average.h`, which contains the inline definition of `average`. The `average.c` file contains a prototype for `average` that uses the `extern` keyword, which causes the definition of `average` included from `average.h` to be treated as an external definition in `average.c`.

The general rule in C99 is that if all top-level declarations of a function in a particular file include `inline` but not `extern`, then the definition of the function in that file is `inline`. If the function is used anywhere in the program (including the file that contains its `inline` definition), then an external definition of the function will need to be provided by some other file. When the function is called, the compiler may choose to perform an ordinary call (using the function's external definition) or perform inline expansion (using the function's `inline` definition). There's no way to tell which choice the compiler will make, so it's crucial that the two definitions be consistent. The technique that we just discussed (using the `average.h` and `average.c` files) guarantees that the definitions are the same.

## Restrictions on Inline Functions

Since inline functions are implemented in a way that's quite different from ordinary functions, they're subject to different rules and restrictions. Variables with static storage duration are a particular problem for inline functions with external linkage. Consequently, C99 imposes the following restrictions on an inline function with external linkage (but not on one with internal linkage):

- The function may not define a modifiable `static` variable.
- The function may not contain references to variables with internal linkage.

Such a function is allowed to define a variable that is both `static` and `const`, but each inline definition of the function may create its own copy of the variable.

## Using Inline Functions with GCC

Some compilers, including GCC, supported inline functions prior to the C99 standard. As a result, their rules for using inline functions may vary from the standard. In particular, the scheme described earlier (using the `average.h` and `average.c` files) may not work with these compilers. Version 4.3 of GCC (not available at the time this book was written) is expected to support inline functions in the way described in the C99 standard.

Functions that are specified to be both `static` and `inline` should work fine, regardless of the version of GCC. This strategy is legal in C99 as well, so it's the safest bet. A `static inline` function can be used within a single file or placed in a header file and included into any source file that needs to call the function.

There's another way to share an inline function among multiple files that works with older versions of GCC but conflicts with C99. This technique involves putting a definition of the function in a header file, specifying that the function is both `extern` and `inline`, then including the header file into any source file that contains a call of the function. A second copy of the definition—without the words `extern` and `inline`—is placed in one of the source files. (That way, if the compiler is unable to “inline” the function for any reason, it will still have a definition.)

A final note about GCC: Functions are “inlined” only when optimization is requested via the `-O` command-line option.

## Q & A

**\*Q:** Why are selection statements and iteration statements (and their “inner” statements) considered to be blocks in C99? [p. 459]

**C99**

**A:** This rather surprising rule stems from a problem that can occur when compound literals are used in selection statements and iteration statements. The problem has to do with the storage duration of compound literals, so let's take a moment to discuss that issue first.

The C99 standard states that the object represented by a compound literal has static storage duration if the compound literal occurs outside the body of a function. Otherwise, it has automatic storage duration; as a result, the memory occupied by the object is deallocated at the end of the block in which the compound literal appears. Consider the following function, which returns a `point` structure created using a compound literal:

```
struct point create_point(int x, int y)
{
 return (struct point) {x, y};
}
```

This function works correctly, because the object created by the compound literal will be copied when the function returns. The original object will no longer exist, but the copy will remain. Now suppose that we change the function slightly:

```
struct point *create_point(int x, int y)
{
 return &(struct point) {x, y};
}
```

This version of `create_point` suffers from undefined behavior, because it returns a pointer to an object that has automatic storage duration and won't exist after the function returns.

Now let's return to the question we started with: Why are selection statements and iteration statements considered to be blocks? Consider the following example:

```
/* Example 1 - if statement without braces */

double *coefficients, value;

if (polynomial_selected == 1)
 coefficients = (double[3]) {1.5, -3.0, 6.0};
else
 coefficients = (double[3]) {4.5, 1.0, -3.5};
value = evaluate_polynomial(coefficients);
```

This program fragment apparently behaves in the desired fashion (but read on). `coefficients` will point to one of two objects created by compound literals, and this object will still exist at the time `evaluate_polynomial` is called. Now consider what happens if we put braces around the “inner” statements—the ones controlled by the `if` statement:

```
/* Example 2 - if statement with braces */

double *coefficients, value;

if (polynomial_selected == 1) {
 coefficients = (double[3]) {1.5, -3.0, 6.0};
} else {
 coefficients = (double[3]) {4.5, 1.0, -3.5};
}
value = evaluate_polynomial(coefficients);
```

Now we're in trouble. Each compound literal causes an object to be created, but that object exists only within the block formed by the braces that enclose the statement in which the literal appears. By the time `evaluate_polynomial` is called, `coefficients` points to an object that no longer exists. The result: undefined behavior.

The creators of C99 were unhappy with this state of affairs, because programmers were unlikely to expect that simply adding braces within an `if` statement would cause undefined behavior. To avoid the problem, they decided that the inner statements would always be considered blocks. As a result, Example 1 and Example 2 are equivalent, with both exhibiting undefined behavior.

A similar problem can arise when a compound literal is part of the controlling expression of a selection statement or iteration statement. For this reason, each entire selection statement and iteration statement is considered to be a block as well (as though an invisible set of braces surrounds the entire statement). So, for example, an `if` statement with an `else` clause consists of three blocks: each of the two inner statements is a block, as is the entire `if` statement.

**Q: You said that storage for a variable with automatic storage duration is allocated when the surrounding block is executed. Is this true for C99's variable-length arrays? [p. 460]**

**A:** No. Storage for a variable-length array isn't allocated at the beginning of the surrounding block, because the length of the array isn't yet known. Instead, it's allocated when the declaration of the array is reached during the execution of the block. In this respect, variable-length arrays are different from all other automatic variables.

**Q: What exactly is the difference between "scope" and "linkage"? [p. 460]**

**A:** Scope is for the benefit of the compiler, while linkage is for the benefit of the linker. The compiler uses the scope of an identifier to determine whether or not it's legal to refer to the identifier at a given point in a file. When the compiler translates a source file into object code, it notes which names have external linkage, eventually storing these names in a table inside the object file. Thus, the linker has access to names with external linkage; names with internal linkage or no linkage are invisible to the linker.

**Q: I don't understand how a name could have block scope but external linkage. Could you elaborate? [p. 463]**

**A:** Certainly. Suppose that one source file defines a variable `i`:

```
int i;
```

Let's assume that the definition of `i` lies outside any function, so `i` has external linkage by default. In another file, there's a function `f` that needs to access `i`, so the body of `f` declares `i` as `extern`:

```
void f(void)
{
 extern int i;
 ...
}
```

In the first file, `i` has file scope. Within `f`, however, `i` has block scope. If other functions besides `f` need access to `i`, they'll need to declare it separately. (Or we

can simply move the declaration of *i* outside *f* so that *i* has file scope.) What's confusing about this entire business is that each declaration or definition of *i* establishes a different scope; sometimes it's file scope, and sometimes it's block scope.

**\*Q:** Why can't `const` objects be used in constant expressions? `const` means "constant," right? [p. 466]

A: In C, `const` means "read-only," not "constant." Let's look at a few examples that illustrate why `const` objects can't be used in constant expressions.

To start with, a `const` object might only be constant during its *lifetime*, not throughout the execution of the program. Suppose that a `const` object is declared inside a function:

```
void f(int n)
{
 const int m = n / 2;
 ...
}
```

When *f* is called, *m* will be initialized to the value of *n* / 2. The value of *m* will then remain constant until *f* returns. When *f* is called the next time, *m* will likely be given a different value. That's where the problem arises. Suppose that *m* appears in a `switch` statement:

```
void f(int n)
{
 const int m = n / 2;
 ...
 switch (...) {
 ...
 case m: ... /* *** WRONG ***/
 ...
 }
 ...
}
```

The value of *m* won't be known until *f* is called, which violates C's rule that the values of case labels must be constant expressions.

Next, let's look at `const` objects declared outside blocks. These objects have external linkage and can be shared among files. If C allowed the use of `const` objects in constant expressions, we could easily find ourselves in the following situation:

```
extern const int n;
int a[n]; /* *** WRONG ***/
```

*n* is probably defined in another file, making it impossible for the compiler to determine *a*'s length. (I'm assuming that *a* is an external variable, so it can't be a variable-length array.)

volatile type qualifier ▶ 20.3

If that's not enough to convince you, consider this: If a `const` object is also declared to be `volatile`, its value may change at any time during execution. Here's an example from the C standard:

```
extern const volatile int real_time_clock;
```

The `real_time_clock` variable may not be changed by the program (because it's declared `const`), yet its value may change via some other mechanism (because it's declared `volatile`).

**Q: Why is the syntax of declarators so odd?**

- A: Declarations are intended to mimic use. A pointer declarator has the form `*p`, which matches the way the indirection operator will later be applied to `p`. An array declarator has the form `a [...]`, which matches the way the array will later be subscripted. A function declarator has the form `f (...)`, which matches the syntax of a function call. This reasoning extends to even the most complicated declarators. Consider the `file_cmd` array of Section 17.7, whose elements are pointers to functions. The declarator for `file_cmd` has the form

```
(*file_cmd[]) (void)
```

and a call of one of the functions has the form

```
(*file_cmd[n]) () ;
```

The parentheses, brackets, and `*` are in identical positions.

## Exercises

### Section 18.1

- For each of the following declarations, identify the storage class, type qualifiers, type specifiers, declarators, and initializers.
  - `static char **lookup(int level);`
  - `volatile unsigned long io_flags;`
  - `extern char *file_name[MAX_FILES], path[];`
  - `static const char token_buf[] = "";`

### Section 18.2

- Answer each of the following questions with `auto`, `extern`, `register`, and/or `static`.
  - Which storage class is used primarily to indicate that a variable or function can be shared by several files?
  - Suppose that a variable `x` is to be shared by several functions in one file but hidden from functions in other files. Which storage class should `x` be declared to have?
  - Which storage classes can affect the storage duration of a variable?
- List the storage duration (static or automatic), scope (block or file), and linkage (internal, external, or none) of each variable and parameter in the following file:

```

extern float a;

void f(register double b)
{
 static int c;
 auto char d;
}

```

- W 4. Let *f* be the following function. What will be the value of *f*(10) if *f* has never been called before? What will be the value of *f*(10) if *f* has been called five times previously?

```

int f(int i)
{
 static int j = 0;
 return i * j++;
}

```

5. State whether each of the following statements is true or false. Justify each answer.

- (a) Every variable with static storage duration has file scope.
- (b) Every variable declared inside a function has no linkage.
- (c) Every variable with internal linkage has static storage duration.
- (d) Every parameter has block scope.

6. The following function is supposed to print an error message. Each message is preceded by an integer, indicating the number of times the function has been called. Unfortunately, the function always displays 1 as the number of the error message. Locate the error and show how to fix it without making any changes outside the function.

```

void print_error(const char *message)
{
 int n = 1;
 printf("Error %d: %s\n", n++, message);
}

```

### Section 18.3

7. Suppose that we declare *x* to be a `const` object. Which one of the following statements about *x* is *false*?
- (a) If *x* is of type `int`, it can be used as the value of a case label in a `switch` statement.
  - (b) The compiler will check that no assignment is made to *x*.
  - (c) *x* is subject to the same scope rules as variables.
  - (d) *x* can be of any type.

### Section 18.4

- W 8. Write a complete description of the type of *x* as specified by each of the following declarations.
- (a) `char (*x[10])(int);`
  - (b) `int (*x(int))[5];`
  - (c) `float *(*x(void))(int);`
  - (d) `void (*x(int, void (*y)(int)))(int);`
9. Use a series of type definitions to simplify each of the declarations in Exercise 8.
- W 10. Write declarations for the following variables and functions:
- (a) *p* is a pointer to a function with a character pointer argument that returns a character pointer.

- (b) *f* is a function with two arguments: *p*, a pointer to a structure with tag *t*, and *n*, a long integer. *f* returns a pointer to a function that has no arguments and returns nothing.
- (c) *a* is an array of four pointers to functions that have no arguments and return nothing. The elements of *a* initially point to functions named *insert*, *search*, *update*, and *print*.
- (d) *b* is an array of 10 pointers to functions with two *int* arguments that return structures with tag *t*.
11. In Section 18.4, we saw that the following declarations are illegal:
- ```
int f(int) [] ;      /* functions can't return arrays */
int g(int)(int) ;    /* functions can't return functions */
int a[10](int) ;     /* array elements can't be functions */
```
- We can, however, achieve similar effects by using pointers: a function can return a *pointer* to the first element in an array, a function can return a *pointer* to a function, and the elements of an array can be *pointers* to functions. Revise each of these declarations accordingly.
- *12. (a) Write a complete description of the type of the function *f*, assuming that it's declared as follows:
- ```
int (*f(float (*)(long), char *)) (double);
```
- (b) Give an example showing how *f* would be called.

**Section 18.5**

- W 13. Which of the following declarations are legal? (Assume that *PI* is a macro that represents 3.14159.)
- (a) `char c = 65;`  
 (b) `static int i = 5, j = i * i;`  
 (c) `double d = 2 * PI;`  
 (d) `double angles[] = {0, PI / 2, PI, 3 * PI / 2};`
14. Which kind of variables cannot be initialized?
- (a) Array variables  
 (b) Enumeration variables  
 (c) Structure variables  
 (d) Union variables  
 (e) None of the above
- W 15. Which property of a variable determines whether or not it has a default initial value?
- (a) Storage duration  
 (b) Scope  
 (c) Linkage  
 (d) Type



# 19 Program Design

*Wherever there is modularity there is the potential for misunderstanding:  
Hiding information implies a need to check communication.*

It's obvious that real-world programs are larger than the examples in this book, but you may not realize just how much larger. Faster CPUs and larger main memories have made it possible to write programs that would have been impractical just a few years ago. The popularity of graphical user interfaces has added greatly to the average length of a program. Most full-featured programs today are at least 100,000 lines long. Million-line programs are commonplace, and it's not unheard-of for a program to have 10 million lines or more.

## Q&A

Although C wasn't designed for writing large programs, many large programs have in fact been written in C. It's tricky, and it requires a great deal of care, but it can be done. In this chapter, I'll discuss techniques that have proved to be helpful for writing large programs and show which C features (the `static` storage class, for example) are especially useful.

Writing large programs (often called "programming-in-the-large") is quite different from writing small ones—it's like the difference between writing a term paper (10 pages double-spaced, of course) and a 1000-page book. A large program requires more attention to style, since many people will be working on it. It requires careful documentation. It requires planning for maintenance, since it will likely be modified many times.

Above all, a large program requires careful design and much more planning than a small program. As Alan Kay, the designer of the Smalltalk programming language, puts it, "You can build a doghouse out of anything." A doghouse can be built without any particular design, using whatever materials are at hand. A house for humans, on the other hand, is too complex to just throw together.

Chapter 15 discussed writing large programs in C, but it concentrated on language details. In this chapter, we'll revisit the topic, this time focusing on techniques for good program design. A complete discussion of program design issues is obviously beyond the scope of this book. However, I'll try to cover—briefly—

some important concepts in program design and show how to use them to create C programs that are readable and maintainable.

Section 19.1 discusses how to view a C program as a collection of modules that provide services to each other. We'll then see how the concepts of information hiding (Section 19.2) and abstract data types (Section 19.3) can improve modules. By focusing on a single example (a stack data type), Section 19.4 illustrates how an abstract data type can be defined and implemented in C. Section 19.5 describes some limitations of C for defining abstract data types and shows how to work around them.

## 19.1 Modules

When designing a C program (or a program in any other language, for that matter), it's often useful to view it as a number of independent **modules**. A module is a collection of services, some of which are made available to other parts of the program (the *clients*). Each module has an *interface* that describes the available services. The details of the module—including the source code for the services themselves—are stored in the module's *implementation*.

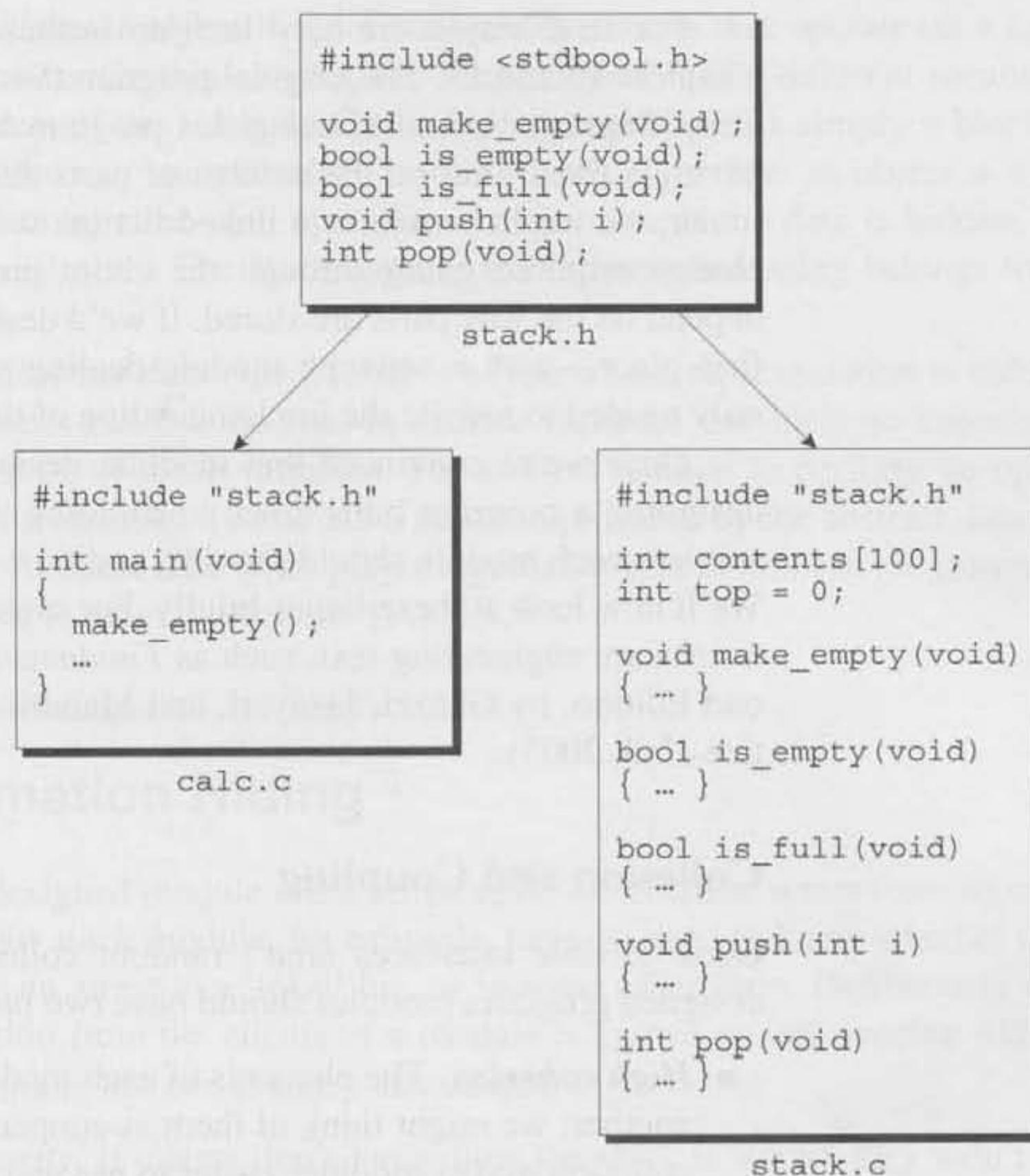
In the context of C, “services” are functions. The interface of a module is a header file containing prototypes for the functions that will be made available to clients (source files). The implementation of a module is a source file that contains definitions of the module's functions.

To illustrate this terminology, let's look at the calculator program that was sketched in Sections 15.1 and 15.2. This program consists of the file `calc.c`, which contains the main function, and a stack module, which is stored in the files `stack.h` and `stack.c` (see the figure at the top of the next page). `calc.c` is a *client* of the stack module. `stack.h` is the *interface* of the stack module; it supplies everything the client needs to know about the module. `stack.c` is the *implementation* of the module; it contains definitions of the stack functions as well as declarations of the variables that make up the stack.

The C library is itself a collection of modules. Each header in the library serves as the interface to a module. `<stdio.h>`, for example, is the interface to a module containing I/O functions, while `<string.h>` is the interface to a module containing string-handling functions.

Dividing a program into modules has several advantages:

- **Abstraction.** If modules are properly designed, we can treat them as **abstractions**; we know what they do, but we don't worry about the details of how they do it. Thanks to abstraction, it's not necessary to understand how the entire program works in order to make changes to one part of it. What's more, abstraction makes it easier for several members of a team to work on the same program. Once the interfaces for the modules have been agreed upon, the responsibility for implementing each module can be delegated to a partic-



ular person. Team members can then work largely independently of one another.

- **Reusability.** Any module that provides services is potentially reusable in other programs. Our stack module, for example, is reusable. Since it's often hard to anticipate the future uses of a module, it's a good idea to design modules for reusability.
- **Maintainability.** A small bug will usually affect only a single module implementation, making the bug easier to locate and fix. Once the bug has been fixed, rebuilding the program requires only a recompilation of the module implementation (followed by linking the entire program). On a larger scale, we could replace an entire module implementation, perhaps to improve performance or when transporting the program to a different platform.

Although all these advantages are important, maintainability is the most critical. Most real-world programs are in service over a period of years, during which bugs are discovered, enhancements are made, and modifications are made to meet changing requirements. Designing a program in a modular fashion makes maintenance much easier. Maintaining a program should be like maintaining a car—fixing a flat tire shouldn't require overhauling the engine.

For an example, we need look no further than the inventory program of Chapters 16 and 17. The original program (Section 16.3) stored part records in an array. Suppose that, after using this program for a while, the customer objects to having a fixed limit on the number of parts that can be stored. To satisfy the customer, we might switch to a linked list (as we did in Section 17.5). Making this change required going through the entire program, looking for all places that depend on the way parts are stored. If we'd designed the program differently in the first place—with a separate module dealing with part storage—we would have only needed to rewrite the implementation of that module, not the entire program.

Once we're convinced that modular design is the way to go, the process of designing a program boils down to deciding what modules it should have, what services each module should provide, and how the modules should be interrelated. We'll now look at these issues briefly. For more information about design, consult a software engineering text, such as *Fundamentals of Software Engineering*, Second Edition, by Ghezzi, Jazayeri, and Mandrioli (Upper Saddle River, N.J.: Prentice-Hall, 2003).

## Cohesion and Coupling

Good module interfaces aren't random collections of declarations. In a well-designed program, modules should have two properties:

- **High cohesion.** The elements of each module should be closely related to one another; we might think of them as cooperating toward a common goal. High cohesion makes modules easier to use and makes the entire program easier to understand.
- **Low coupling.** Modules should be as independent of each other as possible. Low coupling makes it easier to modify the program and reuse modules.

Does the calculator program have these properties? The stack module is clearly cohesive: its functions represent operations on a stack. There's little coupling in the program. The `calc.c` file depends on `stack.h` (and `stack.c` depends on `stack.h`, of course), but there are no other apparent dependencies.

## Types of Modules

Because of the need for high cohesion and low coupling, modules tend to fall into certain typical categories:

- A **data pool** is a collection of related variables and/or constants. In C, a module of this type is often just a header file. From a design standpoint, putting variables in header files isn't usually a good idea, but collecting related constants in a header file can often be useful. In the C library, `<float.h>` and `<limits.h>` are both data pools.
- A **library** is a collection of related functions. The `<string.h>` header, for example, is the interface to a library of string-handling functions.

`<float.h>` header ▶ 23.1  
`<limits.h>` header ▶ 23.2

- An *abstract object* is a collection of functions that operate on a hidden data structure. (In this chapter, the term “object” has a different meaning than in the rest of the book. In C terminology, an object is simply a block of memory that can store a value. In this chapter, however, an object is a collection of data bundled with operations on the data. If the data is hidden, the object is “abstract.”) The stack module we’ve been discussing belongs to this category.
- An *abstract data type (ADT)* is a type whose representation is hidden. Client modules can use the type to declare variables, but have no knowledge of the structure of those variables. For a client module to perform an operation on such a variable, it must call a function provided by the abstract data type module. Abstract data types play a significant role in modern programming; we’ll return to them in Sections 19.3–19.5.

## 19.2 Information Hiding

A well-designed module often keeps some information secret from its clients. Clients of our stack module, for example, have no need to know whether the stack is stored in an array, in a linked list, or in some other form. Deliberately concealing information from the clients of a module is known as *information hiding*. Information hiding has two primary advantages:

- **Security.** If clients don’t know how the stack is stored, they won’t be able to corrupt it by tampering with its internal workings. To perform operations on the stack, they’ll have to call functions that are provided by the module itself—functions that we’ve written and tested.
- **Flexibility.** Making changes—no matter how large—to a module’s internal workings won’t be difficult. For example, we could implement the stack as an array at first, then later switch to a linked list or other representation. We’ll have to rewrite the implementation of the module, of course, but—if the module was designed properly—we won’t have to alter the module’s interface.

static storage class ▶ 18.2

In C, the major tool for enforcing information hiding is the `static` storage class. Declaring a variable with file scope to be `static` gives it internal linkage, thus preventing it from being accessed from other files, including clients of the module. (Declaring a function to be `static` is also useful—the function can be directly called only by other functions in the same file.)

### A Stack Module

To see the benefits of information hiding, let’s look at two implementations of a stack module, one using an array and the other a linked list. The module’s header file will have the following appearance:

```
stack.h #ifndef STACK_H
#define STACK_H

#include <stdbool.h> /* C99 only */

void make_empty(void);
bool is_empty(void);
bool is_full(void);
void push(int i);
int pop(void);

#endif
```

I've included C99's `<stdbool.h>` header so that the `is_empty` and `is_full` functions can return a `bool` result rather than an `int` value.

Let's first use an array to implement the stack:

```
stack1.c #include <stdio.h>
#include <stdlib.h>
#include "stack.h"

#define STACK_SIZE 100

static int contents[STACK_SIZE];
static int top = 0;

static void terminate(const char *message)
{
 printf("%s\n", message);
 exit(EXIT_FAILURE);
}

void make_empty(void)
{
 top = 0;
}

bool is_empty(void)
{
 return top == 0;
}

bool is_full(void)
{
 return top == STACK_SIZE;
}

void push(int i)
{
 if (is_full())
 terminate("Error in push: stack is full.");
 contents[top++] = i;
}
```

```

int pop(void)
{
 if (is_empty())
 terminate("Error in pop: stack is empty.");
 return contents[--top];
}

```

The variables that make up the stack (`contents` and `top`) are both declared `static`, since there's no reason for the rest of the program to access them directly. The `terminate` function is also declared `static`. This function isn't part of the module's interface; instead, it's designed for use solely within the implementation of the module.

As a matter of style, some programmers use macros to indicate which functions and variables are “public” (accessible elsewhere in the program) and which are “private” (limited to a single file):

```

#define PUBLIC /* empty */
#define PRIVATE static

```

The reason for writing `PRIVATE` instead of `static` is that the latter has more than one use in C; `PRIVATE` makes it clear that we're using it to enforce information hiding. Here's what the stack implementation would look like if we were to use `PUBLIC` and `PRIVATE`:

```

PRIVATE int contents[STACK_SIZE];
PRIVATE int top = 0;

PRIVATE void terminate(const char *message) { ... }

PUBLIC void make_empty(void) { ... }

PUBLIC bool is_empty(void) { ... }

PUBLIC bool is_full(void) { ... }

PUBLIC void push(int i) { ... }

PUBLIC int pop(void) { ... }

```

Now we'll switch to a linked-list implementation of the stack module:

```

stack2.c #include <stdio.h>
#include <stdlib.h>
#include "stack.h"

struct node {
 int data;
 struct node *next;
};

static struct node *top = NULL;

```

```

static void terminate(const char *message)
{
 printf("%s\n", message);
 exit(EXIT_FAILURE);
}

void make_empty(void)
{
 while (!is_empty())
 pop();
}

bool is_empty(void)
{
 return top == NULL;
}

bool is_full(void)
{
 return false;
}

void push(int i)
{
 struct node *new_node = malloc(sizeof(struct node));
 if (new_node == NULL)
 terminate("Error in push: stack is full.");

 new_node->data = i;
 new_node->next = top;
 top = new_node;
}

int pop(void)
{
 struct node *old_top;
 int i;

 if (is_empty())
 terminate("Error in pop: stack is empty.");

 old_top = top;
 i = top->data;
 top = top->next;
 free(old_top);
 return i;
}

```

Note that the `is_full` function returns `false` every time it's called. A linked list has no limit on its size, so the stack will never be full. It's possible (but not likely) that the program might run out of memory, which will cause the `push` function to fail, but there's no easy way to test for that condition in advance.

Our stack example shows clearly the advantage of information hiding: it

doesn't matter whether we use `stack1.c` or `stack2.c` to implement the stack module. Both versions match the module's interface, so we can switch from one to the other without having to make changes elsewhere in the program.

## 19.3 Abstract Data Types

A module that serves as an abstract object, like the stack module in the previous section, has a serious disadvantage: there's no way to have multiple instances of the object (more than one stack, in this case). To accomplish this, we'll need to go a step further and create a new *type*.

Once we've defined a `Stack` type, we'll be able to have as many stacks as we want. The following fragment illustrates how we could have two stacks in the same program:

```
Stack s1, s2;

make_empty(&s1);
make_empty(&s2);
push(&s1, 1);
push(&s2, 2);
if (!is_empty(&s1))
 printf("%d\n", pop(&s1)); /* prints "1" */
```

We're not really sure what `s1` and `s2` are (structures? pointers?), but it doesn't matter. To clients, `s1` and `s2` are *abstractions* that respond to certain operations (`make_empty`, `is_empty`, `is_full`, `push`, and `pop`).

Let's convert our `stack.h` header so that it provides a `Stack` type, where `Stack` is a structure. Doing so will require adding a `Stack` (or `Stack *`) parameter to each function. The header will now look like this (changes to `stack.h` are in **bold**; unchanged portions of the header aren't shown):

```
#define STACK_SIZE 100

typedef struct {
 int contents[STACK_SIZE];
 int top;
} Stack;

void make_empty(Stack *s);
bool is_empty(const Stack *s);
bool is_full(const Stack *s);
void push(Stack *s, int i);
int pop(Stack *s);
```

The `stack` parameters to `make_empty`, `push`, and `pop` need to be pointers, since these functions modify the stack. The parameter to `is_empty` and `is_full` doesn't need to be a pointer, but I've made it one anyway. Passing these functions a `Stack pointer` instead of a `Stack value` is more efficient, since the latter would result in a structure being copied.

## Encapsulation

Unfortunately, `Stack` isn't an *abstract* data type, since `stack.h` reveals what the `Stack` type really is. Nothing prevents clients from using a `Stack` variable as a structure:

```
Stack s1;

s1.top = 0;
s1.contents[top++] = 1;
```

Providing access to the `top` and `contents` members allows clients to corrupt the stack. Worse still, we won't be able to change the way stacks are stored without having to assess the effect of the change on clients.

What we need is a way to prevent clients from knowing how the `Stack` type is represented. C has only limited support for *encapsulating* types in this way. Newer C-based languages, including C++, Java, and C#, are better equipped for this purpose.

## Incomplete Types

The only tool that C gives us for encapsulation is the *incomplete type*. (Incomplete types were mentioned briefly in Section 17.9 and in the Q&A section at the end of Chapter 17.) The C standard describes incomplete types as “types that describe objects but lack information needed to determine their sizes.” For example, the declaration

```
struct t; /* incomplete declaration of t */
```

tells the compiler that `t` is a structure tag but doesn't describe the members of the structure. As a result, the compiler doesn't have enough information to determine the size of such a structure. The intent is that an incomplete type will be completed elsewhere in the program.

As long as a type remains incomplete, its uses are limited. Since the compiler doesn't know the size of an incomplete type, it can't be used to declare a variable:

```
struct t s; /* WRONG */
```

However, it's perfectly legal to define a pointer type that references an incomplete type:

```
typedef struct t *T;
```

This type definition states that a variable of type `T` is a pointer to a structure with tag `t`. We can now declare variables of type `T`, pass them as arguments to functions, and perform other operations that are legal for pointers. (The size of a pointer doesn't depend on what it points to, which explains why C allows this behavior.) What we can't do, though, is apply the `->` operator to one of these variables, since the compiler knows nothing about the members of a `t` structure.

**Q&A**

**Q&A**

## 19.4 A Stack Abstract Data Type

To illustrate how abstract data types can be encapsulated using incomplete types, we'll develop a stack ADT based on the stack module described in Section 19.2. In the process, we'll explore three different ways to implement the stack.

### Defining the Interface for the Stack ADT

First, we'll need a header file that defines our stack ADT type and gives prototypes for the functions that represent stack operations. Let's name this file `stackADT.h`. The `Stack` type will be a pointer to a `stack_type` structure that stores the actual contents of the stack. This structure is an incomplete type that will be completed in the file that implements the stack. The members of this structure will depend on how the stack is implemented. Here's what the `stackADT.h` file will look like:

```
stackADT.h (version 1) #ifndef STACKADT_H
#define STACKADT_H

#include <stdbool.h> /* C99 only */

typedef struct stack_type *Stack;

Stack create(void);
void destroy(Stack s);
void make_empty(Stack s);
bool is_empty(Stack s);
bool is_full(Stack s);
void push(Stack s, int i);
int pop(Stack s);

#endif
```

Clients that include `stackADT.h` will be able to declare variables of type `Stack`, each of which is capable of pointing to a `stack_type` structure. Clients can then call the functions declared in `stackADT.h` to perform operations on stack variables. However, clients can't access the members of the `stack_type` structure, since that structure will be defined in a separate file.

Note that each function has a `Stack` parameter or returns a `Stack` value. The stack functions in Section 19.3 had parameters of type `Stack *`. The reason for the difference is that a `Stack` variable is now a pointer; it points to a `stack_type` structure that stores the contents of the stack. If a function needs to modify the stack, it changes the structure itself, not the pointer to the structure.

Also note the presence of the `create` and `destroy` functions. A module

generally doesn't need these functions, but an ADT does. `create` will dynamically allocate memory for a stack (including the memory required for a `stack_type` structure), as well as initializing the stack to its "empty" state. `destroy` will release the stack's dynamically allocated memory.

The following client file can be used to test the stack ADT. It creates two stacks and performs a variety of operations on them.

```
stackclient.c #include <stdio.h>
#include "stackADT.h"

int main(void)
{
 Stack s1, s2;
 int n;

 s1 = create();
 s2 = create();

 push(s1, 1);
 push(s1, 2);

 n = pop(s1);
 printf("Popped %d from s1\n", n);
 push(s2, n);
 n = pop(s1);
 printf("Popped %d from s1\n", n);
 push(s2, n);

 destroy(s1);

 while (!is_empty(s2))
 printf("Popped %d from s2\n", pop(s2));

 push(s2, 3);
 make_empty(s2);
 if (is_empty(s2))
 printf("s2 is empty\n");
 else
 printf("s2 is not empty\n");

 destroy(s2);

 return 0;
}
```

If the stack ADT is implemented correctly, the program should produce the following output:

```
Popped 2 from s1
Popped 1 from s1
Popped 1 from s2
Popped 2 from s2
s2 is empty
```

## Implementing the Stack ADT Using a Fixed-Length Array

There are several ways to implement the stack ADT. Our first approach is the simplest. We'll have the `stackADT.c` file define the `stack_type` structure so that it contains a fixed-length array (to hold the contents of the stack) along with an integer that keeps track of the top of the stack:

```
struct stack_type {
 int contents[STACK_SIZE];
 int top;
};
```

Here's what `stackADT.c` will look like:

```
stackADT.c #include <stdio.h>
#include <stdlib.h>
#include "stackADT.h"

#define STACK_SIZE 100

struct stack_type {
 int contents[STACK_SIZE];
 int top;
};

static void terminate(const char *message)
{
 printf("%s\n", message);
 exit(EXIT_FAILURE);
}

Stack create(void)
{
 Stack s = malloc(sizeof(struct stack_type));
 if (s == NULL)
 terminate("Error in create: stack could not be created.");
 s->top = 0;
 return s;
}

void destroy(Stack s)
{
 free(s);
}

void make_empty(Stack s)
{
 s->top = 0;
}

bool is_empty(Stack s)
{
 return s->top == 0;
}
```

```

bool is_full(Stack s)
{
 return s->top == STACK_SIZE;
}

void push(Stack s, int i)
{
 if (is_full(s))
 terminate("Error in push: stack is full.");
 s->contents[s->top++] = i;
}

int pop(Stack s)
{
 if (is_empty(s))
 terminate("Error in pop: stack is empty.");
 return s->contents[--s->top];
}

```

The most striking thing about the functions in this file is that they use the `->` operator, not the `.` operator, to access the `contents` and `top` members of the `stack_type` structure. The `s` parameter is a pointer to a `stack_type` structure, not a structure itself, so using the `.` operator would be illegal.

## Changing the Item Type in the Stack ADT

Now that we have a working version of the stack ADT, let's try to improve it. First, note that items in the stack must be integers. That's too restrictive; in fact, the item type doesn't really matter. The stack items could just as easily be other basic types (`float`, `double`, `long`, etc.) or even structures, unions, or pointers, for that matter.

To make the stack ADT easier to modify for different item types, let's add a type definition to the `stackADT.h` header. It will define a type named `Item`, representing the type of data to be stored on the stack.

**stackADT.h  
(version 2)**

```

#ifndef STACKADT_H
#define STACKADT_H

#include <stdbool.h> /* C99 only */

typedef int Item;

typedef struct stack_type *Stack;

Stack create(void);
void destroy(Stack s);
void make_empty(Stack s);
bool is_empty(Stack s);
bool is_full(Stack s);

```

```

void push(Stack s, Item i);
Item pop(Stack s);

#endif

```

The changes to the file are shown in **bold**. Besides the addition of the `Item` type, the `push` and `pop` functions have been modified. `push` now has a parameter of type `Item`, and `pop` returns a value of type `Item`. We'll use this version of `stackADT.h` from now on; it replaces the earlier version.

The `stackADT.c` file will need to be modified to match the new `stackADT.h`. The changes are minimal, however. The `stack_type` structure will now contain an array whose elements have type `Item` instead of `int`:

```

struct stack_type {
 Item contents[STACK_SIZE];
 int top;
};

```

The only other changes are to `push` (the second parameter now has type `Item`) and `pop` (which returns a value of type `Item`). The bodies of `push` and `pop` are unchanged.

The `stackclient.c` file can be used to test the new `stackADT.h` and `stackADT.c` to verify that the `Stack` type still works (it does!). Now we can change the item type any time we want by simply modifying the definition of the `Item` type in `stackADT.h`. (Although we won't have to change the `stackADT.c` file, we'll still need to recompile it.)

## Implementing the Stack ADT Using a Dynamic Array

Another problem with the stack ADT as it currently stands is that each stack has a fixed maximum size, which is currently set at 100 items. This limit can be increased to any number we wish, of course, but all stacks created using the `Stack` type will have the same limit. There's no way to have stacks with different capacities or to set the stack size as the program is running.

There are two solutions to this problem. One is to implement the stack as a linked list, in which case there's no fixed limit on its size. We'll investigate this solution in a moment. First, though, let's try the other approach, which involves storing stack items in a dynamically allocated array.

The crux of the latter approach is to modify the `stack_type` structure so that the `contents` member is a *pointer* to the array in which the items are stored, not the array itself:

```

struct stack_type {
 Item *contents;
 int top;
 int size;
};

```

I've also added a new member, `size`, that stores the stack's maximum size (the length of the array that `contents` points to). We'll use this member to check for the "stack full" condition.

The `create` function will now have a parameter that specifies the desired maximum stack size:

```
Stack create(int size);
```

When `create` is called, it will create a `stack_type` structure plus an array of length `size`. The `contents` member of the structure will point to this array.

The `stackADT.h` file will be the same as before, except that we'll need to add a `size` parameter to the `create` function. (Let's name the new version `stackADT2.h`.) The `stackADT.c` file will need more extensive modification, however. The new version appears below, with changes shown in **bold**.

```
stackADT2.c #include <stdio.h>
#include <stdlib.h>
#include "stackADT2.h"

struct stack_type {
 Item *contents;
 int top;
 int size;
};

static void terminate(const char *message)
{
 printf("%s\n", message);
 exit(EXIT_FAILURE);
}

Stack create(int size)
{
 Stack s = malloc(sizeof(struct stack_type));
 if (s == NULL)
 terminate("Error in create: stack could not be created.");
 s->contents = malloc(size * sizeof(Item));
 if (s->contents == NULL) {
 free(s);
 terminate("Error in create: stack could not be created.");
 }
 s->top = 0;
 s->size = size;
 return s;
}

void destroy(Stack s)
{
 free(s->contents);
 free(s);
}
```

```

void make_empty(Stack s)
{
 s->top = 0;
}

bool is_empty(Stack s)
{
 return s->top == 0;
}

bool is_full(Stack s)
{
 return s->top == s->size;
}

void push(Stack s, Item i)
{
 if (is_full(s))
 terminate("Error in push: stack is full.");
 s->contents[s->top++] = i;
}

Item pop(Stack s)
{
 if (is_empty(s))
 terminate("Error in pop: stack is empty.");
 return s->contents[--s->top];
}

```

The `create` function now calls `malloc` twice: once to allocate a `stack_type` structure and once to allocate the array that will contain the stack items. Either call of `malloc` could fail, causing `terminate` to be called. The `destroy` function must call `free` twice to release all the memory allocated by `create`.

The `stackclient.c` file can again be used to test the stack ADT. The calls of `create` will need to be changed, however, since `create` now requires an argument. For example, we could replace the statements

```
s1 = create();
s2 = create();
```

with the following statements:

```
s1 = create(100);
s2 = create(200);
```

## Implementing the Stack ADT Using a Linked List

Implementing the stack ADT using a dynamically allocated array gives us more flexibility than using a fixed-size array. However, the client is still required to specify a maximum size for a stack at the time it's created. If we use a linked-list implementation instead, there won't be any preset limit on the size of a stack.

Our implementation will be similar to the one in the `stack2.c` file of Section 19.2. The linked list will consist of nodes, represented by the following structure:

```
struct node {
 Item data;
 struct node *next;
};
```

The type of the `data` member is now `Item` rather than `int`, but the structure is otherwise the same as before.

The `stack_type` structure will contain a pointer to the first node in the list:

```
struct stack_type {
 struct node *top;
};
```

At first glance, the `stack_type` structure seems superfluous; we could just define `Stack` to be `struct node *` and let a `Stack` value be a pointer to the first node in the list. However, we still need the `stack_type` structure so that the interface to the stack remains unchanged. (If we did away with it, any function that modified the stack would need a `Stack *` parameter instead of a `Stack` parameter.) Moreover, having the `stack_type` structure will make it easier to change the implementation in the future, should we decide to store additional information. For example, if we later decide that the `stack_type` structure should contain a count of how many items are currently stored in the stack, we can easily add a member to the `stack_type` structure to store this information.

We won't need to make any changes to the `stackADT.h` header. (We'll use this header file, not `stackADT2.h`.) We can also use the original `stack-client.c` file for testing. All the changes will be in the `stackADT.c` file. Here's the new version:

```
stackADT3.c #include <stdio.h>
#include <stdlib.h>
#include "stackADT.h"

struct node {
 Item data;
 struct node *next;
};

struct stack_type {
 struct node *top;
};

static void terminate(const char *message)
{
 printf("%s\n", message);
 exit(EXIT_FAILURE);
}
```

```
Stack create(void)
{
 Stack s = malloc(sizeof(struct stack_type));
 if (s == NULL)
 terminate("Error in create: stack could not be created.");
 s->top = NULL;
 return s;
}

void destroy(Stack s)
{
 make_empty(s);
 free(s);
}

void make_empty(Stack s)
{
 while (!is_empty(s))
 pop(s);
}

bool is_empty(Stack s)
{
 return s->top == NULL;
}

bool is_full(Stack s)
{
 return false;
}

void push(Stack s, Item i)
{
 struct node *new_node = malloc(sizeof(struct node));
 if (new_node == NULL)
 terminate("Error in push: stack is full.");
 new_node->data = i;
 new_node->next = s->top;
 s->top = new_node;
}

Item pop(Stack s)
{
 struct node *old_top;
 Item i;

 if (is_empty(s))
 terminate("Error in pop: stack is empty.");

 old_top = s->top;
 i = old_top->data;
 s->top = old_top->next;
 free(old_top);
 return i;
}
```

Note that the `destroy` function calls `make_empty` (to release the memory occupied by the nodes in the linked list) before it calls `free` (to release the memory for the `stack_type` structure).

## 19.5 Design Issues for Abstract Data Types

Section 19.4 described a stack ADT and showed several ways to implement it. Unfortunately, this ADT suffers from several problems that prevent it from being industrial-strength. Let's look at each of these problems and discuss possible solutions.

### Naming Conventions

The stack ADT functions currently have short, easy-to-understand names: `create`, `destroy`, `make_empty`, `is_empty`, `is_full`, `push`, and `pop`. If we have more than one ADT in a program, name clashes are likely, with functions in two modules having the same name. (Each ADT will need its own `create` function, for example.) Therefore, we'll probably need to use function names that incorporate the name of the ADT itself, such as `stack_create` instead of `create`.

### Error Handling

The stack ADT deals with errors by displaying an error message and terminating the program. That's not a bad thing to do. The programmer can avoid popping an empty stack or pushing data onto a full stack by being careful to call `is_empty` prior to each call of `pop` and `is_full` prior to each call of `push`, so in theory there's no reason for a call of `push` or `pop` to fail. (In the linked-list implementation, however, calling `is_full` isn't foolproof; a subsequent call of `push` can still fail.) Nevertheless, we might want to provide a way for a program to recover from these errors rather than terminating.

An alternative is to have the `push` and `pop` functions return a `bool` value to indicate whether or not they succeeded. `push` currently has a `void` return type, so it would be easy to modify it to return `true` if the `push` operation succeeds and `false` if the stack is full. Modifying the `pop` function would be more difficult, since `pop` currently returns the value that was popped. However, if `pop` were to return a *pointer* to this value, instead of the value itself, then `pop` could return `NULL` to indicate that the stack is empty.

A final comment about error handling: The C standard library contains a parameterized macro named `assert` that can terminate a program if a specified condition isn't satisfied. We could use calls of this macro as replacements for the `if` statements and calls of `terminate` that currently appear in the stack ADT.

## Generic ADTs

Midway through Section 19.4, we improved the stack ADT by making it easier to change the type of items stored in a stack—all we had to do was modify the definition of the `Item` type. It's still somewhat of a nuisance to do so; it would be nicer if a stack could accommodate items of any type, without the need to modify the `stack.h` file. Also note that our stack ADT suffers from a serious flaw: a program can't create two stacks whose items have different types. It's easy to create multiple stacks, but those stacks must have items with identical types. To allow stacks with different item types, we'd have to make copies of the stack ADT's header file and source file and modify one set of files so that the `Stack` type and its associated functions have different names.

What we'd like to have is a single "generic" stack type from which we could create a stack of integers, a stack of strings, or any other stack that we might need. There are various ways to create such a type in C, but none are completely satisfactory. The most common approach uses `void *` as the item type, which allows arbitrary pointers to be pushed and popped. With this technique, the `stack-ADT.h` file would be similar to our original version; however, the prototypes of the push and pop functions would have the following appearance:

```
void push(Stack s, void *p);
void *pop(Stack s);
```

`pop` returns a pointer to the item popped from the stack; if the stack is empty, it returns a null pointer.

There are two disadvantages to using `void *` as the item type. One is that this approach doesn't work for data that can't be represented in pointer form. Items could be strings (which are represented by a pointer to the first character in the string) or dynamically allocated structures but not basic types such as `int` and `double`. The other disadvantage is that error checking is no longer possible. A stack that stores `void *` items will happily allow a mixture of pointers of different types; there's no way to detect an error caused by pushing a pointer of the wrong type.

## ADTs in Newer Languages

The problems that we've just discussed are dealt with much more cleanly in newer C-based languages, such as C++, Java, and C#. Name clashes are prevented by defining function names within a *class*. A stack ADT would be represented by a `Stack` class; the stack functions would belong to this class, and would only be recognized by the compiler when applied to a `Stack` object. These languages have a feature known as *exception handling* that allows functions such as `push` and `pop` to "throw" an exception when they detect an error condition. Code in the client can then deal with the error by "catching" the exception. C++, Java, and C# also provide special features for defining generic ADTs. In C++, for example, we would define a stack *template*, leaving the item type unspecified.

## Q & A

**Q:** You said that C wasn't designed for writing large programs. Isn't UNIX a large program? [p. 483]

**A:** Not at the time C was designed. In a 1978 paper, Ken Thompson estimated that the UNIX kernel was about 10,000 lines of C code (plus a small amount of assembler). Other components of UNIX were of comparable size; in another 1978 paper, Dennis Ritchie and colleagues put the size of the PDP-11 C compiler at 9660 lines. By today's standards, these are indeed small programs.

**Q:** Are there any abstract data types in the C library?

**A:** Technically there aren't, but a few come close, including the FILE type (defined in `<stdio.h>`). Before performing an operation on a file, we must declare a variable of type FILE \*:

```
FILE *fp;
```

The fp variable will then be passed to various file-handling functions.

Programmers are expected to treat FILE as an abstraction. It's not necessary to know what a FILE is in order to use the FILE type. Presumably FILE is a structure type, but the C standard doesn't even guarantee that. In fact, it's better not to know too much about how FILE values are stored, since the definition of the FILE type can (and often does) vary from one C compiler to another.

Of course, we can always look in the `stdio.h` file and see what a FILE is. Having done so, there's nothing to prevent us from writing code to access the internals of a FILE. For example, we might discover that FILE is a structure with a member named `bsize` (the file's buffer size):

```
typedef struct {
 ...
 int bsize; /* buffer size */
 ...
} FILE;
```

Once we know about the `bsize` member, there's nothing to prevent us from accessing the buffer size for a particular file:

```
printf("Buffer size: %d\n", fp->bsize);
```

Doing so isn't a good idea, however, because other C compilers might store the buffer size under a different name, or keep track of it in some entirely different way. Changing the `bsize` member is an even worse idea:

```
fp->bsize = 1024;
```

Unless we know all the details about how files are stored, this is a dangerous thing to do. Even if we *do* know the details, they may change with a different compiler or the next release of the same compiler.

FILE type ► 22.1

**Q:** What other incomplete types are there besides incomplete structure types? [p. 492]

**A:** One of the most common incomplete types occurs when an array is declared with no specified size:

```
extern int a[];
```

After this declaration (which we first encountered in Section 15.2), `a` has an incomplete type, because the compiler doesn't know `a`'s length. Presumably `a` is defined in another file within the program; that definition will supply the missing length. Another incomplete type occurs in declarations that specify no length for an array but provide an initializer:

```
int a[] = {1, 2, 3};
```

In this example, the array `a` initially has an incomplete type, but the type is completed by the initializer.

**C99**

flexible array members ▶ 17.9

Declaring a union tag without specifying the members of the union also creates an incomplete type. Flexible array members (a C99 feature) have an incomplete type. Finally, `void` is an incomplete type. The `void` type has the unusual property that it can never be completed, thus making it impossible to declare a variable of this type.

**Q:** What other restrictions are there on the use of incomplete types? [p. 492]

**A:** The `sizeof` operator can't be applied to an incomplete type (not surprisingly, since the size of an incomplete type is unknown). A member of a structure or union (other than a flexible array member) can't have an incomplete type. Similarly, the elements of an array can't have an incomplete type. Finally, a parameter in a function definition can't have an incomplete type (although this is allowed in a function *declaration*). The compiler "adjusts" each array parameter in a function definition so that it has a pointer type, thus preventing it from having an incomplete type.

## Exercises

### Section 19.1

1. A *queue* is similar to a stack, except that items are added at one end but removed from the other in a **FIFO** (first-in, first-out) fashion. Operations on a queue might include:

Inserting an item at the end of the queue

Removing an item from the beginning of the queue

Returning the first item in the queue (without changing the queue)

Returning the last item in the queue (without changing the queue)

Testing whether the queue is empty

Write an interface for a queue module in the form of a header file named `queue.h`.

### Section 19.2

2. Modify the `stack2.c` file to use the `PUBLIC` and `PRIVATE` macros.

3. (a) Write an array-based implementation of the queue module described in Exercise 1. Use three integers to keep track of the queue's status, with one integer storing the position of the first empty slot in the array (used when an item is inserted), the second storing the position of the next item to be removed, and the third storing the number of items in the queue. An insertion or removal that would cause either of the first two integers to be incremented past the end of the array should instead reset the variable to zero, thus causing it to "wrap around" to the beginning of the array.  
(b) Write a linked-list implementation of the queue module described in Exercise 1. Use two pointers, one pointing to the first node in the list and the other pointing to the last node. When an item is inserted into the queue, add it to the end of the list. When an item is removed from the queue, delete the first node in the list.
- Section 19.3**    **W** 4. (a) Write an implementation of the `Stack` type, assuming that `Stack` is a structure containing a fixed-length array.  
(b) Redo the `Stack` type, this time using a linked-list representation instead of an array. (Show both `stack.h` and `stack.c`.)  
5. Modify the `queue.h` header of Exercise 1 so that it defines a `Queue` type, where `Queue` is a structure containing a fixed-length array (see Exercise 3(a)). Modify the functions in `queue.h` to take a `Queue *` parameter.
- Section 19.4**    6. (a) Add a `peek` function to `stackADT.c`. This function will have a parameter of type `Stack`. When called, it returns the top item on the stack but doesn't modify the stack.  
(b) Repeat part (a), modifying `stackADT2.c` this time.  
(c) Repeat part (a), modifying `stackADT3.c` this time.  
7. Modify `stackADT2.c` so that a stack automatically doubles in size when it becomes full. Have the `push` function dynamically allocate a new array that's twice as large as the old one and then copy the stack contents from the old array to the new one. Be sure to have `push` deallocate the old array once the data has been copied.

---

## Programming Projects

1. Modify Programming Project 1 from Chapter 10 so that it uses the stack ADT described in Section 19.4. You may use any of the implementations of the ADT described in that section.
2. Modify Programming Project 6 from Chapter 10 so that it uses the stack ADT described in Section 19.4. You may use any of the implementations of the ADT described in that section.
3. Modify the `stackADT3.c` file of Section 19.4 by adding an `int` member named `len` to the `stack_type` structure. This member will keep track of how many items are currently stored in a stack. Add a new function named `length` that has a `Stack` parameter and returns the value of the `len` member. (Some of the existing functions in `stackADT3.c` will need to be modified as well.) Modify `stackclient.c` so that it calls the `length` function (and displays the value that it returns) after each operation that modifies a stack.
4. Modify the `stackADT.h` and `stackADT3.c` files of Section 19.4 so that a stack stores values of type `void *`, as described in Section 19.5; the `Item` type will no longer be used. Modify `stackclient.c` so that it stores pointers to strings in the `s1` and `s2` stacks.

5. Starting from the `queue.h` header of Exercise 1, create a file named `queueADT.h` that defines the following `Queue` type:

```
typedef struct queue_type *Queue;
```

`queue_type` is an incomplete structure type. Create a file named `queueADT.c` that contains the full definition of `queue_type` as well as definitions for all the functions in `queue.h`. Use a fixed-length array to store the items in a queue (see Exercise 3(a)). Create a file named `queueclient.c` (similar to the `stackclient.c` file of Section 19.4) that creates two queues and performs operations on them. Be sure to provide `create` and `destroy` functions for your ADT.

6. Modify Programming Project 5 so that the items in a queue are stored in a dynamically allocated array whose length is passed to the `create` function.
7. Modify Programming Project 5 so that the items in a queue are stored in a linked list (see Exercise 3(b)).



# 20 Low-Level Programming

*A programming language is low level when its programs require attention to the irrelevant.*

Previous chapters have described C’s high-level, machine-independent features. Although these features are adequate for many applications, some programs need to perform operations at the bit level. Bit manipulation and other low-level operations are especially useful for writing systems programs (including compilers and operating systems), encryption programs, graphics programs, and programs for which fast execution and/or efficient use of space is critical.

Section 20.1 covers C’s bitwise operators, which provide easy access to both individual bits and bit-fields. Section 20.2 then shows how to declare structures that contain bit-fields. Finally, Section 20.3 describes how certain ordinary C features (type definitions, unions, and pointers) can help in writing low-level programs.

Some of the techniques described in this chapter depend on knowledge of how data is stored in memory, which can vary depending on the machine and the compiler. Relying on these techniques will most likely make a program nonportable, so it’s best to avoid them unless absolutely necessary. If you do need them, try to limit their use to certain modules in your program; don’t spread them around. And, above all, be sure to document what you’re doing!

## 20.1 Bitwise Operators

C provides six *bitwise operators*, which operate on integer data at the bit level. We’ll discuss the two bitwise shift operators first, followed by the four other bitwise operators (bitwise complement, bitwise *and*, bitwise exclusive *or*, and bitwise inclusive *or*).

## Bitwise Shift Operators

The bitwise shift operators can transform the binary representation of an integer by shifting its bits to the left or right. C provides two shift operators, which are shown in Table 20.1.

**Table 20.1**  
Bitwise Shift Operators

| Symbol | Meaning     |
|--------|-------------|
| <<     | left shift  |
| >>     | right shift |

The operands for << and >> may be of any integer type (including `char`). The integer promotions are performed on both operands; the result has the type of the left operand after promotion.

The value of `i << j` is the result when the bits in `i` are shifted left by `j` places. For each bit that is “shifted off” the left end of `i`, a zero bit enters at the right. The value of `i >> j` is the result when `i` is shifted right by `j` places. If `i` is of an unsigned type or if the value of `i` is nonnegative, zeros are added at the left as needed. If `i` is a negative number, the result is implementation-defined; some implementations add zeros at the left end, while others preserve the sign bit by adding ones.

**portability tip**

*For portability, it's best to perform shifts only on unsigned numbers.*

The following examples illustrate the effect of applying the shift operators to the number 13. (For simplicity, these examples—and others in this section—use short integers, which are typically 16 bits.)

```
unsigned short i, j;

i = 13; /* i is now 13 (binary 000000000001101) */
j = i << 2; /* j is now 52 (binary 0000000000110100) */
j = i >> 2; /* j is now 3 (binary 0000000000000011) */
```

As these examples show, neither operator modifies its operands. To modify a variable by shifting its bits, we'd use the compound assignment operators <<= and >>=:

```
i = 13; /* i is now 13 (binary 000000000001101) */
i <<= 2; /* i is now 52 (binary 0000000000110100) */
i >>= 2; /* i is now 13 (binary 000000000001101) */
```




---

The bitwise shift operators have lower precedence than the arithmetic operators, which can cause surprises. For example, `i << 2 + 1` means `i << (2 + 1)`, not `(i << 2) + 1`.

## Bitwise Complement, And, Exclusive Or, and Inclusive Or

Table 20.2 lists the remaining bitwise operators.

**Table 20.2**

Other Bitwise Operators

| Symbol             | Meaning                     |
|--------------------|-----------------------------|
| <code>~</code>     | bitwise complement          |
| <code>&amp;</code> | bitwise <i>and</i>          |
| <code>^</code>     | bitwise exclusive <i>or</i> |
| <code> </code>     | bitwise inclusive <i>or</i> |

The `~` operator is unary; the integer promotions are performed on its operand. The other operators are binary; the usual arithmetic conversions are performed on their operands.

The `~, &, ^, and |` operators perform Boolean operations on all bits in their operands. The `~` operator produces the complement of its operand, with zeros replaced by ones and ones replaced by zeros. The `&` operator performs a Boolean *and* operation on all corresponding bits in its two operands. The `^` and `|` operators are similar (both perform a Boolean *or* operation on the bits in their operands); however, `^` produces 0 whenever both operands have a 1 bit, whereas `|` produces 1.



Don't confuse the *bitwise* operators `&` and `|` with the *logical* operators `&&` and `||`. The bitwise operators sometimes produce the same results as the logical operators, but they're not equivalent.

The following examples illustrate the effect of the `~, &, ^, and |` operators:

```
unsigned short i, j, k;

i = 21; /* i is now 21 (binary 0000000000010101) */
j = 56; /* j is now 56 (binary 00000000000111000) */
k = ~i; /* k is now 65514 (binary 111111111101010) */
k = i & j; /* k is now 16 (binary 0000000000010000) */
k = i ^ j; /* k is now 45 (binary 0000000000101101) */
k = i | j; /* k is now 61 (binary 0000000000111101) */
```

The value shown for `~i` is based on the assumption that an `unsigned short` value occupies 16 bits.

The `~` operator deserves special mention, since we can use it to help make even low-level programs more portable. Suppose that we need an integer whose bits are all 1. The preferred technique is to write `~0`, which doesn't depend on the number of bits in an integer. Similarly, if we need an integer whose bits are all 1 except for the last five, we could write `~0x1f`.

Each of the `~`, `&`, `^`, and `|` operators has a different precedence:

Highest:     `~`  
                 `&`  
                 `^`

Lowest:     `|`

As a result, we can combine these operators in expressions without having to use parentheses. For example, we could write `i & ~j | k` instead of `(i & (~j)) | k` and `i ^ j & ~k` instead of `i ^ (j & (~k))`. Of course, it doesn't hurt to use parentheses to avoid confusion.



table of operators ► Appendix A

The precedence of `&`, `^`, and `|` is lower than the precedence of the relational and equality operators. Consequently, statements like the following one won't have the desired effect:

```
if (status & 0x4000 != 0) ...
```

Instead of testing whether `status & 0x4000` isn't zero, this statement will evaluate `0x4000 != 0` (which has the value 1), then test whether the value of `status & 1` isn't zero.

The compound assignment operators `&=`, `^=`, and `|=` correspond to the bitwise operators `&`, `^`, and `|`:

```
i = 21; /* i is now 21 (binary 000000000010101) */

j = 56; /* j is now 56 (binary 0000000000111000) */

i &= j; /* i is now 16 (binary 000000000010000) */

i ^= j; /* i is now 40 (binary 0000000000101000) */

i |= j; /* i is now 56 (binary 0000000000111000) */
```

## Using the Bitwise Operators to Access Bits

When we do low-level programming, we'll often need to store information as single bits or collections of bits. In graphics programming, for example, we may want to squeeze two or more pixels into a single byte. Using the bitwise operators, we can extract or modify data that's stored in a small number of bits.

Let's assume that `i` is a 16-bit unsigned short variable. Let's see how to perform the most common single-bit operations on `i`:

- **Setting a bit.** Suppose that we want to set bit 4 of `i`. (We'll assume that the leftmost—or **most significant**—bit is numbered 15 and the least significant is numbered 0.) The easiest way to set bit 4 is to *or* the value of `i` with the constant `0x0010` (a “mask” that contains a 1 bit in position 4):

```
i = 0x0000; /* i is now 0000000000000000 */

i |= 0x0010; /* i is now 0000000000001000 */
```

More generally, if the position of the bit is stored in the variable `j`, we can use a shift operator to create the mask:

**idiom**      `i |= 1 << j; /* sets bit j */`

For example, if `j` has the value 3, then `1 << j` is `0x0008`.

- **Clearing a bit.** To clear bit 4 of `i`, we'd use a mask with a 0 bit in position 4 and 1 bits everywhere else:

```
i = 0x00ff; /* i is now 0000000011111111 */
i &= ~0x0010; /* i is now 0000000011101111 */
```

Using the same idea, we can easily write a statement that clears a bit whose position is stored in a variable:

**idiom**      `i &= ~(1 << j); /* clears bit j */`

- **Testing a bit.** The following `if` statement tests whether bit 4 of `i` is set:

```
if (i & 0x0010) ... /* tests bit 4 */
```

To test whether bit `j` is set, we'd use the following statement:

**idiom**      `if (i & 1 << j) ... /* tests bit j */`

To make working with bits easier, we'll often give them names. For example, suppose that we want bits 0, 1, and 2 of a number to correspond to the colors blue, green, and red, respectively. First, we define names that represent the three bit positions:

```
#define BLUE 1
#define GREEN 2
#define RED 4
```

Setting, clearing, and testing the `BLUE` bit would be done as follows:

```
i |= BLUE; /* sets BLUE bit */
i &= ~BLUE; /* clears BLUE bit */
if (i & BLUE) ... /* tests BLUE bit */
```

It's also easy to set, clear, or test several bits at time:

```
i |= BLUE | GREEN; /* sets BLUE and GREEN bits */
i &= ~ (BLUE | GREEN); /* clears BLUE and GREEN bits */
if (i & (BLUE | GREEN)) ... /* tests BLUE and GREEN bits */
```

The `if` statement tests whether either the `BLUE` bit *or* the `GREEN` bit is set.

## Using the Bitwise Operators to Access Bit-Fields

Dealing with a group of several consecutive bits (a *bit-field*) is slightly more complicated than working with single bits. Here are examples of the two most common bit-field operations:

- **Modifying a bit-field.** Modifying a bit-field requires a bitwise *and* (to clear the bit-field), followed by a bitwise *or* (to store new bits in the bit-field). The following statement shows how we might store the binary value 101 in bits 4–6 of the variable `i`:

```
i = i & ~0x0070 | 0x0050; /* stores 101 in bits 4-6 */
```

The `&` operator clears bits 4–6 of `i`; the `|` operator then sets bits 6 and 4. Notice that `i |= 0x0050` by itself wouldn't always work: it would set bits 6 and 4 but not change bit 5. To generalize the example a little, let's assume that the variable `j` contains the value to be stored in bits 4–6 of `i`. We'll need to shift `j` into position before performing the bitwise `or`:

```
i = (i & ~0x0070) | (j << 4); /* stores j in bits 4-6 */
```

The `|` operator has lower precedence than `&` and `<<`, so we can drop the parentheses if we wish:

```
i = i & ~0x0070 | j << 4;
```

- **Retrieving a bit-field.** When the bit-field is at the right end of a number (in the least significant bits), fetching its value is easy. For example, the following statement retrieves bits 0–2 in the variable `i`:

```
j = i & 0x0007; /* retrieves bits 0-2 */
```

If the bit-field isn't at the right end of `i`, then we can first shift the bit-field to the end before extracting the field using the `&` operator. To extract bits 4–6 of `i`, for example, we could use the following statement:

```
j = (i >> 4) & 0x0007; /* retrieves bits 4-6 */
```

## PROGRAM XOR Encryption

One of the simplest ways to encrypt data is to exclusive-*or* (XOR) each character with a secret key. Suppose that the key is the `&` character. If we XOR this key with the character `z`, we'll get the `\` character (assuming that we're using the ASCII character set):

|                     |                                      |
|---------------------|--------------------------------------|
| 00100110            | (ASCII code for <code>&amp;</code> ) |
| XOR <u>01111010</u> | (ASCII code for <code>z</code> )     |
| 01011100            | (ASCII code for <code>\</code> )     |

To decrypt a message, we just apply the same algorithm. In other words, by encrypting an already-encrypted message, we'll recover the original message. If we XOR the `&` character with the `\` character, for example, we'll get the original character, `z`:

|                     |                                      |
|---------------------|--------------------------------------|
| 00100110            | (ASCII code for <code>&amp;</code> ) |
| XOR <u>01011100</u> | (ASCII code for <code>\</code> )     |
| 01111010            | (ASCII code for <code>z</code> )     |

The following program, `xor.c`, encrypts a message by XORing each character with the `&` character. The original message can be entered by the user or read from a file using input redirection; the encrypted message can be viewed on the screen or saved in a file using output redirection. For example, suppose that the file

`msg` contains the following lines:

Trust not him with your secrets, who, when left alone in your room, turns over your papers.

--Johann Kaspar Lavater (1741-1801)

To encrypt the `msg` file, saving the encrypted message in `newmsg`, we'd use the following command:

```
xor <msg> newmsg
```

`newmsg` will now contain these lines:

```
rTSUR HIR NOK QORN _IST UCETCRU, QNI, QNCH JC@R
GJIHC OH _IST TIIK, RSTHU IPCT _IST VGVCTU.
--LINGHH mGUVGT jGPGRCT (1741-1801)
```

To recover the original message, we'd use the command

```
xor <newmsg>
```

which will display it on the screen.

As the example shows, our program won't change some characters, including digits. XORing these characters with `&` would produce invisible control characters, which could cause problems with some operating systems. In Chapter 22, we'll see how to avoid problems when reading and writing files that contain control characters. Until then, we'll play it safe by using the `isprint` function to make sure that both the original character and the new (encrypted) character are printing characters (i.e., not control characters). If either character fails this test, we'll have the program write the original character instead of the new character.

Here's the finished program, which is remarkably short:

```
xor.c /* Performs XOR encryption */

#include <ctype.h>
#include <stdio.h>

#define KEY '&'

int main(void)
{
 int orig_char, new_char;

 while ((orig_char = getchar()) != EOF) {
 new_char = orig_char ^ KEY;
 if (isprint(orig_char) && isprint(new_char))
 putchar(new_char);
 else
 putchar(orig_char);
 }

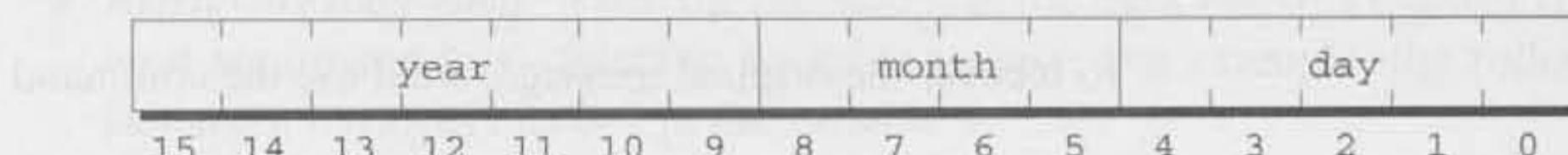
 return 0;
}
```

isprint function ▶ 23.5

## 20.2 Bit-Fields in Structures

Although the techniques of Section 20.1 allow us to work with bit-fields, these techniques can be tricky to use and potentially confusing. Fortunately, C provides an alternative: declaring structures whose members represent bit-fields.

As an example, let's look at how the MS-DOS operating system (often just called DOS) stores the date at which a file was created or last modified. Since days, months, and years are small numbers, storing them as normal integers would waste space. Instead, DOS allocates only 16 bits for a date, with 5 bits for the day, 4 bits for the month, and 7 bits for the year:



Using bit-fields, we can define a C structure with an identical layout:

```
struct file_date {
 unsigned int day: 5;
 unsigned int month: 4;
 unsigned int year: 7;
};
```

The number after each member indicates its length in bits. Since the members all have the same type, we can condense the declaration if we want:

```
struct file_date {
 unsigned int day: 5, month: 4, year: 7;
};
```

The type of a bit-field must be either int, unsigned int, or signed int. Using int is ambiguous; some compilers treat the field's high-order bit as a sign bit, but others don't.

#### **portability tip**

*Declare all bit-fields to be either unsigned int or signed int.*

In C99, bit-fields may also have type `_Bool`. C99 compilers may allow additional bit-field types.

We can use a bit-field just like any other member of a structure, as the following example shows:

```
struct file_date fd;

fd.day = 28;
fd.month = 12;
fd.year = 88; /* represents 1988 */
```

Note that the year member is stored relative to 1980 (the year the world began).

according to Microsoft). After these assignments, the `fd` variable will have the following appearance:

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 0  | 0  | 0  | 1  | 0  | 0  | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

We could have used the bitwise operators to accomplish the same effect; using these operators might even make the program a little faster. However, having a readable program is usually more important than gaining a few microseconds.

Bit-fields do have one restriction that doesn't apply to other members of a structure. Since bit-fields don't have addresses in the usual sense, C doesn't allow us to apply the address operator (`&`) to a bit-field. Because of this rule, functions such as `scanf` can't store data directly in a bit-field:

```
scanf ("%d", &fd.day); /* *** WRONG ***/
```

Of course, we can always use `scanf` to read input into an ordinary variable and then assign it to `fd.day`.

## How Bit-Fields Are Stored

Let's take a close look at how a compiler processes the declaration of a structure that has bit-field members. As we'll see, the C standard allows the compiler considerable latitude in choosing how it stores bit-fields.

The rules concerning how the compiler handles bit-fields depend on the notion of "storage units." The size of a storage unit is implementation-defined; typical values are 8 bits, 16 bits, and 32 bits. As it processes a structure declaration, the compiler packs bit-fields one by one into a storage unit, with no gaps between the fields, until there's not enough room for the next field. At that point, some compilers skip to the beginning of the next storage unit, while others split the bit-field across the storage units. (Which one occurs is implementation-defined.) The order in which bit-fields are allocated (left to right or right to left) is also implementation-defined.

Our `file_date` example assumes that storage units are 16 bits long. (An 8-bit storage unit would also be acceptable, provided that the compiler splits the `month` field across two storage units.) We also assume that bit-fields are allocated from right to left (with the first bit-field occupying the low-order bits).

C allows us to omit the name of any bit-field. Unnamed bit-fields are useful as "padding" to ensure that other bit fields are properly positioned. Consider the time associated with a DOS file, which is stored in the following way:

```
struct file_time {
 unsigned int seconds: 5;
 unsigned int minutes: 6;
 unsigned int hours: 5;
};
```

(You may be wondering how it's possible to store the seconds—a number between 0 and 59—in a field with only 5 bits. Well, DOS cheats: it divides the number of seconds by 2, so the `seconds` member is actually between 0 and 29.) If we're not interested in the `seconds` field, we can leave out its name:

```
struct file_time {
 unsigned int : 5; /* not used */
 unsigned int minutes: 6;
 unsigned int hours: 5;
};
```

The remaining bit-fields will be aligned as if the `seconds` field were still present.

Another trick that we can use to control the storage of bit-fields is to specify 0 as the length of an unnamed bit-field:

```
struct s {
 unsigned int a: 4;
 unsigned int : 0; /* 0-length bit-field */
 unsigned int b: 8;
};
```

A 0-length bit-field is a signal to the compiler to align the following bit-field at the beginning of a storage unit. If storage units are 8 bits long, the compiler will allocate 4 bits for the `a` member, skip 4 bits to the next storage unit, and then allocate 8 bits for `b`. If storage units are 16 bits long, the compiler will allocate 4 bits for `a`, skip 12 bits, and then allocate 8 bits for `b`.

## 20.3 Other Low-Level Techniques

Some of the language features that we've covered in previous chapters are used often in low-level programming. To wrap up this chapter, we'll take a look at several important examples: defining types that represent units of storage, using unions to bypass normal type-checking, and using pointers as addresses. We'll also cover the `volatile` type qualifier, which we avoided discussing in Section 18.3 because of its low-level nature.

### Defining Machine-Dependent Types

Since the `char` type—by definition—occupies one byte, we'll sometimes treat characters as bytes, using them to store data that's not necessarily in character form. When we do so, it's a good idea to define a `BYTE` type:

```
typedef unsigned char BYTE;
```

Depending on the machine, we may want to define additional types. The x86 architecture makes extensive use of 16-bit words, so the following definition would be useful for that platform:

```
typedef unsigned short WORD;
```

We'll use the BYTE and WORD types in later examples.

## Using Unions to Provide Multiple Views of Data

Although unions can be used in a portable way—see Section 16.4 for examples—they're often used in C for an entirely different purpose: viewing a block of memory in two or more different ways.

Here's a simple example based on the `file_date` structure described in Section 20.2. Since a `file_date` structure fits into two bytes, we can think of any two-byte value as a `file_date` structure. In particular, we could view an unsigned short value as a `file_date` structure (assuming that short integers are 16 bits long). The following union allows us to easily convert a short integer to a file date or vice versa:

```
union int_date {
 unsigned short i;
 struct file_date fd;
};
```

With the help of this union, we could fetch a file date from disk as two bytes, then extract its month, day, and year fields. Conversely, we could construct a date as a `file_date` structure, then write it to disk as a pair of bytes.

As an example of how we might use the `int_date` union, here's a function that, when passed an unsigned short argument, prints it as a file date:

```
void print_date(unsigned short n)
{
 union int_date u;

 u.i = n;
 printf("%d/%d/%d\n", u.fd.month, u.fd.day, u.fd.year + 1980);
}
```

Using unions to allow multiple views of data is especially useful when working with registers, which are often divided into smaller units. x86 processors, for example, have 16-bit registers named AX, BX, CX, and DX. Each of these registers can be treated as two 8-bit registers. AX, for example, is divided into registers named AH and AL. (The H and L stand for “high” and “low.”)

When writing low-level applications for x86-based computers, we may need variables that represent the contents of the AX, BX, CX, and DX registers. We want access to both the 16- and 8-bit registers; at the same time, we need to take their relationships into account (a change to AX affects both AH and AL; changing AH or AL modifies AX). The solution is to set up two structures, one containing members that correspond to the 16-bit registers, and the other containing members that match the 8-bit registers. We then create a union that encloses the two structures:

```

union {
 struct {
 WORD ax, bx, cx, dx;
 } word;
 struct {
 BYTE al, ah, bl, bh, cl, ch, dl, dh;
 } byte;
} regs;

```

The members of the `word` structure will be overlaid with the members of the `byte` structure; for example, `ax` will occupy the same memory as `al` and `ah`. And that, of course, is exactly what we wanted. Here's an example showing how the `regs` union might be used:

```

regs.byte.ah = 0x12;
regs.byte.al = 0x34;
printf("AX: %hx\n", regs.word.ax);

```

Changing `ah` and `al` affects `ax`, so the output will be

`AX: 1234`

Note that the `byte` structure lists `al` before `ah`, even though the `AL` register is the "low" half of `AX` and `AH` is the "high" half. Here's the reason. When a data item consists of more than one byte, there are two logical ways to store it in memory: with the bytes in the "natural" order (with the leftmost byte stored first) or with the bytes in reverse order (the leftmost byte is stored last). The first alternative is called **big-endian**; the second is known as **little-endian**. C doesn't require a specific byte ordering, since that depends on the CPU on which a program will be executed. Some CPUs use the big-endian approach and some use the little-endian approach. What does this have to do with the `byte` structure? It turns out that x86 processors assume that data is stored in little-endian order, so the first byte of `regs.word.ax` is the low byte.

We don't normally need to worry about byte ordering. However, programs that deal with memory at a low level must be aware of the order in which bytes are stored (as the `regs` example illustrates). It's also relevant when working with files that contain non-character data.



Be careful when using unions to provide multiple views of data. Data that is valid in its original format may be invalid when viewed as a different type, causing unexpected problems.

## Using Pointers as Addresses

We saw in Section 11.1 that a pointer is really some kind of memory address, although we usually don't need to know the details. When we do low-level programming, however, the details matter.

An address often has the same number of bits as an integer (or long integer). Creating a pointer that represents a specific address is easy: we just cast an integer into a pointer. For example, here's how we might store the address 1000 (hex) in a pointer variable:

```
BYTE *p;
p = (BYTE *) 0x1000; /* p contains address 0x1000 */
```

## PROGRAM Viewing Memory Locations

Our next program allows the user to view segments of computer memory; it relies on C's willingness to allow an integer to be used as a pointer. Most CPUs execute programs in "protected mode," however, which means that a program can access only those portions of memory that belong to the program. This prevents a program from accessing (or changing) memory that belongs to another application or to the operating system itself. As a result, we'll only be able to use our program to view areas of memory that have been allocated for use by the program itself. Going outside these regions will cause the program to crash.

The `viewmemory.c` program begins by displaying the address of its own `main` function as well as the address of one of its variables. This will give the user a clue as to which areas of memory can be probed. The program next prompts the user to enter an address (in the form of a hexadecimal integer) plus the number of bytes to view. The program then displays a block of bytes of the chosen length, starting at the specified address.

Bytes are displayed in groups of 10 (except for the last group, which may have fewer than 10 bytes). The address of a group of bytes is displayed at the beginning of a line, followed by the bytes in the group (displayed as hexadecimal numbers); followed by the same bytes displayed as characters (just in case the bytes happen to represent characters, as some of them may). Only printing characters (as determined by the `isprint` function) will be displayed; other characters will be shown as periods.

We'll assume that `int` values are stored using 32 bits and that addresses are also 32 bits long. Addresses are displayed in hexadecimal, as is customary.

```
viewmemory.c /* Allows the user to view regions of computer memory */

#include <ctype.h>
#include <stdio.h>

typedef unsigned char BYTE;

int main(void)
{
 unsigned int addr;
 int i, n;
 BYTE *ptr;

 printf("Address of main function: %x\n", (unsigned int) main);
 printf("Address of addr variable: %x\n", (unsigned int) &addr);
```

```

printf("\nEnter a (hex) address: ");
scanf("%x", &addr);
printf("Enter number of bytes to view: ");
scanf("%d", &n);

printf("\n");
printf(" Address Bytes Characters\n");
printf(" ----- ----- -----");
ptr = (BYTE *) addr;
for (; n > 0; n -= 10) {
 printf("%8X ", (unsigned int) ptr);
 for (i = 0; i < 10 && i < n; i++)
 printf("%.2X ", *(ptr + i));
 for (; i < 10; i++)
 printf(" ");
 printf(" ");
 for (i = 0; i < 10 && i < n; i++) {
 BYTE ch = *(ptr + i);
 if (!isprint(ch))
 ch = '.';
 printf("%c", ch);
 }
 printf("\n");
 ptr += 10;
}
return 0;
}

```

The program is complicated somewhat by the possibility that the value of *n* isn't a multiple of 10, so there may be fewer than 10 bytes in the last group. Two of the `for` statements are controlled by the condition *i* < 10 && *i* < *n*. This condition causes the loops to execute 10 times or *n* times, whichever is smaller. There's also a `for` statement that compensates for any missing bytes in the last group by displaying three spaces for each missing byte. That way, the characters that follow the last group of bytes will align properly with the character groups on previous lines.

The `%X` conversion specifier used in this program is similar to `%x`, which was discussed in Section 7.1. The difference is that `%X` displays the hexadecimal digits A, B, C, D, E, and F as upper-case letters; `%x` displays them in lower case.

Here's what happened when I compiled the program using GCC and tested it on an x86 system running Linux:

```
Address of main function: 804847C
Address of addr variable: bff41154
```

```
Enter a (hex) address: 8048000
Enter number of bytes to view: 40
```

| Address | Bytes                         | Characters |
|---------|-------------------------------|------------|
| -----   | -----                         | -----      |
| 8048000 | 7F 45 4C 46 01 01 01 00 00 00 | .ELF.....  |
| 804800A | 00 00 00 00 00 00 02 00 03 00 | .....      |
| 8048014 | 01 00 00 00 C0 83 04 08 34 00 | .....4.    |
| 804801E | 00 00 C0 0A 00 00 00 00 00 00 | .....      |

I asked the program to display 40 bytes starting at address 8048000, which precedes the address of the main function. Note the 7F byte followed by bytes representing the letters E, L, and F. These four bytes identify the format (ELF) in which the executable file was stored. ELF (Executable and Linking Format) is widely used by UNIX systems, including Linux. 8048000 is the default address at which ELF executables are loaded on x86 platforms.

Let's run the program again, this time displaying a block of bytes that starts at the address of the `addr` variable:

```
Address of main function: 804847C
Address of addr variable: bfec5484

Enter a (hex) address: bfec5484
Enter number of bytes to view: 64
```

| Address  | Bytes                         | Characters |
|----------|-------------------------------|------------|
| BFEC5484 | 84 54 EC BF B0 54 EC BF F4 6F | .T...T...o |
| BFEC548E | 68 00 34 55 EC BF C0 54 EC BF | h.4U...T.. |
| BFEC5498 | 08 55 EC BF E3 3D 57 00 00 00 | .U...=W... |
| BFEC54A2 | 00 00 A0 BC 55 00 08 55 EC BF | ....U..U.. |
| BFEC54AC | E3 3D 57 00 01 00 00 00 34 55 | =W.....4U  |
| BFEC54B6 | EC BF 3C 55 EC BF 56 11 55 00 | ..<U..V.U. |
| BFEC54C0 | F4 6F 68 00                   | .oh.       |

None of the data stored in this region of memory is in character form, so it's a bit hard to follow. However, we do know one thing: the `addr` variable occupies the first four bytes of this region. When reversed, these bytes form the number BFEC5484, the address entered by the user. Why the reversal? Because x86 processors store data in little-endian order, as we saw earlier in this section.

## The `volatile` Type Qualifier

On some computers, certain memory locations are “volatile”; the value stored at such a location can change as a program is running, even though the program itself isn't storing new values there. For example, some memory locations might hold data coming directly from input devices.

The `volatile` type qualifier allows us to inform the compiler if any of the data used in a program is volatile. `volatile` typically appears in the declaration of a pointer variable that will point to a volatile memory location:

```
volatile BYTE *p; /* p will point to a volatile byte */
```

To see why `volatile` is needed, suppose that `p` points to a memory location that contains the most recent character typed at the user's keyboard. This location is volatile: its value changes each time the user enters a character. We might use the following loop to obtain characters from the keyboard and store them in a buffer array:

```

while (buffer not full) {
 wait for input;
 buffer[i] = *p;
 if (buffer[i++] == '\n')
 break;
}

```

A sophisticated compiler might notice that this loop changes neither *p* nor *\*p*, so it could optimize the program by altering it so that *\*p* is fetched just once:

```

store *p in a register;
while (buffer not full) {
 wait for input;
 buffer[i] = value stored in register;
 if (buffer[i++] == '\n')
 break;
}

```

The optimized program will fill the buffer with many copies of the same character—not exactly what we had in mind. Declaring that *p* points to volatile data avoids this problem by telling the compiler that *\*p* must be fetched from memory each time it's needed.

## Q & A

- Q: What do you mean by saying that the `&` and `|` operators sometimes produce the same results as the `&&` and `||` operators, but not always? [p. 511]**
- A:** Let's compare *i* `&` *j* with *i* `&&` *j* (similar remarks apply to `|` and `||`). As long as *i* and *j* have the value 0 or 1 (in any combination), the two expressions will have the same value. However, if *i* and *j* should have other values, the expressions may not always match. If *i* is 1 and *j* is 2, for example, then *i* `&` *j* has the value 0 (*i* and *j* have no corresponding 1 bits), while *i* `&&` *j* has the value 1. If *i* is 3 and *j* is 2, then *i* `&` *j* has the value 2, while *i* `&&` *j* has the value 1.
- Side effects are another difference. Evaluating *i* `&` *j* `++` *always* increments *j* as a side effect, whereas evaluating *i* `&&` *j* `++` *sometimes* increments *j*.
- Q: Who cares how DOS stores file dates? Isn't DOS dead? [p. 516]**
- A:** For the most part, yes. However, there are still plenty of files created years ago whose dates are stored in the DOS format. In any event, DOS file dates are a good example of how bit-fields are used.
- Q: Where do the terms “big-endian” and “little-endian” come from? [p. 520]**
- A:** In Jonathan Swift's novel *Gulliver's Travels*, the fictional islands of Lilliput and Blefuscus are perpetually at odds over whether to open boiled eggs on the big end or the little end. The choice is arbitrary, of course, just like the order of bytes in a data item.

## Exercises

### Section 20.1

- \*1. Show the output produced by each of the following program fragments. Assume that *i*, *j*, and *k* are unsigned short variables.
- i* = 8; *j* = 9;  
`printf("%d", i >> 1 + j >> 1);`
  - i* = 1;  
`printf("%d", i & ~i);`
  - i* = 2; *j* = 1; *k* = 0;  
`printf("%d", ~i & j ^ k);`
  - i* = 7; *j* = 8; *k* = 9;  
`printf("%d", i ^ j & k);`
- W 2. Describe a simple way to “toggle” a bit (change it from 0 to 1 or from 1 to 0). Illustrate the technique by writing a statement that toggles bit 4 of the variable *i*.
- \*3. Explain what effect the following macro has on its arguments. You may assume that the arguments have the same type.
- ```
#define M(x,y) ((x)^=(y), (y)^=(x), (x)^=(y))
```
- W 4. In computer graphics, colors are often stored as three numbers, representing red, green, and blue intensities. Suppose that each number requires eight bits, and we’d like to store all three values in a single long integer. Write a macro named MK_COLOR with three parameters (the red, green, and blue intensities). MK_COLOR should return a long in which the last three bytes contain the red, green, and blue intensities, with the red value as the last byte and the green value as the next-to-last byte.
5. Write macros named GET_RED, GET_GREEN, and GET_BLUE that, when given a color as an argument (see Exercise 4), return its 8-bit red, green, and blue intensities.
- W 6. (a) Use the bitwise operators to write the following function:
- ```
unsigned short swap_bytes(unsigned short i);
```
- swap\_bytes* should return the number that results from swapping the two bytes in *i*. (Short integers occupy two bytes on most computers.) For example, if *i* has the value 0x1234 (00010010 00110100 in binary), then *swap\_bytes* should return 0x3412 (00110100 00010010 in binary). Test your function by writing a program that reads a number in hexadecimal, then writes the number with its bytes swapped:
- ```
Enter a hexadecimal number (up to four digits): 1234
Number with bytes swapped: 3412
```
- Hint:* Use the %hx conversion to read and write the hex numbers.
- (b) Condense the *swap_bytes* function so that its body is a single statement.
7. Write the following functions:
- ```
unsigned int rotate_left(unsigned int i, int n);
unsigned int rotate_right(unsigned int i, int n);
```
- rotate\_left* should return the result of shifting the bits in *i* to the left by *n* places, with the bits that were “shifted off” moved to the right end of *i*. (For example, the call

`rotate_left(0x12345678, 4)` should return `0x23456781` if integers are 32 bits long.) `rotate_right` is similar, but it should “rotate” bits to the right instead of the left.

- W 8. Let `f` be the following function:

```
unsigned int f(unsigned int i, int m, int n)
{
 return (i >> (m + 1 - n)) & ~(~0 << n);
```

- (a) What is the value of `~(~0 << n)`?  
 (b) What does this function do?

9. (a) Write the following function:

```
int count_ones(unsigned char ch);
count_ones should return the number of 1 bits in ch.
```

- (b) Write the function in part (a) without using a loop.

10. Write the following function:

```
unsigned int reverse_bits(unsigned int n);
```

`reverse_bits` should return an unsigned integer whose bits are the same as those in `n` but in reverse order.

11. Each of the following macros defines the position of a single bit within an integer:

```
#define SHIFT_BIT 1
#define CTRL_BIT 2
#define ALT_BIT 4
```

The following statement is supposed to test whether any of the three bits have been set, but it never displays the specified message. Explain why the statement doesn’t work and show how to fix it. Assume that `key_code` is an `int` variable.

```
if (key_code & (SHIFT_BIT | CTRL_BIT | ALT_BIT) == 0)
 printf("No modifier keys pressed\n");
```

12. The following function supposedly combines two bytes to form an unsigned short integer. Explain why the function doesn’t work and show how to fix it.

```
unsigned short create_short(unsigned char high_byte,
 unsigned char low_byte)
{
 return high_byte << 8 + low_byte;
```

- \*13. If `n` is an `unsigned int` variable, what effect does the following statement have on the bits in `n`?

```
n &= n - 1;
```

*Hint:* Consider the effect on `n` if this statement is executed more than once.

## Section 20.2

- W 14. When stored according to the IEEE floating-point standard, a `float` value consists of a 1-bit sign (the leftmost—or most significant—bit), an 8-bit exponent, and a 23-bit fraction, in that order. Design a structure type that occupies 32 bits, with bit-field members corresponding to the sign, exponent, and fraction. Declare the bit-fields to have type `unsigned int`. Check the manual for your compiler to determine the order of the bit-fields.

- \*15. (a) Assume that the variable `s` has been declared as follows:

```
struct {
 int flag: 1;
} s;
```

With some compilers, executing the following statements causes 1 to be displayed, but with other compilers, the output is -1. Explain the reason for this behavior.

```
s.flag = 1;
printf("%d\n", s.flag);
```

- (b) How can this problem be avoided?

### Section 20.3

16. Starting with the 386 processor, x86 CPUs have 32-bit registers named EAX, EBX, ECX, and EDX. The second half (the least significant bits) of these registers is the same as AX, BX, CX, and DX, respectively. Modify the `regs` union so that it includes these registers as well as the older ones. Your union should be set up so that modifying EAX changes AX and modifying AX changes the second half of EAX. (The other new registers will work in a similar fashion.) You'll need to add some "dummy" members to the `word` and `byte` structures, corresponding to the other half of EAX, EBX, ECX, and EDX. Declare the type of the new registers to be `DWORD` (double word), which should be defined as `unsigned long`. Don't forget that the x86 architecture is little-endian.

## Programming Projects

1. Design a union that makes it possible to view a 32-bit value as either a `float` or the structure described in Exercise 14. Write a program that stores 1 in the structure's sign field, 128 in the exponent field, and 0 in the fraction field, then prints the `float` value stored in the union. (The answer should be -2.0 if you've set up the bit-fields correctly.)



# 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

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

|                                            |                                             |                                            |                                           |
|--------------------------------------------|---------------------------------------------|--------------------------------------------|-------------------------------------------|
| <code>&lt;assert.h&gt;</code>              | <code>&lt;inttypes.h&gt;<sup>†</sup></code> | <code>&lt;signal.h&gt;</code>              | <code>&lt;stdlib.h&gt;</code>             |
| <code>&lt;complex.h&gt;<sup>†</sup></code> | <code>&lt;iso646.h&gt;<sup>†</sup></code>   | <code>&lt;stdarg.h&gt;</code>              | <code>&lt;string.h&gt;</code>             |
| <code>&lt;ctype.h&gt;</code>               | <code>&lt;limits.h&gt;</code>               | <code>&lt;stdbool.h&gt;<sup>†</sup></code> | <code>&lt;tgmath.h&gt;<sup>†</sup></code> |
| <code>&lt;errno.h&gt;</code>               | <code>&lt;locale.h&gt;</code>               | <code>&lt;stddef.h&gt;</code>              | <code>&lt;time.h&gt;</code>               |
| <code>&lt;fenv.h&gt;<sup>†</sup></code>    | <code>&lt;math.h&gt;</code>                 | <code>&lt;stdint.h&gt;<sup>†</sup></code>  | <code>&lt;wchar.h&gt;<sup>†</sup></code>  |
| <code>&lt;float.h&gt;</code>               | <code>&lt;setjmp.h&gt;</code>               | <code>&lt;stdio.h&gt;</code>               | <code>&lt;wctype.h&gt;<sup>†</sup></code> |

<sup>†</sup>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.

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.

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>`.

`fwrite` function ▶ 22.6

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

## Programming Projects

- 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          |

**Q&A**

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*.

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.
- ***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.

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.

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

C99

restrict keyword ▶ 17.8

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

| <i>String</i> | <i>Meaning</i>                                         |
|---------------|--------------------------------------------------------|
| "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

| <i>String</i>  | <i>Meaning</i>                                         |
|----------------|--------------------------------------------------------|
| "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>`).

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 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:

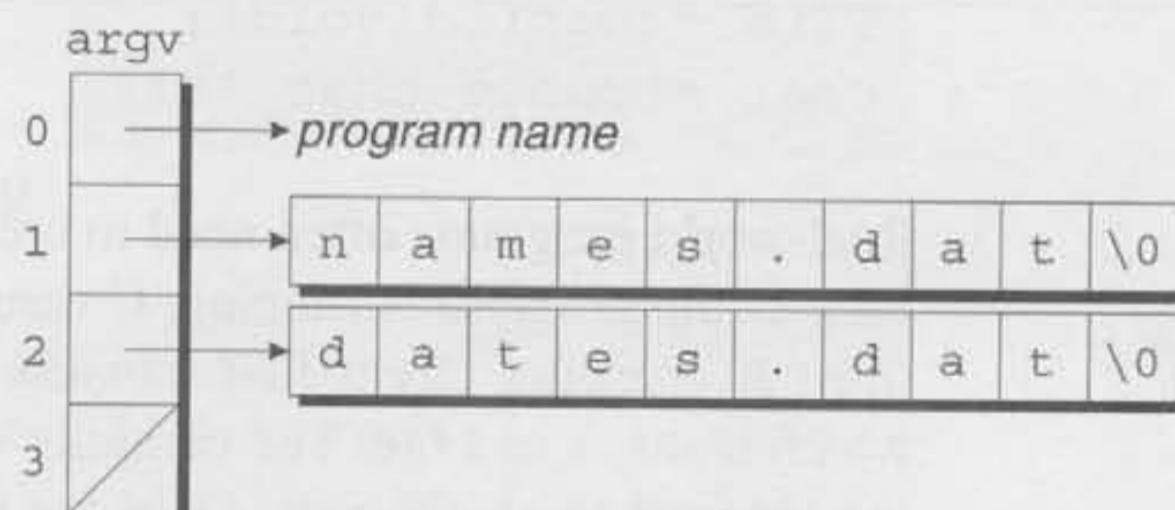
```
demo names.dat dates.dat
```

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[])
{
 ...
}
```

### Q&A

`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;
...
temp	ptr = 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

```
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.

**Q&A**

**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");
```

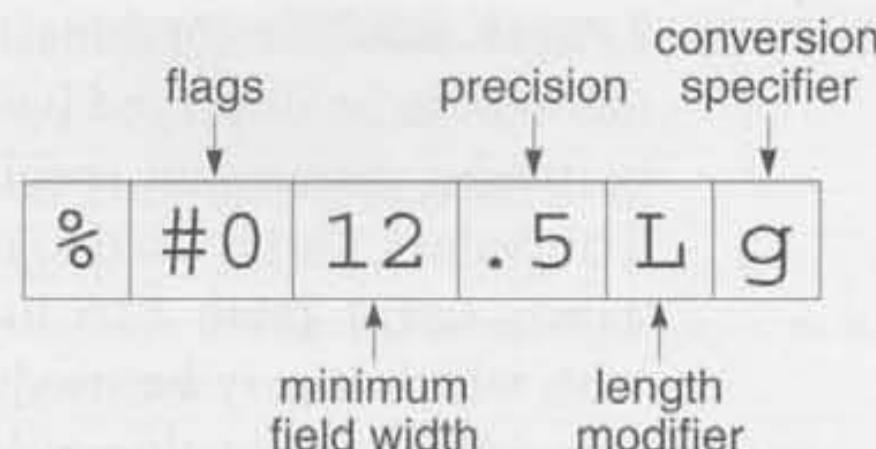
Writing the message to `stderr` guarantees that it will appear on the screen even if the user redirects `stdout`.

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<br>(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<br>Modifier           | Conversion Specifiers  | Meaning                               |
|------------------------------|------------------------|---------------------------------------|
| hh <sup>†</sup>              | 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<br>(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 <sup>†</sup><br>(ell-ell) | d, i, o, u, x, X       | long long int, unsigned long long int |
|                              | n                      | long long int *                       |
| j <sup>†</sup>               | d, i, o, u, x, X       | intmax_t, uintmax_t                   |
|                              | n                      | intmax_t *                            |
| z <sup>†</sup>               | d, i, o, u, x, X       | size_t                                |
|                              | n                      | size_t *                              |
| t <sup>†</sup>               | d, i, o, u, x, X       | ptrdiff_t                             |
|                              | n                      | ptrdiff_t *                           |
| L                            | a, A, e, E, f, F, g, G | long double                           |

<sup>†</sup>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 <sup>†</sup>               | 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 <sup>†</sup> , A <sup>†</sup> | Converts a double value to hexadecimal scientific notation using the form [-]0xh. hhhhp±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 %.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |

<sup>†</sup>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.

IEEE floating-point standard ▶ 23.4

wide characters ▶ 25.2

- **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.
- **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).
- **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.

C99

multibyte characters ▶ 25.2

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.

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.** 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.

white-space characters ▶ 3.2

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 <sup>†</sup>              | 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<br>(ell)                   | d, i, o, u, x, X, n<br>a, A, e, E, f, F, g, G<br>c, s, or [ | long int *, unsigned long int *<br>double *<br>wchar_t * |
| ll <sup>†</sup><br>(ell-ell) | d, i, o, u, x, X, n                                         | long long int *,<br>unsigned long long int *             |
| j <sup>†</sup>               | d, i, o, u, x, X, n                                         | intmax_t *, uintmax_t *                                  |
| z <sup>†</sup>               | d, i, o, u, x, X, n                                         | size_t *                                                 |
| t <sup>†</sup>               | d, i, o, u, x, X, n                                         | ptrdiff_t *                                              |
| L                            | a, A, e, E, f, F, g, G                                      | long double *                                            |

<sup>†</sup>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

| <i>Conversion Specifier</i>                                         | <i>Meaning</i>                                                                                                                                                                                                                                                                                        |
|---------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| d                                                                   | Matches a decimal integer; the corresponding argument is assumed to have type <code>int *</code> .                                                                                                                                                                                                    |
| i                                                                   | Matches an integer; the corresponding argument is assumed to have type <code>int *</code> . 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 <code>unsigned int *</code> .                                                                                                                                                                                            |
| u                                                                   | Matches a decimal integer; the corresponding argument is assumed to have type <code>unsigned int *</code> .                                                                                                                                                                                           |
| x, X                                                                | Matches a hexadecimal integer; the corresponding argument is assumed to have type <code>unsigned int *</code> .                                                                                                                                                                                       |
| a <sup>†</sup> , A <sup>†</sup> , e, E,<br>f, F <sup>†</sup> , g, G | Matches a floating-point number; the corresponding argument is assumed to have type <code>float *</code> . In C99, the number can be infinity or NaN.                                                                                                                                                 |
| c                                                                   | Matches <i>n</i> characters, where <i>n</i> 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 <code>void *</code> object.                                                                                                                                      |
| n                                                                   | The corresponding argument must point to an object of type <code>int</code> . 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 %.                                                                                                                                                                                                                                                                              |

<sup>†</sup>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]

numeric conversion functions ➤ 26.2

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

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

| <i>Conversion Specifier</i> | <i>Numeric Conversion Function</i> |
|-----------------------------|------------------------------------|
| 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**

## 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)

|  | <code>scanf Call</code>                             | <code>Input</code>                                                     | <code>Variables</code>                                               |
|--|-----------------------------------------------------|------------------------------------------------------------------------|----------------------------------------------------------------------|
|  | <code>n = scanf ("%d%d", &amp;i, &amp;j);</code>    | <del>12</del> <sup>*</sup> , <del>•</del> <sup>*</sup> 34 <del>□</del> | <code>n: 1</code><br><code>i: 12</code><br><code>j: unchanged</code> |
|  | <code>n = scanf ("%d, %d", &amp;i, &amp;j);</code>  | <del>12</del> <sup>*</sup> , <del>•</del> <sup>*</sup> 34 <del>□</del> | <code>n: 1</code><br><code>i: 12</code><br><code>j: unchanged</code> |
|  | <code>n = scanf ("%d , %d", &amp;i, &amp;j);</code> | <del>12</del> <sup>*</sup> , <del>•</del> <sup>*</sup> 34 <del>□</del> | <code>n: 2</code><br><code>i: 12</code><br><code>j: 34</code>        |
|  | <code>n = scanf ("%d, %d", &amp;i, &amp;j);</code>  | <del>12</del> <sup>*</sup> , <del>•</del> <sup>*</sup> 34 <del>□</del> | <code>n: 1</code><br><code>i: 12</code><br><code>j: unchanged</code> |

**Table 22.15**  
`scanf` Examples  
(Group 2)

|  | <code>scanf Call</code>                                       | <code>Input</code>                                                                   | <code>Variables</code>                                                                  |
|--|---------------------------------------------------------------|--------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------|
|  | <code>n = scanf ("%*d%d", &amp;i);</code>                     | <del>12</del> <sup>*</sup> <del>34</del> <del>□</del>                                | <code>n: 1</code><br><code>i: 34</code>                                                 |
|  | <code>n = scanf ("%*s%s", str);</code>                        | <del>My</del> <sup>*</sup> <del>Fair</del> <sup>*</sup> <del>Lady</del> <del>□</del> | <code>n: 1</code><br><code>str: "Fair"</code>                                           |
|  | <code>n = scanf ("%1d%2d%3d", &amp;i, &amp;j, &amp;k);</code> | <del>12345</del> <del>□</del>                                                        | <code>n: 3</code><br><code>i: 1</code><br><code>j: 23</code><br><code>k: 45</code>      |
|  | <code>n = scanf ("%2d%2s%2d", &amp;i, str, &amp;j);</code>    | <del>123456</del> <del>□</del>                                                       | <code>n: 3</code><br><code>i: 12</code><br><code>str: "34"</code><br><code>j: 56</code> |

**Table 22.16**  
**scanf Examples**  
 (Group 3)

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

**Q&A**

**feof**  
**ferror**

**Q&A**

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

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.

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) /* can't open file */
 return -1;

 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` `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.

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.

### Q&A

## 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 */
```

file-positioning functions ▶ 22.7

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.

`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:

|                       |                       |
|-----------------------|-----------------------|
| <code>SEEK_SET</code> | Beginning of file     |
| <code>SEEK_CUR</code> | Current file position |
| <code>SEEK_END</code> | 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 `fseek` 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 `fseek` 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.

**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 "`%ls`" 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 |

fesetround function ➤ 27.6

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).

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                                                 |
|----------------------------|-----------|-------------------------------------------------------------|
| <code>FLT_MANT_DIG</code>  |           | Number of significant digits (base <code>FLT_RADIX</code> ) |
| <code>DBL_MANT_DIG</code>  |           |                                                             |
| <code>LDBL_MANT_DIG</code> |           |                                                             |
| <code>FLT_DIG</code>       | $\geq 6$  | Number of significant digits (base 10)                      |
| <code>DBL_DIG</code>       | $\geq 10$ |                                                             |
| <code>LDBL_DIG</code>      | $\geq 10$ |                                                             |

Table 23.3 lists macros having to do with exponents.

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

| Name                         | Value      | Description                                                                  |
|------------------------------|------------|------------------------------------------------------------------------------|
| <code>FLT_MIN_EXP</code>     |            | Smallest (most negative) power to which <code>FLT_RADIX</code> can be raised |
| <code>DBL_MIN_EXP</code>     |            |                                                                              |
| <code>LDBL_MIN_EXP</code>    |            |                                                                              |
| <code>FLT_MIN_10_EXP</code>  | $\leq -37$ | Smallest (most negative) power to which 10 can be raised                     |
| <code>DBL_MIN_10_EXP</code>  | $\leq -37$ |                                                                              |
| <code>LDBL_MIN_10_EXP</code> | $\leq -37$ |                                                                              |
| <code>FLT_MAX_EXP</code>     |            | Largest power to which <code>FLT_RADIX</code> can be raised                  |
| <code>DBL_MAX_EXP</code>     |            |                                                                              |
| <code>LDBL_MAX_EXP</code>    |            |                                                                              |
| <code>FLT_MAX_10_EXP</code>  | $\geq +37$ | Largest power to which 10 can be raised                                      |
| <code>DBL_MAX_10_EXP</code>  | $\geq +37$ |                                                                              |
| <code>LDBL_MAX_10_EXP</code> | $\geq +37$ |                                                                              |

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 <code>float</code> and <code>double</code> to the range and precision of the <code>double</code> type |
| 2     | Evaluate all operations and constants to the range and precision of the <code>long double</code> 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   | $\geq 8$    | Number of bits per byte                                                                       |
| SCHAR_MIN  | $\leq -127$ | Minimum signed char value                                                                     |
| SCHAR_MAX  | $\geq +127$ | Maximum signed char value                                                                     |
| UCHAR_MAX  | $\geq 255$  | Maximum unsigned char value                                                                   |
| CHAR_MIN   | $\dagger$   | Minimum char value                                                                            |
| CHAR_MAX   | $\ddagger$  | Maximum char value                                                                            |
| MB_LEN_MAX | $\geq 1$    | Maximum number of bytes per multibyte character<br>in any supported locale (see Section 25.2) |

$\dagger$ CHAR\_MIN is equal to SCHAR\_MIN if `char` is treated as a signed type; otherwise, CHAR\_MIN is 0.

$\ddagger$ 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                | $\leq -32767$               | $-(2^{15}-1)$ | Minimum short int value              |
| SHRT_MAX                | $\geq +32767$               | $2^{15}-1$    | Maximum short int value              |
| USHRT_MAX               | $\geq 65535$                | $2^{16}-1$    | Maximum unsigned short int value     |
| INT_MIN                 | $\leq -32767$               | $-(2^{15}-1)$ | Minimum int value                    |
| INT_MAX                 | $\geq +32767$               | $2^{15}-1$    | Maximum int value                    |
| UINT_MAX                | $\geq 65535$                | $2^{16}-1$    | Maximum unsigned int value           |
| LONG_MIN                | $\leq -2147483647$          | $-(2^{31}-1)$ | Minimum long int value               |
| LONG_MAX                | $\geq +2147483647$          | $2^{31}-1$    | Maximum long int value               |
| ULONG_MAX               | $\geq 4294967295$           | $2^{32}-1$    | Maximum unsigned long int value      |
| LLONG_MIN <sup>†</sup>  | $\leq -9223372036854775807$ | $-(2^{63}-1)$ | Minimum long long int value          |
| LLONG_MAX <sup>†</sup>  | $\geq +9223372036854775807$ | $2^{63}-1$    | Maximum long long int value          |
| ULLONG_MAX <sup>†</sup> | $\geq 18446744073709551615$ | $2^{64}-1$    | Maximum unsigned long long int value |

$\dagger$ 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`.)

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`

NaN ▶ 23.4

(“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.

underflow ▶ 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`.

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  $\pi$ . `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  $\pi$ . 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**      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**      `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  $0.5 \leq f < 1$  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  $x \times 2^{\text{exp}}$ .

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** 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 $\langle\text{math.h}\rangle$ 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`.

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

| <code>Value of<br/>FLT_EVAL_METHOD</code> | <code>Meaning of<br/>float_t</code> | <code>Meaning of<br/>double_t</code> |
|-------------------------------------------|-------------------------------------|--------------------------------------|
| 0                                         | <code>float</code>                  | <code>double</code>                  |
| 1                                         | <code>double</code>                 | <code>double</code>                  |
| 2                                         | <code>long double</code>            | <code>long double</code>             |
| 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 `sqrto` (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);
```

*fpclassify*

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`).

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

*isfinite*  
*isinf*  
*isnan*  
*isnormal*

*signbit*

| Name                      | Meaning                                        |
|---------------------------|------------------------------------------------|
| <code>FP_INFINITE</code>  | Infinity (positive or negative)                |
| <code>FP_NAN</code>       | Not a number                                   |
| <code>FP_NORMAL</code>    | Normal (not zero, subnormal, infinite, or NaN) |
| <code>FP_SUBNORMAL</code> | Subnormal                                      |
| <code>FP_ZERO</code>      | Zero (positive or negative)                    |

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).

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

|                                                |                               |
|------------------------------------------------|-------------------------------|
| <code>float acosf(float x);</code>             | <i>see</i> <code>acos</code>  |
| <code>long double acosl(long double x);</code> | <i>see</i> <code>acos</code>  |
| <code>float asinf(float x);</code>             | <i>see</i> <code>asin</code>  |
| <code>long double asinl(long double x);</code> | <i>see</i> <code>asin</code>  |
| <code>float atanf(float x);</code>             | <i>see</i> <code>atan</code>  |
| <code>long double atanl(long double x);</code> | <i>see</i> <code>atan</code>  |
| <code>float atan2f(float y, float x);</code>   | <i>see</i> <code>atan2</code> |
| <code>long double atan2l(long double y,</code> |                               |
| <code>                  long double x);</code> | <i>see</i> <code>atan2</code> |
| <code>float cosf(float x);</code>              | <i>see</i> <code>cos</code>   |
| <code>long double cosl(long double x);</code>  | <i>see</i> <code>cos</code>   |
| <code>float sinf(float x);</code>              | <i>see</i> <code>sin</code>   |
| <code>long double sinl(long double x);</code>  | <i>see</i> <code>sin</code>   |
| <code>float tanf(float x);</code>              | <i>see</i> <code>tan</code>   |
| <code>long double tanl(long double x);</code>  | <i>see</i> <code>tan</code>   |

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

## Hyperbolic Functions

|                                                 |                              |
|-------------------------------------------------|------------------------------|
| <code>double acosh(double x);</code>            |                              |
| <code>float acoshf(float x);</code>             |                              |
| <code>long double acoshl(long double x);</code> |                              |
| <code>double asinh(double x);</code>            |                              |
| <code>float asinhf(float x);</code>             |                              |
| <code>long double asinhl(long double x);</code> |                              |
| <code>double atanh(double x);</code>            |                              |
| <code>float atanhf(float x);</code>             |                              |
| <code>long double atanhl(long double x);</code> |                              |
| <code>float coshf(float x);</code>              | <i>see</i> <code>cosh</code> |
| <code>long double coshl(long double x);</code>  | <i>see</i> <code>cosh</code> |
| <code>float sinhf(float x);</code>              | <i>see</i> <code>sinh</code> |
| <code>long double sinhl(long double x);</code>  | <i>see</i> <code>sinh</code> |
| <code>float tanhf(float x);</code>              | <i>see</i> <code>tanh</code> |
| <code>long double tanhl(long double x);</code>  | <i>see</i> <code>tanh</code> |

*acosh* 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

|                                                    |                  |
|----------------------------------------------------|------------------|
| <i>float expf(float x);</i>                        | <i>see exp</i>   |
| <i>long double expl(long double x);</i>            | <i>see exp</i>   |
| <i>double exp2(double x);</i>                      |                  |
| <i>float exp2f(float x);</i>                       |                  |
| <i>long double exp2l(long double x);</i>           |                  |
| <i>double expm1(double x);</i>                     |                  |
| <i>float expmlf(float x);</i>                      |                  |
| <i>long double expmll(long double x);</i>          |                  |
| <i>float frexpf(float value, int *exp);</i>        | <i>see frexp</i> |
| <i>long double frexpl(long double value,</i>       |                  |
| <i>                  int *exp);</i>                | <i>see frexp</i> |
| <i>int ilogb(double x);</i>                        |                  |
| <i>int ilogbf(float x);</i>                        |                  |
| <i>int ilogbl(long double x);</i>                  |                  |
| <i>float ldexpf(float x, int exp);</i>             | <i>see ldexp</i> |
| <i>long double ldexpl(long double x, int exp);</i> | <i>see ldexp</i> |
| <i>float logf(float x);</i>                        | <i>see log</i>   |
| <i>long double logl(long double x);</i>            | <i>see log</i>   |
| <i>float log10f(float x);</i>                      | <i>see log10</i> |
| <i>long double log10l(long double x);</i>          | <i>see log10</i> |
| <i>double log1p(double x);</i>                     |                  |
| <i>float log1pf(float x);</i>                      |                  |
| <i>long double log1pl(long double x);</i>          |                  |
| <i>double log2(double x);</i>                      |                  |
| <i>float log2f(float x);</i>                       |                  |
| <i>long double log2l(long double x);</i>           |                  |
| <i>double logb(double x);</i>                      |                  |
| <i>float logbf(float x);</i>                       |                  |
| <i>long double logbl(long double x);</i>           |                  |
| <i>float modff(float value, float *iptr);</i>      | <i>see modf</i>  |
| <i>long double modfl(long double value,</i>        |                  |
| <i>                  long double *iptr);</i>       | <i>see modf</i>  |

```
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`

**Q&A**  
`logb`  
`ilogb`  
`log1p`  
`log2`

`scalbn`  
`scalbln`

In addition 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  $e^x - 1$ .

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.

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

## Power and Absolute Value Functions

|                                                                |                 |
|----------------------------------------------------------------|-----------------|
| <code>double cbrt(double x);</code>                            |                 |
| <code>float cbrtf(float x);</code>                             |                 |
| <code>long double cbrtl(long double x);</code>                 |                 |
| <code>float fabsf(float x);</code>                             | <i>see fabs</i> |
| <code>long double fabsl(long double x);</code>                 | <i>see fabs</i> |
| <code>double hypot(double x, double y);</code>                 |                 |
| <code>float hypotf(float x, float y);</code>                   |                 |
| <code>long double hypotl(long double x, long double y);</code> |                 |
| <code>float powf(float x, float y);</code>                     | <i>see pow</i>  |
| <code>long double powl(long double x,</code>                   |                 |
| <code>long double y);</code>                                   | <i>see pow</i>  |
| <code>float sqrtf(float x);</code>                             | <i>see sqrt</i> |
| <code>long double sqrtl(long double x);</code>                 | <i>see sqrt</i> |

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

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 erfc1(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** The `erf` function computes the *error function*  $\text{erf}$  (also known as the *Gaussian error function*), which is used in probability, statistics and partial differential equations. The mathematical definition of  $\text{erf}$  is

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

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

**lgamma** The *gamma function*  $\Gamma$  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  $\Gamma$  for nonintegers is more complicated. The `tgamma` function computes  $\Gamma$ . 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  $\Gamma$  grows so quickly that using it in calculations may cause overflow.

**Q&A**

## Nearest Integer Functions

|                                                                      |                                      |
|----------------------------------------------------------------------|--------------------------------------|
| <pre>float ceilf(float x); long double ceill(long double x);</pre>   | <i>see ceil</i><br><i>see ceil</i>   |
| <pre>float floorf(float x); long double floorl(long double x);</pre> | <i>see floor</i><br><i>see floor</i> |

```

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*      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  $x \text{ REM } y$ , where *REM* is a function defined in the IEEE standard. For  $y \neq 0$ , the value of  $x \text{ REM } y$  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**

strtod function ➤ 26.2

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.)

**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*:

$$\begin{cases} x - y & \text{if } x > y \\ +0 & \text{if } x \leq y \end{cases}$$

*fmax*    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.

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.

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.

The `isunordered` macro returns 1 if its arguments are unordered (at least one of them is a `NAN`) and 0 otherwise.

*isgreater*  
*isgreaterequal*  
*isless*  
*islessequal*  
*islessgreater*  
*isunordered*

## 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

| <i>Function</i>          | <i>Test</i>                                            |
|--------------------------|--------------------------------------------------------|
| <code>isalnum(c)</code>  | Is <i>c</i> alphanumeric?                              |
| <code>isalpha(c)</code>  | Is <i>c</i> alphabetic?                                |
| <code>isblank(c)</code>  | Is <i>c</i> a blank? <sup>†</sup>                      |
| <code>iscntrl(c)</code>  | Is <i>c</i> a control character? <sup>††</sup>         |
| <code>isdigit(c)</code>  | Is <i>c</i> a decimal digit?                           |
| <code>isgraph(c)</code>  | Is <i>c</i> a printing character (other than a space)? |
| <code>islower(c)</code>  | Is <i>c</i> a lower-case letter?                       |
| <code>isprint(c)</code>  | Is <i>c</i> a printing character (including a space)?  |
| <code>ispunct(c)</code>  | Is <i>c</i> punctuation? <sup>†††</sup>                |
| <code>isspace(c)</code>  | Is <i>c</i> a white-space character? <sup>††††</sup>   |
| <code>isupper(c)</code>  | Is <i>c</i> an upper-case letter?                      |
| <code>isxdigit(c)</code> | Is <i>c</i> a hexadecimal digit?                       |

<sup>†</sup>The standard blank characters are space and horizontal tab (\t). This function is new in C99.

<sup>††</sup>In ASCII, the control characters are \x00 through \x1f plus \x7f.

<sup>†††</sup>All printing characters except those for which `isspace` or `isalnum` are true are considered punctuation.

<sup>††††</sup>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);
 TESTisspace);
 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     |        |
| z:    | x     | x     |       | x     | x     |        |
| A:    | x     | x     |       | x     | x     | x      |
| Z:    | x     | x     |       | x     | x     | x      |
| 0:    | 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);
```

locales ➤ 25.1

memcmp  
strcmp  
strncmp

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.

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` 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.

`strupr`

`strupr` 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 =strupr(str, "mn"); /* finds first 'm' or 'n' */
```

In this example, `p` will point to the letter `m` in `Form`. `strupr` returns a null pointer if no match is found.

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 non-empty sequence of characters that are *not* in the `s2` string. `strtok` marks the end

strspn  
strcspn

**Q&A**

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*

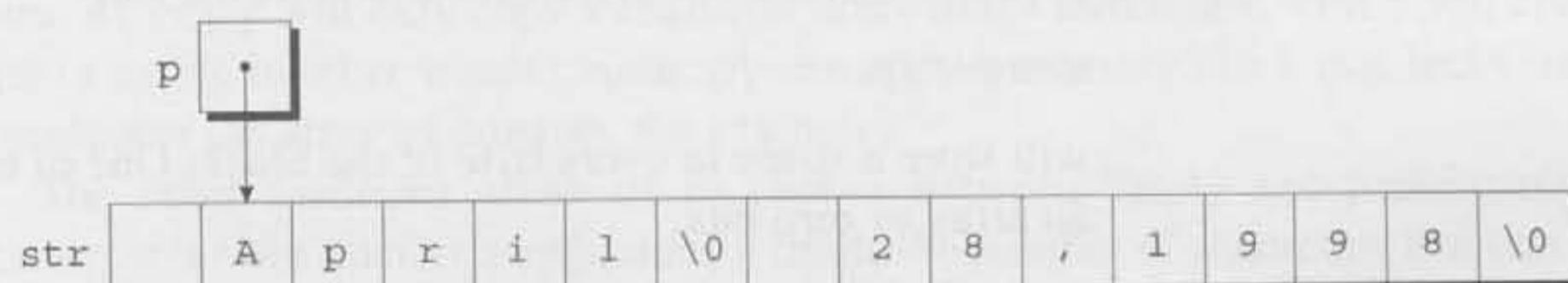
where spaces and/or tabs separate the month from the day and the day from the year. In addition, spaces and tabs may precede the comma. Let's say that the string *str* has the following appearance to start with:



After the call

```
p = strtok(str, " \t");
```

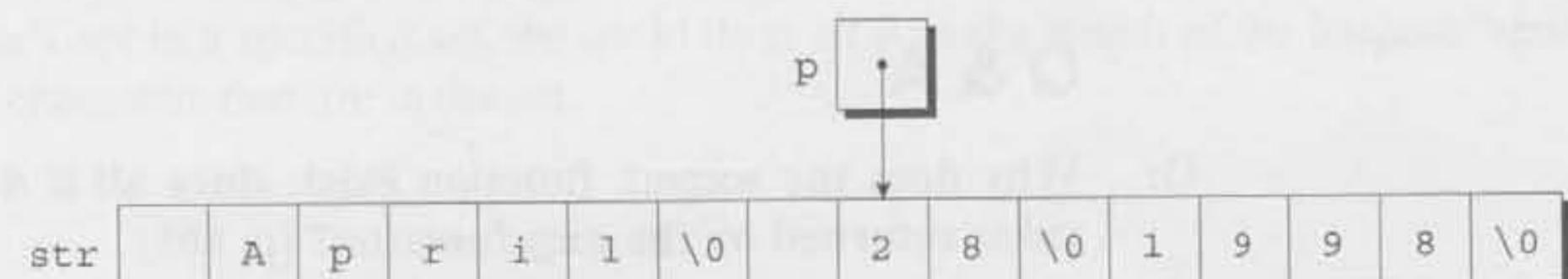
*str* will have the following appearance:



*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,");
```

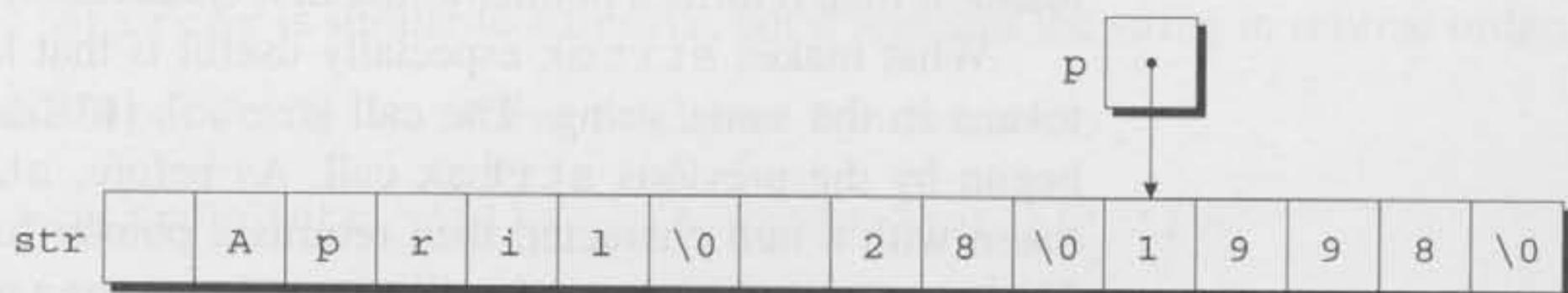
After this call, *p* points to the first character in the day:



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\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.

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  $a_nx^n + a_{n-1}x^{n-1} + \dots + a_0$ , 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:

$$((\dots((a_nx + a_{n-1})x + a_{n-2})x + \dots)x + a_1)x + a_0$$

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) `strcmpi(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  $\$1000 \times e^{0.06 \times 10} = \$1000 \times e^6 = \$1000 \times 1.8221188 = \$1,822.12$ . 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.

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

lvalues ▶ 4.2

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.

sqrt function ▶ 23.3

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:

```
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**  
`<wchar.h>` header ➤ 25.5  
 encoding error ➤ 22.3

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.

## The `perror` and `strerror` Functions

```
void perror(const char *s);
char *strerror(int errnum);
```

*from <stdio.h>  
 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**  
`stderr` stream ➤ 22.1

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.

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 <code>abort</code> )      |
| 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.

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 */
...
```

abort function ➤ 26.2

static storage duration ➤ 18.2

**Q&A**

```
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

**setjmp****Q&A****longjmp**

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.

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.

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**  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**  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*:

- `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.
- `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`.
- `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.

`<string.h>` header ▶ 23.6

`<ctype.h>` header ▶ 23.5

numeric conversion functions ▶ 26.2

`strftime` function ▶ 26.3

C99

`wcsftime` function ▶ 25.5

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),

UTF-8 ▶ 25.2 "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).

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/](http://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<br>"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 <sup>†</sup>                       |

<sup>†</sup>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<br>"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 <sup>†</sup>  | CHAR_MAX            | 1 if int_curr_symbol precedes nonnegative quantity; 0 if it succeeds quantity            |
| int_n_cs_precedes <sup>†</sup>  | CHAR_MAX            | 1 if int_curr_symbol precedes negative quantity; 0 if it succeeds quantity               |
| int_p_sep_by_space <sup>†</sup> | CHAR_MAX            | Separation of int_curr_symbol and sign string from nonnegative quantity (see Table 25.4) |
| int_n_sep_by_space <sup>†</sup> | CHAR_MAX            | Separation of int_curr_symbol and sign string from negative quantity (see Table 25.4)    |
| int_p_sign_posn <sup>†</sup>    | CHAR_MAX            | Position of positive_sign for nonnegative quantity (see Table 25.5)                      |
| int_n_sign_posn <sup>†</sup>    | CHAR_MAX            | Position of negative_sign for negative quantity (see Table 25.5)                         |

<sup>†</sup>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

| <i>Member</i>      | <i>U.S.A.</i> | <i>Finland</i> |
|--------------------|---------------|----------------|
| 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:

|                                 | <i>U.S.A.</i> | <i>Finland</i> |
|---------------------------------|---------------|----------------|
| 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`.

**C99**

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*.

## 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

**Q&A**

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.

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](http://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

| <i>Code Point Range<br/>(Hexadecimal)</i> | <i>UTF-8 Byte Sequence<br/>(Binary)</i> |
|-------------------------------------------|-----------------------------------------|
| 000000-00007F                             | 0xxxxxxxx                               |
| 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 11000011 10100100. 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

tokens ▶ 2.8

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.

**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 `<iiso646.h>` header, described next, helps with this problem.

**C99**

### The `<iiso646.h>` Header: Alternative Spellings

The `<iiso646.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  
`<iiso646.h>`

| Macro               | Value                   |
|---------------------|-------------------------|
| <code>and</code>    | <code>&amp;&amp;</code> |
| <code>and_eq</code> | <code>&amp;=</code>     |
| <code>bitand</code> | <code>&amp;</code>      |
| <code>bitor</code>  | <code> </code>          |
| <code>compl</code>  | <code>~</code>          |
| <code>not</code>    | <code>!</code>          |
| <code>not_eq</code> | <code>!=</code>         |
| <code>or</code>     | <code>  </code>         |
| <code>or_eq</code>  | <code> =</code>         |
| <code>xor</code>    | <code>^</code>          |
| <code>xor_eq</code> | <code>^=</code>         |

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 `<iiso646.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 (`\udddd` and `\Uddddddddd`), where each *d* is a hexadecimal digit. In the form `\Uddddddddd`, the *d*'s form an eight-digit hexadecimal number that identifies the UCS code point of the desired character. The form `\udddd` 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/](http://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.

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.

standard streams ▶ 22.1

`freopen` function ▶ 22.2

`errno` variable ▶ 24.2

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 `fwipe` 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.

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.

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

| <code>&lt;wchar.h&gt;</code> Function | <code>&lt;stdio.h&gt;</code> Equivalent |
|---------------------------------------|-----------------------------------------|
| <code>fwprintf</code>                 | <code>fprintf</code>                    |
| <code>fwscanf</code>                  | <code>fscanf</code>                     |
| <code>swprintf</code>                 | <code>snprintf, sprintf</code>          |
| <code>swscanf</code>                  | <code>sscanf</code>                     |
| <code>vfwprintf</code>                | <code>vfprintf</code>                   |
| <code>vfwscanf</code>                 | <code>vfscanf</code>                    |
| <code>vswprintf</code>                | <code>vsnprintf, vsprintf</code>        |
| <code>vswscanf</code>                 | <code>vscanf</code>                     |
| <code>vwprintf</code>                 | <code>vprintf</code>                    |
| <code>vwscanf</code>                  | <code>vscanf</code>                     |
| <code>wprintf</code>                  | <code>printf</code>                     |
| <code>wscanf</code>                   | <code>scanf</code>                      |

***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` 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

| <code>&lt;wchar.h&gt;</code> Function | <code>&lt;stdio.h&gt;</code> Equivalent |
|---------------------------------------|-----------------------------------------|
| <code>fgetwc</code>                   | <code>fgetc</code>                      |
| <code>fgetws</code>                   | <code>fgets</code>                      |
| <code>fputwc</code>                   | <code>fputc</code>                      |
| <code>fputws</code>                   | <code>fputs</code>                      |
| <code>fwide</code>                    | —                                       |
| <code>getwc</code>                    | <code>getc</code>                       |
| <code>getwchar</code>                 | <code>getchar</code>                    |
| <code>putwc</code>                    | <code>putc</code>                       |
| <code>putwchar</code>                 | <code>putchar</code>                    |
| <code>ungetwc</code>                  | <code>ungetc</code>                     |

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*

*fputwc  
putwc  
putwchar  
fputws*

*fwide*

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.

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.

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

| <code>&lt;wchar.h&gt;</code> Function | <code>&lt;stdlib.h&gt;</code> Equivalent |
|---------------------------------------|------------------------------------------|
| <code>wcstod</code>                   | <code>strtod</code>                      |
| <code>wcstof</code>                   | <code>strtof</code>                      |
| <code>wcstold</code>                  | <code>strtold</code>                     |
| <code>wcstol</code>                   | <code>strtol</code>                      |
| <code>wcstoll</code>                  | <code>strtoll</code>                     |
| <code>wcstoul</code>                  | <code>strtoul</code>                     |
| <code>wcstoull</code>                 | <code>strtoull</code>                    |

### 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

| <i>&lt;wchar.h&gt;</i> Function | <i>&lt;string.h&gt;</i> Equivalent |
|---------------------------------|------------------------------------|
| wcscpy                          | strcpy                             |
| wcsncpy                         | strncpy                            |
| wmemcp                          | 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

| <i>&lt;wchar.h&gt;</i> Function | <i>&lt;string.h&gt;</i> Equivalent |
|---------------------------------|------------------------------------|
| wcscat                          | strcat                             |
| wcsncat                         | strncat                            |

### Wide-String Comparison Functions

```
int wcscmp(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

| <code>&lt;wchar.h&gt;</code> Function | <code>&lt;string.h&gt;</code> Equivalent |
|---------------------------------------|------------------------------------------|
| <code>wcsncmp</code>                  | <code>strcmp</code>                      |
| <code>wcscoll</code>                  | <code>strcoll</code>                     |
| <code>wcsncmp</code>                  | <code>strncmp</code>                     |
| <code>wcsxfrm</code>                  | <code>strxfrm</code>                     |
| <code>wmemcmp</code>                  | <code>memcmp</code>                      |

### 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 *wmemcmp(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

| <code>&lt;wchar.h&gt;</code> Function | <code>&lt;string.h&gt;</code> Equivalent |
|---------------------------------------|------------------------------------------|
| <code>wcschr</code>                   | <code>strchr</code>                      |
| <code>wcscspn</code>                  | <code>strcspn</code>                     |
| <code>wcspbrk</code>                  | <code>strpbrk</code>                     |
| <code>wcsrchr</code>                  | <code>strrchr</code>                     |
| <code>wcsspn</code>                   | <code>strspn</code>                      |
| <code>wcsstr</code>                   | <code>strstr</code>                      |
| <code>wcstok</code>                   | <code>strtok</code>                      |
| <code>wmemcmp</code>                  | <code>memchr</code>                      |

`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

| <i>&lt;wchar.h&gt;</i> Function | <i>&lt;string.h&gt;</i> Equivalent |
|---------------------------------|------------------------------------|
| <i>wcslen</i>                   | <i>strlen</i>                      |
| <i>wmemset</i>                  | <i>memset</i>                      |

## 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.

- |              |                                                                                                                                                                                                                                                                                                                                      |
|--------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>btowc</i> | The <code>btowc</code> function returns <code>WEOF</code> if <code>c</code> is equal to <code>EOF</code> or if <code>c</code> (when cast to <code>unsigned char</code> ) isn't a valid single-byte character in the initial shift state. Otherwise, <code>btowc</code> returns the wide-character representation of <code>c</code> . |
| <i>wctob</i> | The <code>wctob</code> function is the opposite of <code>btowc</code> . It returns <code>EOF</code> if <code>c</code> doesn't correspond to one multibyte character in the initial shift state. Otherwise, it returns the single-byte representation of <code>c</code> .                                                             |

### ***Conversion-State Functions***

```
int mbsinit(const mbstate_t *ps);
```

- |                |                                                                                                                                                                                                                                    |
|----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>mbsinit</i> | This group consists of a single function, <code>mbsinit</code> , which returns a nonzero value if <code>ps</code> is a null pointer or it points to an <code>mbstate_t</code> 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

&lt;ctype.h&gt; 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                                                          |
|----------------------------|---------------------------------------------------------------|
| <code>iswalnum(wc)</code>  | Is <code>wc</code> alphanumeric?                              |
| <code>iswalpha(wc)</code>  | Is <code>wc</code> alphabetic?                                |
| <code>iswblank(wc)</code>  | Is <code>wc</code> a blank? <sup>†</sup>                      |
| <code>iswcntrl(wc)</code>  | Is <code>wc</code> a control character?                       |
| <code>iswdigit(wc)</code>  | Is <code>wc</code> a decimal digit?                           |
| <code>iswgraph(wc)</code>  | Is <code>wc</code> a printing character (other than a space)? |
| <code>iswlower(wc)</code>  | Is <code>wc</code> a lower-case letter?                       |
| <code>iswprint(wc)</code>  | Is <code>wc</code> a printing character (including a space)?  |
| <code>iswpunct(wc)</code>  | Is <code>wc</code> punctuation?                               |
| <code>iswspace(wc)</code>  | Is <code>wc</code> a white-space character?                   |
| <code>iswupper(wc)</code>  | Is <code>wc</code> an upper-case letter?                      |
| <code>iswxdigit(wc)</code> | Is <code>wc</code> a hexadecimal digit?                       |

<sup>†</sup>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

***iswctype***

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.

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***  
***toupper***

The `towlower` and `toupper` 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);
```

***wctrans***

The `wctrans` and `towctrans` functions are used together to support generalized wide-character mapping.

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, ï, 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\xvu".
  - (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\xaa\x9j\xc3\xaa\x0 vu".

**Section 25.3**    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

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);
```

C99

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`.

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.

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).

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.)

*atof*  
*atoi*  
*atol*

*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 |        |                  |
|----------------------|--------------|--------|------------------|
| <code>atof</code>    | 0            |        |                  |
| <code>atoi</code>    | 0            |        |                  |
| <code>atol</code>    | 0            |        |                  |
| Function             | Return Value | Valid? | String Consumed? |
| <code>strtod</code>  | 0            | Yes    | No               |
| <code>strtol</code>  | 0            | Yes    | No               |
| <code>strtoul</code> | 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.

## 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`**Q&A**

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.

`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.

`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.

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

**Q&A**`signal function` ▶ 24.3`getenv`

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** 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);

```

|                                                                             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-----------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>ldiv</b><br><b>Q&amp;A</b><br><b>llabs</b><br><b>lldiv</b><br><b>C99</b> | <p>The <code>ldiv</code> function is similar but works with long integers; it returns an <code>ldiv_t</code> structure, which also has <code>quot</code> and <code>rem</code> members. (The <code>div_t</code> and <code>ldiv_t</code> types are declared in <code>&lt;stdlib.h&gt;</code>.)</p> <p>C99 provides two additional functions. The <code>llabs</code> function returns the absolute value of a <code>long long int</code> value. <code>lldiv</code> is similar to <code>div</code> and <code>ldiv</code>, except that it divides two <code>long long int</code> values and returns an <code>lldiv_t</code> structure. (The <code>lldiv_t</code> type was also added in C99.)</p> |
|-----------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## 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   |
|-----------------------|---------------------------|---------------|-----------------|
| <code>tm_sec</code>   | Seconds after the minute  | 0             | 61 <sup>†</sup> |
| <code>tm_min</code>   | Minutes after the hour    | 0             | 59              |
| <code>tm_hour</code>  | Hours since midnight      | 0             | 23              |
| <code>tm_mday</code>  | Day of the month          | 1             | 31              |
| <code>tm_mon</code>   | Months since January      | 0             | 11              |
| <code>tm_year</code>  | Years since 1900          | 0             | —               |
| <code>tm_wday</code>  | Days since Sunday         | 0             | 6               |
| <code>tm_yday</code>  | Days since January 1      | 0             | 365             |
| <code>tm_isdst</code> | Daylight Saving Time flag | ++            | ++              |

<sup>†</sup>Allows for two extra “leap seconds.” In C99, the maximum value is 60.

<sup>++</sup>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
```

**C99**

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.

**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 `mktimed` will bring the members of the structure back into their proper ranges:

```
mktimed(&t);
```

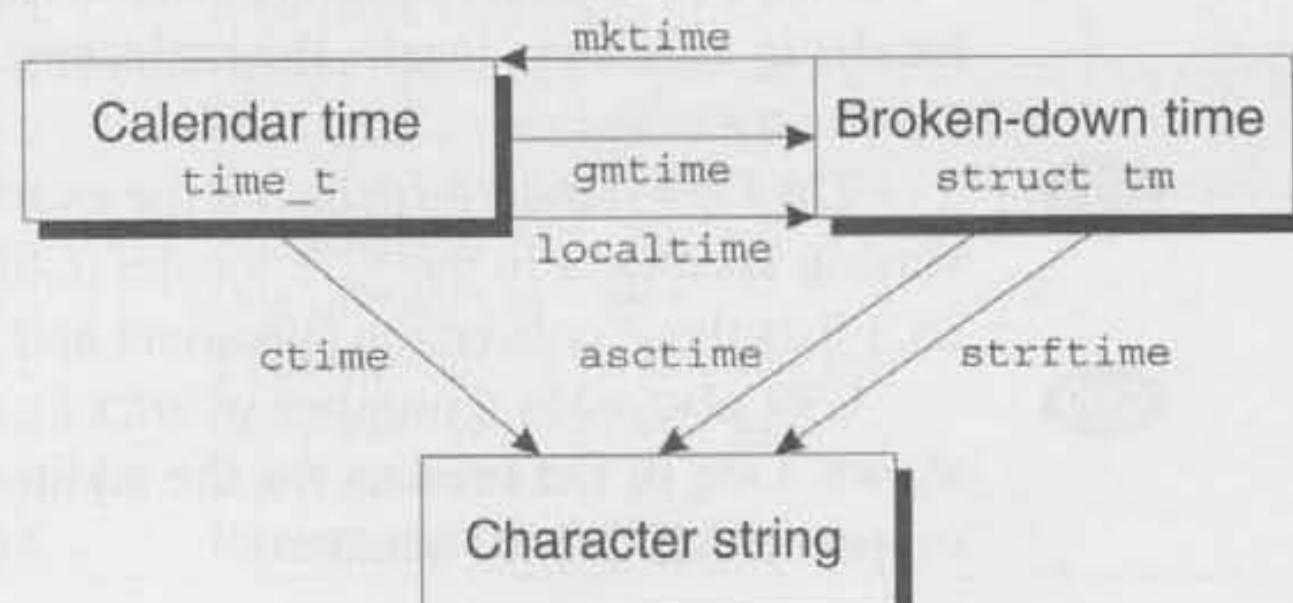
We'll discard `mktimed`'s return value, since we're interested only in the function's effect on `t`. The relevant members of `t` now have the following values:

| <i>Member</i>        | <i>Value</i> | <i>Meaning</i>        |
|----------------------|--------------|-----------------------|
| <code>tm_mday</code> | 12           | 12                    |
| <code>tm_mon</code>  | 7            | August                |
| <code>tm_year</code> | 112          | 2012                  |
| <code>tm_wday</code> | 0            | Sunday                |
| <code>tm_yday</code> | 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:



gmtime  
localtime

**Q&A**

asctime

ctime

strftime

sprintf function ➤ 22.8

locales ➤ 25.1

**C99**
**C99**

The figure includes the `mktime` function, which the C standard classifies as a “manipulation” function rather than a “conversion” function.

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.

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.

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.

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`.

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`.

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 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

| <i>Conversion</i> | <i>Replacement</i>                                         |
|-------------------|------------------------------------------------------------|
| %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 <sup>†</sup>   | Year divided by 100 and truncated to an integer (00–99)    |
| %d                | Day of month (01–31)                                       |
| %D <sup>†</sup>   | Equivalent to %m/%d/%y                                     |
| %e <sup>†</sup>   | Day of month (1–31); a single digit is preceded by a space |
| %F <sup>†</sup>   | Equivalent to %Y-%m-%d                                     |
| %g <sup>†</sup>   | Last two digits of ISO 8601 week-based year (00–99)        |
| %G <sup>†</sup>   | ISO 8601 week-based year                                   |
| %h <sup>†</sup>   | 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 <sup>†</sup>   | New-line character                                         |
| %p                | AM/PM designator (AM or PM)                                |
| %r <sup>†</sup>   | 12-hour clock time (e.g., 05:48:34 PM)                     |
| %R <sup>†</sup>   | Equivalent to %H:%M                                        |
| %S                | Second (00–61); maximum value in C99 is 60                 |
| %t <sup>†</sup>   | Horizontal-tab character                                   |
| %T <sup>†</sup>   | Equivalent to %H:%M:%S                                     |
| %u <sup>†</sup>   | ISO 8601 weekday (1–7); Monday is 1                        |
| %U                | Week number (00–53); first Sunday is beginning of week 1   |
| %V <sup>†</sup>   | 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 <sup>†</sup>   | Offset from UTC in ISO 8601 format (e.g., -0530 or +0200)  |
| %Z                | Time zone name or abbreviation (e.g., EST)                 |
| %%                | %                                                          |

<sup>†</sup>C99 only

**Table 26.3**

Replacement Strings for `strftime` Conversion Specifiers in the "C" Locale

| <i>Conversion</i> | <i>Replacement</i>                            |
|-------------------|-----------------------------------------------|
| %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 |    |    |    |    |    |    | Year | Week |
|--------------|----|----|----|----|----|----|------|------|
| Mo           | Tu | We | Th | Fr | Sa | Su |      |      |
|              |    |    |    |    | 1  | 2  | 2010 | 52   |
| 3            | 4  | 5  | 6  | 7  | 8  | 9  | 2011 | 1    |
| 10           | 11 | 12 | 13 | 14 | 15 | 16 | 2011 | 2    |
| 17           | 18 | 19 | 20 | 21 | 22 | 23 | 2011 | 3    |
| 24           | 25 | 26 | 27 | 28 | 29 | 30 | 2011 | 4    |
| 31           |    |    |    |    |    |    | 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)

| <i>Conversion</i> | <i>Replacement</i>                                                                                                                            |
|-------------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| <code>%Ec</code>  | Alternative date and time representation                                                                                                      |
| <code>%EC</code>  | Name of base year (period) in alternative representation                                                                                      |
| <code>%Ex</code>  | Alternative date representation                                                                                                               |
| <code>%EX</code>  | Alternative time representation                                                                                                               |
| <code>%Ey</code>  | Offset from <code>%EC</code> (year only) in alternative representation                                                                        |
| <code>%EY</code>  | Full alternative year representation                                                                                                          |
| <code>%Od</code>  | Day of month, using alternative numeric symbols (filled with leading zeros or with leading spaces if there is no alternative symbol for zero) |
| <code>%Oe</code>  | Day of month, using alternative numeric symbols (filled with leading spaces)                                                                  |
| <code>%OH</code>  | Hour on 24-hour clock, using alternative numeric symbols                                                                                      |
| <code>%OI</code>  | Hour on 12-hour clock, using alternative numeric symbols                                                                                      |
| <code>%Om</code>  | Month, using alternative numeric symbols                                                                                                      |
| <code>%OM</code>  | Minute, using alternative numeric symbols                                                                                                     |
| <code>%OS</code>  | Second, using alternative numeric symbols                                                                                                     |
| <code>%Ou</code>  | ISO 8601 weekday as a number in alternative representation, where Monday is 1                                                                 |
| <code>%OU</code>  | Week number, using alternative numeric symbols                                                                                                |
| <code>%OV</code>  | ISO 8601 week number, using alternative numeric symbols                                                                                       |
| <code>%Ow</code>  | Weekday as a number, using alternative numeric symbols                                                                                        |
| <code>%OW</code>  | Week number, using alternative numeric symbols                                                                                                |
| <code>%Oy</code>  | 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(¤t));

 /* Print date and time, using strftime to format */
 strftime(date_time, sizeof(date_time),
 "%m-%d-%Y %I:%M%p\n", localtime(¤t));
 puts(date_time);

 /* Print date and time, using printf to format */
 ptr = localtime(¤t);
 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:

```
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.

- \*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.

**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.

**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*  $\div$  *j* is rounded up or down is implementation-defined, as is the sign of *i*  $\bmod$  *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:

| <i>n</i> | <i>d</i> | <i>q</i> | <i>r</i> |
|----------|----------|----------|----------|
| 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

- 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 $0.0 \leq d < 1.0$.
10. Convert the following calls of `atoi`, `atol`, and `atoll` into calls of `strtol`, `strtold`, 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-Www-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-DDThh:mm:ss`
- Give `strftime` strings that correspond to each of these formats.

Programming Projects

- W 1. (a) Write a program that calls the `xrand` 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 `xrand`'s return value aren't especially random.)
 (b) How can we improve the randomness of `xrand` 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

27 Additional C99 Support for Mathematics

Simplicity does not precede complexity, but follows it.

This chapter completes our coverage of the standard library by describing five headers that are new in C99. These headers, like some of the older ones, provide support for working with numbers. However, the new headers are more specialized than the old ones. Some of them will appeal primarily to engineers, scientists, and mathematicians, who may need complex numbers as well as greater control over the representation of numbers and the way floating-point arithmetic is performed.

The first two sections discuss headers related to the integer types. The `<stdint.h>` header (Section 27.1) declares integer types that have a specified number of bits. The `<inttypes.h>` header (Section 27.2) provides macros that are useful for reading and writing values of the `<stdint.h>` types.

The next two sections describe C99's support for complex numbers. Section 27.3 includes a review of complex numbers as well as a discussion of C99's complex types. Section 27.4 then covers the `<complex.h>` header, which supplies functions that perform mathematical operations on complex numbers.

The headers discussed in the last two sections are related to the floating types. The `<tgmath.h>` header (Section 27.5) provides type-generic macros that make it easier to call library functions in `<complex.h>` and `<math.h>`. The functions in the `<fenv.h>` header (Section 27.6) give programs access to floating-point status flags and control modes.

27.1 The `<stdint.h>` Header (C99): Integer Types

The `<stdint.h>` header declares integer types containing a specified number of bits. In addition, it defines macros that represent the minimum and maximum values of these types as well as of integer types declared in other headers.

<limits.h> header ▶ 23.2

(These macros augment the ones in the <limits.h> header.) <stdint.h> also defines parameterized macros that construct integer constants with specific types. There are no functions in <stdint.h>.

The primary motivation for the <stdint.h> header lies in an observation made in Section 7.5, which discussed the role of type definitions in making programs portable. For example, if *i* is an *int* variable, the assignment

```
i = 100000;
```

is fine if *int* is a 32-bit type but will fail if *int* is a 16-bit type. The problem is that the C standard doesn't specify exactly how many bits an *int* value has. The standard *does* guarantee that the values of the *int* type must include all numbers between -32767 and +32767 (which requires at least 16 bits), but that's all it has to say on the matter. In the case of the variable *i*, which needs to be able to store 100000, the traditional solution is to declare *i* to be of some type *T*, where *T* is a type name created using *typedef*. The declaration of *T* can then be adjusted based on the sizes of integers in a particular implementation. (On a 16-bit machine, *T* would need to be *long int*, but on a 32-bit machine, it can be *int*.) This is the strategy that Section 7.5 discusses.

If your compiler supports C99, there's a better technique. The <stdint.h> header declares names for types based on the *width* of the type (the number of bits used to store values of the type, including the sign bit, if any). The *typedef* names declared in <stdint.h> may refer to basic types (such as *int*, *unsigned int*, and *long int*) or to extended integer types that are supported by a particular implementation.

<stdint.h> Types

The types declared in <stdint.h> fall into five groups:

- **Exact-width integer types.** Each name of the form *intN_t* represents a signed integer type with *N* bits, stored in two's-complement form. (Two's complement, a technique used to represent signed integers in binary, is nearly universal among modern computers.) For example, a value of type *int16_t* would be a 16-bit signed integer. A name of the form *uintN_t* represents an unsigned integer type with *N* bits. An implementation is required to provide both *intN_t* and *uintN_t* for *N* = 8, 16, 32, and 64 if it supports integers with these widths.
- **Minimum-width integer types.** Each name of the form *int_leastN_t* represents a signed integer type with at least *N* bits. A name of the form *uint_leastN_t* represents an unsigned integer type with *N* or more bits. <stdint.h> is required to provide at least the following minimum-width types:

<i>int_least8_t</i>	<i>uint_least8_t</i>
<i>int_least16_t</i>	<i>uint_least16_t</i>

```
int_least32_t    uint_least32_t
int_least64_t    uint_least64_t
```

- **Fastest minimum-width integer types.** Each name of the form `int_fastN_t` represents the fastest signed integer type with at least N bits. (The meaning of “fastest” is up to the implementation. If there’s no reason to classify a particular type as the fastest, the implementation may choose any signed integer type with at least N bits.) Each name of the form `uint_fastN_t` represents the fastest unsigned integer type with N or more bits. `<stdint.h>` is required to provide at least the following fastest minimum-width types:

```
int_fast8_t      uint_fast8_t
int_fast16_t     uint_fast16_t
int_fast32_t     uint_fast32_t
int_fast64_t     uint_fast64_t
```

- **Integer types capable of holding object pointers.** The `intptr_t` type represents a signed integer type that can safely store any `void *` value. More precisely, if a `void *` pointer is converted to `intptr_t` type and then back to `void *`, the resulting pointer and the original pointer will compare equal. The `uintptr_t` type is an unsigned integer type with the same property as `intptr_t`. The `<stdint.h>` header isn’t required to provide either type.
- **Greatest-width integer types.** `intmax_t` is a signed integer type that includes all values that belong to any signed integer type. `uintmax_t` is an unsigned integer type that includes all values that belong to any unsigned integer type. `<stdint.h>` is required to provide both types, which might be wider than `long long int`.

The names in the first three groups are declared using `typedef`.

An implementation may provide exact-width integer types, minimum-width integer types, and fastest minimum-width integer types for values of N in addition to the ones listed above. Also, N isn’t required to be a power of 2 (although it will normally be a multiple of 8). For example, an implementation might provide types named `int24_t` and `uint24_t`.

Limits of Specified-Width Integer Types

For each signed integer type declared in `<stdint.h>`, the header defines macros that specify the type’s minimum and maximum values. For each unsigned integer type, `<stdint.h>` defines a macro that specifies the type’s maximum value. The first three rows of Table 27.1 show the values of these macros for the exact-width integer types. The remaining rows show the constraints imposed by the C99 standard on the minimum and maximum values of the other `<stdint.h>` types. (The precise values of these macros are implementation-defined.) All macros in the table represent constant expressions.

Table 27.1

`<stdint.h>` Limit Macros for Specified-Width Integer Types

Name	Value	Description
<code>INTN_MIN</code>	$-(2^{N-1})$	Minimum <code>intN_t</code> value
<code>INTN_MAX</code>	$2^{N-1}-1$	Maximum <code>intN_t</code> value
<code>UINTN_MAX</code>	2^N-1	Maximum <code>uintN_t</code> value
<code>INT_LEASTN_MIN</code>	$\leq -(2^{N-1}-1)$	Minimum <code>int_leastN_t</code> value
<code>INT_LEASTN_MAX</code>	$\geq 2^{N-1}-1$	Maximum <code>int_leastN_t</code> value
<code>UINT_LEASTN_MAX</code>	$\geq 2^N-1$	Maximum <code>uint_leastN_t</code> value
<code>INT_FASTN_MIN</code>	$\leq -(2^{N-1}-1)$	Minimum <code>int_fastN_t</code> value
<code>INT_FASTN_MAX</code>	$\geq 2^{N-1}-1$	Maximum <code>int_fastN_t</code> value
<code>UINT_FASTN_MAX</code>	$\geq 2^N-1$	Maximum <code>uint_fastN_t</code> value
<code>INTPTR_MIN</code>	$\leq -(2^{15}-1)$	Minimum <code>intptr_t</code> value
<code>INTPTR_MAX</code>	$\geq 2^{15}-1$	Maximum <code>intptr_t</code> value
<code>UINTPTR_MAX</code>	$\geq 2^{16}-1$	Maximum <code>uintptr_t</code> value
<code>INTMAX_MIN</code>	$\leq -(2^{63}-1)$	Minimum <code>intmax_t</code> value
<code>INTMAX_MAX</code>	$\geq 2^{63}-1$	Maximum <code>intmax_t</code> value
<code>UINTMAX_MAX</code>	$\geq 2^{64}-1$	Maximum <code>uintmax_t</code> value

Limits of Other Integer Types

When the C99 committee created the `<stdint.h>` header, they decided that it would be a good place to put macros describing the limits of integer types besides the ones declared in `<stdint.h>` itself. These types are `ptrdiff_t`, `size_t`, and `wchar_t` (which belong to `<stddef.h>`), `sig_atomic_t` (declared in `<signal.h>`), and `wint_t` (declared in `<wchar.h>`). Table 27.2 lists these macros and shows the value of each (or any constraints on the value imposed by the C99 standard). In some cases, the constraints on the minimum and maximum values of a type depend on whether the type is signed or unsigned. The macros in Table 27.2, like the ones in Table 27.1, represent constant expressions.

Macros for Integer Constants

The `<stdint.h>` header also provides function-like macros that are able to convert an integer constant (expressed in decimal, octal, or hexadecimal, but without a U and/or L suffix) into a constant expression belonging to a minimum-width integer type or greatest-width integer type.

For each `int_leastN_t` type declared in `<stdint.h>`, the header defines a parameterized macro named `INTN_C` that converts an integer constant to this type (possibly using the integer promotions). For each `uint_leastN_t` type, there's a similar parameterized macro named `UINTN_C`. These macros are useful for initializing variables, among other things. For example, if `i` is a variable of type `int_least32_t`, writing

`<stddef.h>` header ▶ 21.4
`<signal.h>` header ▶ 24.3
`<wchar.h>` header ▶ 25.5

integer constants ▶ 7.1

integer promotions ▶ 7.4

Table 27.2

<stdint.h> Limit Macros for Other Integer Types

Name	Value	Description
PTRDIFF_MIN	≤ -65535	Minimum ptrdiff_t value
PTRDIFF_MAX	$\geq +65535$	Maximum ptrdiff_t value
SIG_ATOMIC_MIN	≤ -127 (if signed) 0 (if unsigned)	Minimum sig_atomic_t value
SIG_ATOMIC_MAX	$\geq +127$ (if signed) ≥ 255 (if unsigned)	Maximum sig_atomic_t value
SIZE_MAX	≥ 65535	Maximum size_t value
WCHAR_MIN	≤ -127 (if signed) 0 (if unsigned)	Minimum wchar_t value
WCHAR_MAX	$\geq +127$ (if signed) ≥ 255 (if unsigned)	Maximum wchar_t value
WINT_MIN	≤ -32767 (if signed) 0 (if unsigned)	Minimum wint_t value
WINT_MAX	$\geq +32767$ (if signed) ≥ 65535 (if unsigned)	Maximum wint_t value

```
i = 100000;
```

is problematic, because the constant 100000 might be too large to represent using type int (if int is a 16-bit type). However, the statement

```
i = INT32_C(100000);
```

is safe. If int_least32_t represents the int type, then INT32_C(100000) has type int. But if int_least32_t corresponds to long int, then INT32_C(100000) has type long int.

<stdint.h> has two other parameterized macros. INTMAX_C converts an integer constant to type intmax_t, and UINTMAX_C converts an integer constant to type uintmax_t.

27.2 The *<inttypes.h>* Header (C99) Format Conversion of Integer Types

Q&A

The *<inttypes.h>* header is closely related to the *<stdint.h>* header, the topic of Section 27.1. In fact, *<inttypes.h>* includes *<stdint.h>*, so programs that include *<inttypes.h>* don't need to include *<stdint.h>* as well. The *<inttypes.h>* header extends *<stdint.h>* in two ways. First, it defines macros that can be used in ...printf and ...scanf format strings for input/output of the integer types declared in *<stdint.h>*. Second, it provides functions for working with greatest-width integers.

Macros for Format Specifiers

The types declared in the `<stdint.h>` header can be used to make programs more portable, but they create new headaches for the programmer. Consider the problem of displaying the value of the variable `i`, where `i` has type `int_least32_t`. The statement

```
printf("i = %d\n", i);
```

may not work, because `i` doesn't necessarily have `int` type. If `int_least32_t` is another name for the `long int` type, then the correct conversion specification is `%ld`, not `%d`. In order to use the `...printf` and `...scanf` functions in a portable manner, we need a way to write conversion specifications that correspond to each of the types declared in `<stdint.h>`. That's where the `<inttypes.h>` header comes in. For each `<stdint.h>` type, `<inttypes.h>` provides a macro that expands into a string literal containing the proper conversion specifier for that type.

Each macro name has three parts:

- The name begins with either `PRI` or `SCN`, depending on whether the macro will be used in a call of a `...printf` function or a `...scanf` function.
- Next comes a one-letter conversion specifier (`d` or `i` for a signed type; `o`, `u`, `x`, or `X` for an unsigned type).
- The last part of the name indicates which `<stdint.h>` type is involved. For example, the name of a macro that corresponds to the `int_leastN_t` type would end with `LEASTN`.

Let's return to our previous example, which involved displaying an integer of type `int_least32_t`. Instead of using `d` as the conversion specifier, we'll switch to the `PRIldLEAST32` macro. To use the macro, we'll split the `printf` format string into three pieces and replace the `d` in `%d` by `PRIldLEAST32`:

```
printf("i = %" PRIldLEAST32 "\n", i);
```

The value of `PRIldLEAST32` is probably either `"d"` (if `int_least32_t` is the same as the `int` type) or `"ld"` (if `int_least32_t` is the same as `long int`). Let's assume that it's `"ld"` for the sake of discussion. After macro replacement, the statement becomes

```
printf("i = %" "ld" "\n", i);
```

Once the compiler joins the three string literals into one (which it will do automatically), the statement will have the following appearance:

```
printf("i = %ld\n", i);
```

Note that we can still include flags, a field width, and other options in our conversion specification; `PRIldLEAST32` supplies only the conversion specifier and possibly a length modifier, such as the letter `l`.

Table 27.3 lists the `<inttypes.h>` macros.

Table 27.3

Format-Specifier Macros
in <inttypes.h>

<i>...printf Macros for Signed Integers</i>				
PRIdN	PRIdLEASTN	PRIdFASTN	PRIdMAX	PRIdPTR
PRIiN	PRIiLEASTN	PRIiFASTN	PRIiMAX	PRIiPTR
<i>...printf Macros for Unsigned Integers</i>				
PRIoN	PRIoLEASTN	PRIoFASTN	PRIoMAX	PRIoPTR
PRIuN	PRIuLEASTN	PRIuFASTN	PRIuMAX	PRIuPTR
PRIxN	PRIxLEASTN	PRIxFASTN	PRIxMAX	PRIxPTR
PRIxN	PRIxLEASTN	PRIxFASTN	PRIxMAX	PRIxPTR
<i>...scanf Macros for Signed Integers</i>				
SCNdN	SCNdLEASTN	SCNdFASTN	SCNdMAX	SCNdPTR
SCNiN	SCNiLEASTN	SCNiFASTN	SCNiMAX	SCNiPTR
<i>...scanf Macros for Unsigned Integers</i>				
SCNoN	SCNoLEASTN	SCNoFASTN	SCNoMAX	SCNoPTR
SCNuN	SCNuLEASTN	SCNuFASTN	SCNuMAX	SCNuPTR
SCNxN	SCNxLEASTN	SCNxFASTN	SCNxMAX	SCNxPTR

Functions for Greatest-Width Integer Types

```
intmax_t imaxabs(intmax_t j);
imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
intmax_t strtointmax(const char * restrict nptr,
                      char ** restrict endptr,
                      int base);
uintmax_t strtoumax(const char * restrict nptr,
                     char ** restrict endptr,
                     int base);
intmax_t wcstoimmax(const wchar_t * restrict nptr,
                     wchar_t ** restrict endptr,
                     int base);
uintmax_t wcstoumax(const wchar_t * restrict nptr,
                     wchar_t ** restrict endptr,
                     int base);
```

In addition to defining macros, the <inttypes.h> header provides functions for working with greatest-width integers, which were introduced in Section 27.1. A greatest-width integer has type `intmax_t` (the widest signed integer type supported by an implementation) or `uintmax_t` (the widest unsigned integer type). These types might be the same width as the `long long int` type, but they could be wider. For example, `long long int` might be 64 bits wide and `intmax_t` and `uintmax_t` might be 128 bits wide.

The `imaxabs` and `imaxdiv` functions are greatest-width versions of the integer arithmetic functions declared in <stdlib.h>. The `imaxabs` function returns the absolute value of its argument. Both the argument and the return value have type `intmax_t`. The `imaxdiv` function divides its first argument by its

`imaxabs`
`imaxdiv`

<stdlib.h> header ➤ 26.2

second, returning an `imaxdiv_t` value. `imaxdiv_t` is a structure that contains both a quotient member (named `quot`) and a remainder member (`rem`); both members have type `intmax_t`.

`strtoimax`
`strtoumax`

The `strtoimax` and `strtoumax` functions are greatest-width versions of the numeric conversion functions of `<stdlib.h>`. The `strtoimax` function is the same as `strtol` and `strtoll`, except that it returns a value of type `intmax_t`. The `strtoumax` function is equivalent to `strtoul` and `strtoull`, except that it returns a value of type `uintmax_t`. Both `strtoimax` and `strtoumax` return zero if no conversion could be performed. Both functions store `ERANGE` in `errno` if a conversion produces a value that's outside the range of the function's return type. In addition, `strtoimax` returns the smallest or largest `intmax_t` value (`INTMAX_MIN` or `INTMAX_MAX`); `strtoumax` returns the largest `uintmax_t` value, `UINTMAX_MAX`.

`wcstoimax`
`wcstoumax`
`<wchar.h>` header ▶ 25.5

The `wcstoimax` and `wcstoumax` functions are greatest-width versions of the wide-string numeric conversion functions of `<wchar.h>`. The `wcstoimax` function is the same as `wcstol` and `wcstoll`, except that it returns a value of type `intmax_t`. The `wcstoumax` function is equivalent to `wcstoul` and `wcstoull`, except that it returns a value of type `uintmax_t`. Both `wcstoimax` and `wcstoumax` return zero if no conversion could be performed. Both functions store `ERANGE` in `errno` if a conversion produces a value that's outside the range of the function's return type. In addition, `wcstoimax` returns the smallest or largest `intmax_t` value (`INTMAX_MIN` or `INTMAX_MAX`); `wcstoumax` returns the largest `uintmax_t` value, `UINTMAX_MAX`.

27.3 Complex Numbers (C99)

Complex numbers are used in scientific and engineering applications as well as in mathematics. C99 provides several complex types, allows operators to have complex operands, and adds a header named `<complex.h>` to the standard library. There's a catch, though: complex numbers aren't supported by all implementations of C99. Section 14.3 discussed the difference between a *hosted* C99 implementation and a *freestanding* implementation. A hosted implementation must accept any program that conforms to the C99 standard, whereas a freestanding implementation doesn't have to compile programs that use complex types or standard headers other than `<float.h>`, `<iso646.h>`, `<limits.h>`, `<stdarg.h>`, `<stdbool.h>`, `<stddef.h>`, and `<stdint.h>`. Thus, a freestanding implementation may lack both complex types and the `<complex.h>` header.

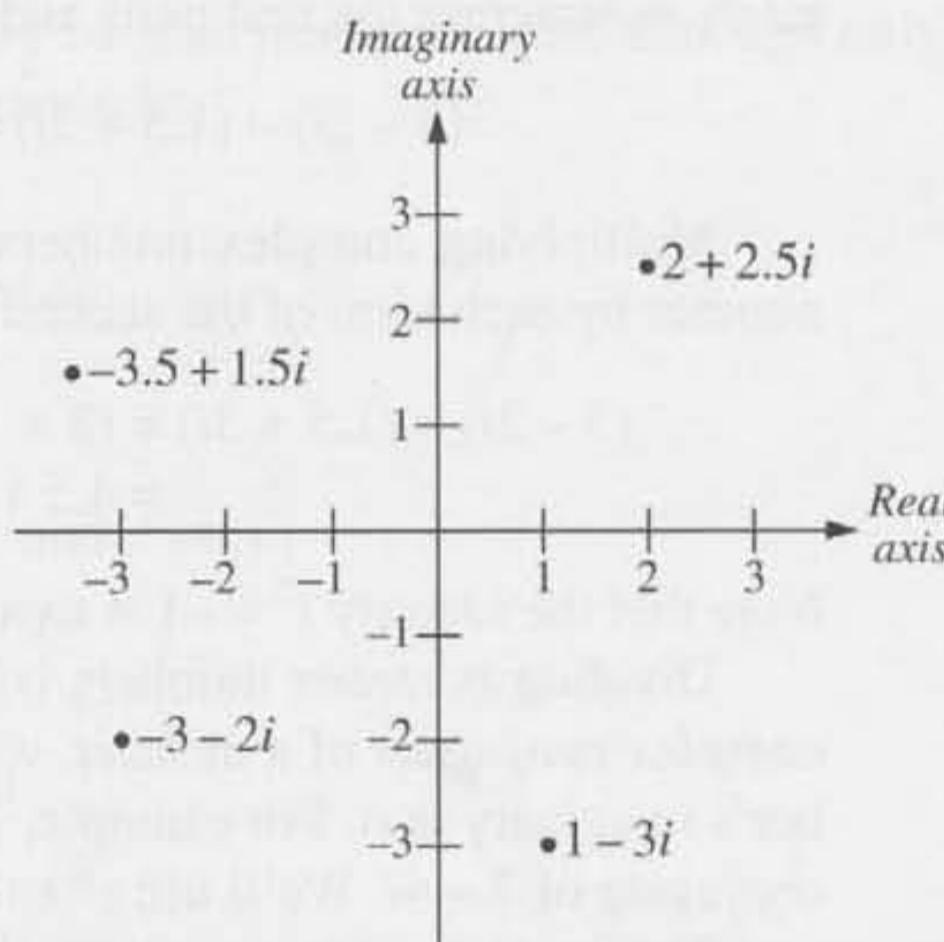
We'll start with a review of the mathematical definition of complex numbers and complex arithmetic. We'll then look at C99's complex types and the operations that can be performed on values of these types. Coverage of complex numbers continues in Section 27.4, which describes the `<complex.h>` header.

Definition of Complex Numbers

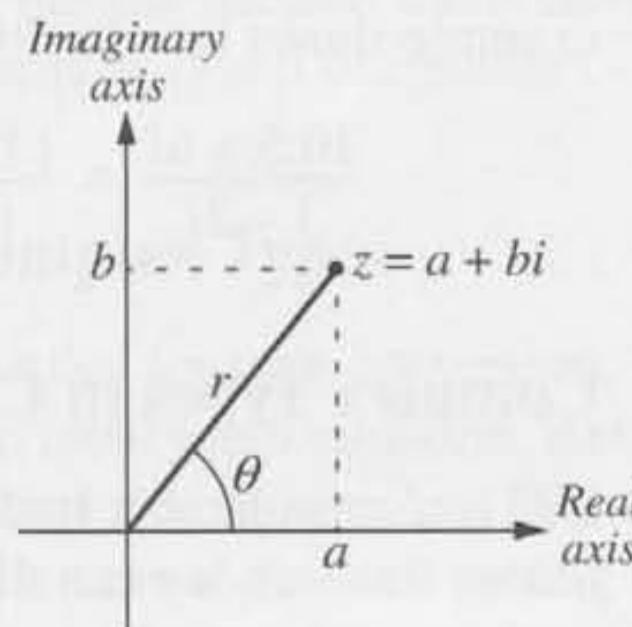
Let i be the square root of -1 (a number such that $i^2 = -1$). i is known as the **imaginary unit**; engineers often represent it by the symbol j instead of i . A **complex number** has the form $a + bi$, where a and b are real numbers. a is said to be the **real part** of the number, and b is the **imaginary part**. Note that the complex numbers include the real numbers as a special case (when $b = 0$).

Why are complex numbers useful? For one thing, they allow solutions to problems that are otherwise unsolvable. Consider the equation $x^2 + 1 = 0$, which has no solution if x is restricted to the real numbers. If complex numbers are allowed, there are two solutions: $x = i$ and $x = -i$.

Complex numbers can be thought of as points in a two-dimensional space known as the **complex plane**. Each complex number—a point in the complex plane—is represented by Cartesian coordinates, where the real part of the number corresponds to the x -coordinate of the point, and the imaginary part corresponds to the y -coordinate. For example, the complex numbers $2 + 2.5i$, $1 - 3i$, $-3 - 2i$, and $-3.5 + 1.5i$ can be plotted as follows:



An alternative system known as **polar coordinates** can also be used to specify a point on the complex plane. With polar coordinates, a complex number z is represented by the values r and θ , where r is the length of a line segment from the origin to z , and θ is the angle between this segment and the real axis:



r is called the **absolute value** of z . (The absolute value is also known as the *norm*, *modulus*, or *magnitude*.) θ is said to be the **argument** (or *phase angle*) of z . The absolute value of $a + bi$ is given by the following equation:

$$|a + bi| = \sqrt{a^2 + b^2}$$

For additional information about converting from Cartesian coordinates to polar coordinates and vice versa, see the Programming Projects at the end of the chapter.

Complex Arithmetic

The sum of two complex numbers is found by separately adding the real parts of the two numbers and the imaginary parts. For example,

$$(3 - 2i) + (1.5 + 3i) = (3 + 1.5) + (-2 + 3)i = 4.5 + i$$

The difference of two complex numbers is computed in a similar manner, by separately subtracting the real parts and the imaginary parts. For example,

$$(3 - 2i) - (1.5 + 3i) = (3 - 1.5) + (-2 - 3)i = 1.5 - 5i$$

Multiplying complex numbers is done by multiplying each term of the first number by each term of the second and then summing the products:

$$\begin{aligned} (3 - 2i) \times (1.5 + 3i) &= (3 \times 1.5) + (3 \times 3i) + (-2i \times 1.5) + (-2i \times 3i) \\ &= 4.5 + 9i - 3i - 6i^2 = 10.5 + 6i \end{aligned}$$

Note that the identity $i^2 = -1$ is used to simplify the result.

Dividing complex numbers is a bit harder. First, we need the concept of the **complex conjugate** of a number, which is found by switching the sign of the number's imaginary part. For example, $7 - 4i$ is the conjugate of $7 + 4i$, and $7 + 4i$ is the conjugate of $7 - 4i$. We'll use z^* to denote the conjugate of a complex number z .

The quotient of two complex numbers y and z is given by the formula

$$y/z = yz^*/zz^*$$

It turns out that zz^* is always a real number, so dividing zz^* into yz^* is easy (just divide both the real part and the imaginary part of yz^* separately). The following example shows how to divide $10.5 + 6i$ by $3 - 2i$:

$$\frac{10.5 + 6i}{3 - 2i} = \frac{(10.5 + 6i)(3 + 2i)}{(3 - 2i)(3 + 2i)} = \frac{19.5 + 39i}{13} = 1.5 + 3i$$

Complex Types in C99

C99 has considerable built-in support for complex numbers. Without including any library headers, we can declare variables that represent complex numbers and then perform arithmetic and other operations on these variables.

C99 provides three complex types, which were first introduced in Section 7.2: `float _Complex`, `double _Complex`, and `long double _Complex`. These types can be used in the same way as other types in C: to declare variables, parameters, return types, array elements, members of structures and unions, and so forth. For example, we could declare three variables as follows:

```
float _Complex x;
double _Complex y;
long double _Complex z;
```

Each of these variables is stored just like an array of two ordinary floating-point numbers. Thus, `y` is stored as two adjacent `double` values, with the first value containing the real part of `y` and the second containing the imaginary part.

C99 also allows implementations to provide imaginary types (the keyword `_Imaginary` is reserved for this purpose) but doesn't make this a requirement.

Operations on Complex Numbers

Complex numbers may be used in expressions, although only the following operators allow complex operands:

- Unary `+` and `-`
- Logical negation (`!`)
- `sizeof`
- Cast
- Multiplicative (`*` and `/` only)
- Additive (`+` and `-`)
- Equality (`==` and `!=`)
- Logical *and* (`&&`)
- Logical *or* (`||`)
- Conditional (`? :`)
- Simple assignment (`=`)
- Compound assignment (`*=`, `/=`, `+=`, and `-=` only)
- Comma (`,`)

Some notable omissions from the list include the relational operators (`<`, `<=`, `>`, and `>=`), along with the increment (`++`) and decrement (`--`) operators.

Conversion Rules for Complex Types

Section 7.4 described the C99 rules for type conversion, but without covering the complex types. It's now time to rectify that situation. Before we get to the conversion rules, though, we'll need some new terminology. For each floating type there is a *corresponding real type*. In the case of the real floating types (`float`, `double`, and `long double`), the corresponding real type is the same as the original type.

For the complex types, the corresponding real type is the original type without the word `_Complex`. (The corresponding real type for `float _Complex` is `float`, for example.)

We're now ready to discuss the general rules that govern type conversions involving complex types. I'll group them into three categories.

- **Complex to complex.** The first rule concerns conversions from one complex type to another, such as converting from `float _Complex` to `double _Complex`. In this situation, the real and imaginary parts are converted separately, using the rules for the corresponding real types (see Section 7.4). In our example, the real part of the `float _Complex` value would be converted to `double`, yielding the real part of the `double _Complex` value; the imaginary part would be converted to `double` in a similar fashion.
- **Real to complex.** When a value of a real type is converted to a complex type, the real part of the number is converted using the rules for converting from one real type to another. The imaginary part of the result is set to positive or unsigned zero.
- **Complex to real.** When a value of a complex type is converted to a real type, the imaginary part of the number is discarded; the real part is converted using the rules for converting from one real type to another.

One particular set of type conversions, known as the usual arithmetic conversions, are automatically applied to the operands of most binary operators. There are special rules for performing the usual arithmetic conversions when at least one of the two operands has a complex type:

1. If the corresponding real type of either operand is `long double`, convert the other operand so that its corresponding real type is `long double`.
2. Otherwise, if the corresponding real type of either operand is `double`, convert the other operand so that its corresponding real type is `double`.
3. Otherwise, one of the operands must have `float` as its corresponding real type. Convert the other operand so that its corresponding real type is also `float`.

A real operand still belongs to a real type after conversion, and a complex operand still belongs to a complex type.

Normally, the goal of the usual arithmetic conversions is to convert both operands to a common type. However, when a real operand is mixed with a complex operand, performing the usual arithmetic conversions causes the operands to have a common real type, but not necessarily the *same* type. For example, adding a `float` operand and a `double _Complex` operand causes the `float` operand to be converted to `double` rather than `double _Complex`. The type of the result will be the complex type whose corresponding real type matches the common real type. In our example, the type of the result will be `double _Complex`.

27.4 The `<complex.h>` Header (C99): Complex Arithmetic

As we saw in Section 27.3, C99 has significant built-in support for complex numbers. The `<complex.h>` header provides additional support in the form of mathematical functions on complex numbers, as well as some very useful macros and a pragma. Let's look at the macros first.

`<complex.h>` Macros

The `<complex.h>` header defines the macros shown in Table 27.4.

Table 27.4

`<complex.h>` Macros

Name	Value
<code>complex</code>	<code>_Complex</code>
<code>_Complex_I</code>	Imaginary unit; has type <code>const float _Complex</code>
<code>I</code>	<code>_Complex_I</code>

`complex` serves as an alternative name for the awkward `_Complex` keyword. We've seen a situation like this before with the Boolean type: the C99 committee chose a new keyword (`_Bool`) that shouldn't break existing programs, but provided a better name (`bool`) as a macro defined in the `<stdbool.h>` header. Programs that include `<complex.h>` may use `complex` instead of `_Complex`, just as programs that include `<stdbool.h>` may use `bool` rather than `_Bool`.

The `I` macro plays an important role in C99. There's no special language feature for creating a complex number from its real part and imaginary part. Instead, a complex number can be constructed by multiplying the imaginary part by `I` and adding the real part:

```
double complex dc = 2.0 + 3.5 * I;
```

The value of the variable `dc` is $2 + 3.5i$.

Note that both `_Complex_I` and `I` represent the imaginary unit i . Presumably most programmers will use `I` rather than `_Complex_I`. However, since `I` might already be used in existing code for some other purpose, `_Complex_I` is available as a backup. If the name `I` causes a conflict, it can always be undefined:

```
#include <complex.h>
#undef I
```

The programmer might then define a different—but still short—name for i , such as `J`:

```
#define J _Complex_I
```

Also note that the type of `_Complex_I` (and hence the type of `I`) is `float _Complex`, not `double _Complex`. When it's used in expressions, `I` will automatically be widened to `double _Complex` or `long double _Complex` if necessary.

The CX_LIMITED_RANGE Pragma

#pragma directive ▶ 14.5 The `<complex.h>` header provides a pragma named `CX_LIMITED_RANGE` that allows the compiler to use the following standard formulas for multiplication, division, and absolute value:

$$(a + bi) \times (c + di) = (ac - bd) + (bc + ad)i$$

$$(a + bi) / (c + di) = [(ac + bd) + (bc - ad)i] / (c^2 + d^2)$$

$$|a + bi| = \sqrt{a^2 + b^2}$$

Using these formulas may cause anomalous results in some cases because of overflow or underflow; moreover, the formulas don't handle infinities properly. Because of these potential problems, C99 doesn't use the formulas without the programmer's permission.

The `CX_LIMITED_RANGE` pragma has the following appearance:

```
#pragma STDC CX_LIMITED_RANGE on-off-switch
```

where *on-off-switch* is either `ON`, `OFF`, or `DEFAULT`. If the pragma is used with the value `ON`, it allows the compiler to use the formulas listed above. The value `OFF` causes the compiler to perform the calculations in a way that's safer but possibly slower. The default setting, indicated by the `DEFAULT` choice, is equivalent to `OFF`.

The duration of the `CX_LIMITED_RANGE` pragma depends on where it's used in a program. When it appears at the top level of a source file, outside any external declarations, it remains in effect until the next `CX_LIMITED_RANGE` pragma or the end of the file. The only other place that a `CX_LIMITED_RANGE` pragma might appear is at the beginning of a compound statement (possibly the body of a function); in that case, the pragma remains in effect until the next `CX_LIMITED_RANGE` pragma (even one inside a nested compound statement) or the end of the compound statement. At the end of a compound statement, the state of the switch returns to its value before the compound statement was entered.

`<complex.h>` Functions

The `<complex.h>` header provides functions similar to those in the C99 version of `<math.h>`. The `<complex.h>` functions are divided into groups, just as they were in `<math.h>`: trigonometric, hyperbolic, exponential and logarithmic, and power and absolute-value. The only functions that are unique to complex numbers are the manipulation functions, the last group discussed in this section.

Each `<complex.h>` function comes in three versions: a `float complex` version, a `double complex` version, and a `long double complex` version. The name of the `float complex` version ends with `f`, and the name of the `long double complex` version ends with `l`.

errno variable ➤ 24.2

Before we delve into the `<complex.h>` functions, a few general comments are in order. First, as with the `<math.h>` functions, the `<complex.h>` functions expect angle measurements to be in radians, not degrees. Second, when an error occurs, the `<complex.h>` functions may store a value in the `errno` variable, but aren't required to.

There's one last thing we'll need before tackling the `<complex.h>` functions. The term *branch cut* often appears in descriptions of functions that might conceivably have more than one possible return value. In the realm of complex numbers, choosing which value to return creates a branch cut: a curve (often just a line) in the complex plane around which a function is discontinuous. Branch cuts are usually not unique, but rather are determined by convention. An exact definition of branch cuts takes us further into complex analysis than I'd like to go, so I'll simply reproduce the restrictions from the C99 standard without further explanation.

Trigonometric Functions

```
double complex cacos(double complex z);
float complex cacosf(float complex z);
long double complex cacosl(long double complex z);

double complex casin(double complex z);
float complex casinf(float complex z);
long double complex casinl(long double complex z);

double complex catan(double complex z);
float complex catanf(float complex z);
long double complex catanl(long double complex z);

double complex ccos(double complex z);
float complex ccosf(float complex z);
long double complex ccosl(long double complex z);

double complex csin(double complex z);
float complex csinf(float complex z);
long double complex csinl(long double complex z);

double complex ctan(double complex z);
float complex ctanf(float complex z);
long double complex ctanl(long double complex z);
```

- cacos** The `cacos` function computes the complex arc cosine, with branch cuts outside the interval $[-1, +1]$ along the real axis. The return value lies in a strip mathematically unbounded along the imaginary axis and in the interval $[0, \pi]$ along the real axis.

casin

The *casin* function computes the complex arc sine, with branch cuts outside the interval $[-1, +1]$ along the real axis. The return value lies in a strip mathematically unbounded along the imaginary axis and in the interval $[-\pi/2, +\pi/2]$ along the real axis.

catan

The *catan* function computes the complex arc tangent, with branch cuts outside the interval $[-i, +i]$ along the imaginary axis. The return value lies in a strip mathematically unbounded along the imaginary axis and in the interval $[-\pi/2, +\pi/2]$ along the real axis.

ccos

The *ccos* function computes the complex cosine, the *csin* function computes the complex sine, and the *ctan* function computes the complex tangent.

*csin**ctan*

Hyperbolic Functions

```
double complex cacosh(double complex z);
float complex cacoshf(float complex z);
long double complex cacoshl(long double complex z);

double complex casinh(double complex z);
float complex casinhf(float complex z);
long double complex casinhl(long double complex z);

double complex catanh(double complex z);
float complex catanhf(float complex z);
long double complex catanhl(long double complex z);

double complex ccosh(double complex z);
float complex ccoshf(float complex z);
long double complex ccoshl(long double complex z);

double complex csinh(double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);

double complex ctanh(double complex z);
float complex ctanhf(float complex z);
long double complex ctanhl(long double complex z);
```

cacosh

The *cacosh* function computes the complex arc hyperbolic cosine, with a branch cut at values less than 1 along the real axis. The return value lies in a half-strip of nonnegative values along the real axis and in the interval $[-i\pi, +i\pi]$ along the imaginary axis.

casinh

The *casinh* function computes the complex arc hyperbolic sine, with branch cuts outside the interval $[-i, +i]$ along the imaginary axis. The return value lies in a strip mathematically unbounded along the real axis and in the interval $[-i\pi/2, +i\pi/2]$ along the imaginary axis.

catanh

The *catanh* function computes the complex arc hyperbolic tangent, with branch cuts outside the interval $[-1, +1]$ along the real axis. The return value lies in

a strip mathematically unbounded along the real axis and in the interval $[-i\pi/2, +i\pi/2]$ along the imaginary axis.

*ccosh**csinh**ctanh*

The `ccosh` function computes the complex hyperbolic cosine, the `csinh` function computes the complex hyperbolic sine, and the `ctanh` function computes the complex hyperbolic tangent.

Exponential and Logarithmic Functions

```
double complex cexp(double complex z);
float complex cexpf(float complex z);
long double complex cexpl(long double complex z);

double complex clog(double complex z);
float complex clogf(float complex z);
long double complex clogl(long double complex z);
```

*cexp**clog*

The `cexp` function computes the complex base-*e* exponential value.

The `clog` function computes the complex natural (base-*e*) logarithm, with a branch cut along the negative real axis. The return value lies in a strip mathematically unbounded along the real axis and in the interval $[-i\pi, +i\pi]$ along the imaginary axis.

Power and Absolute-Value Functions

```
double cabs(double complex z);
float cabsf(float complex z);
long double cabsl(long double complex z);

double complex cpow(double complex x,
                     double complex y);
float complex cpowf(float complex x,
                     float complex y);
long double complex cpowl(long double complex x,
                           long double complex y);

double complex csqrt(double complex z);
float complex csqrtf(float complex z);
long double complex csqrtl(long double complex z);
```

*cabs**cpow**csqrt*

The `cabs` function computes the complex absolute value.

The `cpow` function returns *x* raised to the power *y*, with a branch cut for the first parameter along the negative real axis.

The `csqrt` function computes the complex square root, with a branch cut along the negative real axis. The return value lies in the right half-plane (including the imaginary axis).

Manipulation Functions

```
double carg(double complex z);
float cargf(float complex z);
long double cargl(long double complex z);

double cimag(double complex z);
float cimagf(float complex z);
long double cimaml(long double complex z);

double complex conj(double complex z);
float complex conjf(float complex z);
long double complex conjl(long double complex z);

double complex cproj(double complex z);
float complex cprojf(float complex z);
long double complex cprojl(long double complex z);

double creal(double complex z);
float crealf(float complex z);
long double creall(long double complex z);
```

- carg* The `carg` function returns the argument (phase angle) of z , with a branch cut along the negative real axis. The return value lies in the interval $[-\pi, +\pi]$.
- cimag* The `cimag` function returns the imaginary part of z .
- conj* The `conj` function returns the complex conjugate of z .
- cproj* The `cproj` function computes a projection of z onto the Riemann sphere. The return value is equal to z unless one of its parts is infinite, in which case `cproj` returns `INFINITY + I * copysign(0.0, cimag(z))`.
- creal* The `creal` function returns the real part of z .

PROGRAM Finding the Roots of a Quadratic Equation

The roots of the quadratic equation

$$ax^2 + bx + c = 0$$

are given by the *quadratic formula*:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

In general, the value of x will be a complex number, because the square root of $b^2 - 4ac$ is imaginary if $b^2 - 4ac$ (known as the *discriminant*) is less than 0.

For example, suppose that $a = 5$, $b = 2$, and $c = 1$, which gives us the equation

$$5x^2 + 2x + 1 = 0$$

The value of the discriminant is $4 - 20 = -16$, so the roots of the equation will be

complex numbers. The following program, which uses several `<complex.h>` functions, computes and displays the roots.

```
quadratic.c /* Finds the roots of the equation 5x**2 + 2x + 1 = 0 */

#include <complex.h>
#include <stdio.h>

int main(void)
{
    double a = 5, b = 2, c = 1;
    double complex discriminant_sqrt = csqrt(b * b - 4 * a * c);
    double complex root1 = (-b + discriminant_sqrt) / (2 * a);
    double complex root2 = (-b - discriminant_sqrt) / (2 * a);

    printf("root1 = %g + %gi\n", creal(root1), cimag(root1));
    printf("root2 = %g + %gi\n", creal(root2), cimag(root2));

    return 0;
}
```

Here's the output of the program:

```
root1 = -0.2 + 0.4i
root2 = -0.2 + -0.4i
```

The `quadratic.c` program shows how to display a complex number by extracting the real and imaginary parts and then writing each as a floating-point number. `printf` lacks conversion specifiers for complex numbers, so there's no easier technique. There's also no shortcut for reading complex numbers; a program will need to obtain the real and imaginary parts separately and then combine them into a single complex number.

27.5 The `<tgmath.h>` Header (C99): Type-Generic Math

The `<tgmath.h>` header provides parameterized macros with names that match functions in `<math.h>` and `<complex.h>`. These *type-generic macros* can detect the types of the arguments passed to them and substitute a call of the appropriate version of a `<math.h>` or `<complex.h>` function.

In C99, there are multiple versions of many math functions, as we saw in Sections 23.3, 23.4, and 27.4. For example, the `sqrt` function comes in a `double` version (`sqrt`), a `float` version (`sqrtf`), and a `long double` version (`sqrtl`), as well as three versions for complex numbers (`csqrt`, `csqrif`, and `csqrifl`). By using `<tgmath.h>`, the programmer can simply invoke `sqrt` without having to worry about which version is needed: the call `sqrt(x)` could be a call of any of the six versions of `sqrt`, depending on the type of `x`.

One advantage of using `<tgmath.h>` is that calls of math functions become easier to write (and read!). More importantly, a call of a type-generic macro won't have to be modified in the future should the type of its argument(s) change.

The `<tgmath.h>` header includes both `<math.h>` and `<complex.h>`, by the way, so including `<tgmath.h>` provides access to the functions in both headers.

Type-Generic Macros

The type-generic macros defined in the `<tgmath.h>` header fall into three groups, depending on whether they correspond to functions in `<math.h>`, `<complex.h>`, or both headers.

Table 27.5 lists the type-generic macros that correspond to functions in both `<math.h>` and `<complex.h>`. Note that the name of each type-generic macro matches the name of the “unsuffixed” `<math.h>` function (`acos` as opposed to `acosf` or `acosl`, for example).

Table 27.5

Type-Generic Macros in `<tgmath.h>` (Group 1)

<code><math.h></code> <i>Function</i>	<code><complex.h></code> <i>Function</i>	<i>Type-Generic Macro</i>
<code>acos</code>	<code>cacos</code>	<code>acos</code>
<code>asin</code>	<code>casin</code>	<code>asin</code>
<code>atan</code>	<code>catan</code>	<code>atan</code>
<code>acosh</code>	<code>cacosh</code>	<code>acosh</code>
<code>asinh</code>	<code>casinh</code>	<code>asinh</code>
<code>atanh</code>	<code>catanh</code>	<code>atanh</code>
<code>cos</code>	<code>ccos</code>	<code>cos</code>
<code>sin</code>	<code>csin</code>	<code>sin</code>
<code>tan</code>	<code>ctan</code>	<code>tan</code>
<code>cosh</code>	<code>ccosh</code>	<code>cosh</code>
<code>sinh</code>	<code>csinh</code>	<code>sinh</code>
<code>tanh</code>	<code>ctanh</code>	<code>tanh</code>
<code>exp</code>	<code>cexp</code>	<code>exp</code>
<code>log</code>	<code>clog</code>	<code>log</code>
<code>pow</code>	<code>cpow</code>	<code>pow</code>
<code>sqrt</code>	<code>csqrt</code>	<code>sqrt</code>
<code>fabs</code>	<code>cabs</code>	<code>fabs</code>

The macros in the second group (Table 27.6) correspond only to functions in `<math.h>`. Each macro has the same name as the unsuffixed `<math.h>` function. Passing a complex argument to any of these macros causes undefined behavior.

Table 27.6

Type-Generic Macros in `<tgmath.h>` (Group 2)

<code>atan2</code>	<code>fma</code>	<code>llround</code>	<code>remainder</code>
<code>cbrt</code>	<code>fmax</code>	<code>log10</code>	<code>remquo</code>
<code>ceil</code>	<code>fmin</code>	<code>log1p</code>	<code>rint</code>
<code>copysign</code>	<code>fmod</code>	<code>log2</code>	<code>round</code>
<code>erf</code>	<code>frexp</code>	<code>logb</code>	<code>scalbn</code>
<code>erfc</code>	<code>hypot</code>	<code>lrint</code>	<code>scalbln</code>
<code>exp2</code>	<code>ilogb</code>	<code>lround</code>	<code>tgamma</code>
<code>expm1</code>	<code>ldexp</code>	<code>nearbyint</code>	<code>trunc</code>
<code>fdim</code>	<code>lgamma</code>	<code>nextafter</code>	
<code>floor</code>	<code>llrint</code>	<code>nexttoward</code>	

The macros in the final group (Table 27.7) correspond only to functions in `<complex.h>`.

Table 27.7

Type-Generic Macros in
`<tgmath.h>` (Group 3)

carg	conj	creal
cimag	cproj	

Q&A

Between the three tables, all functions in `<math.h>` and `<complex.h>` that have multiple versions are accounted for, with the exception of `modf`.

Invoking a Type-Generic Macro

To understand what happens when a type-generic macro is invoked, we first need the concept of a *generic parameter*. Consider the prototypes for the three versions of the `nextafter` function (from `<math.h>`):

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

The types of both `x` and `y` change depending on the version of `nextafter`, so both parameters are generic. Now consider the prototypes for the three versions of the `nexttoward` function:

```
double nexttoward(double x, long double y);
float nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);
```

The first parameter is generic, but the second is not (it always has type `long double`). Generic parameters always have type `double` (or `double complex`) in the unsuffixed version of a function.

When a type-generic macro is invoked, the first step is to determine whether it should be replaced by a `<math.h>` function or a `<complex.h>` function. (This step doesn't apply to the macros in Table 27.6, which are always replaced by a `<math.h>` function, or the macros in Table 27.7, which are always replaced by a `<complex.h>` function.) The rule is simple: if any argument corresponding to a generic parameter is complex, then a `<complex.h>` function is chosen; otherwise, a `<math.h>` function is selected.

The next step is to deduce which version of the `<math.h>` function or `<complex.h>` function is being called. Let's assume that the function being called belongs to `<math.h>`. (The rules for the `<complex.h>` case are analogous.) The following rules are used, in the order listed:

1. If any argument corresponding to a generic parameter has type `long double`, the `long double` version of the function is called.
2. If any argument corresponding to a generic parameter has type `double` or any integer type, the `double` version of the function is called.
3. Otherwise, the `float` version of the function is called.

Rule 2 is a little unusual: it states that an integer argument causes the `double` version of a function to be called, not the `float` version, which you might expect.

Q&A

As an example, assume that the following variables have been declared:

```
int i;
float f;
double d;
long double ld;
float complex fc;
double complex dc;
long double complex ldc;
```

For each macro invocation in the left column below, the corresponding function call appears in the right column:

<i>Macro Invocation</i>	<i>Equivalent Function Call</i>
<code>sqrt(i)</code>	<code>sqrt(i)</code>
<code>sqrt(f)</code>	<code>sqrtf(f)</code>
<code>sqrt(d)</code>	<code>sqrt(d)</code>
<code>sqrt(ld)</code>	<code>sqrtl(ld)</code>
<code>sqrt(fc)</code>	<code>csqrtf(fc)</code>
<code>sqrt(dc)</code>	<code>csqrt(dc)</code>
<code>sqrt(ldc)</code>	<code>csqrto(ldc)</code>

Note that writing `sqrt(i)` causes the double version of `sqrt` to be called, not the `float` version.

These rules also cover macros with more than one parameter. For example, the macro invocation `pow(1d, f)` will be replaced by the call `powl(1d, f)`. Both of `pow`'s parameters are generic; because one of the arguments has type `long double`, rule 1 states that the `long double` version of `pow` will be called.

27.6 The `<fenv.h>` Header (C99): Floating-Point Environment

IEEE Standard 754 is the most widely used representation for floating-point numbers. (This standard is also known as IEC 60559, which is how the C99 standard refers to it.) The purpose of the `<fenv.h>` header is to give programs access to the floating-point status flags and control modes specified in the IEEE standard. Although `<fenv.h>` was designed in a general fashion that allows it to work with other floating-point representations, supporting the IEEE standard was the reason for the header's creation.

A discussion of why programs might need access to status flags and control modes is beyond the scope of this book. For good examples, see “What every computer scientist should know about floating-point arithmetic” by David Goldberg (*ACM Computing Surveys*, vol. 23, no. 1 (March 1991): 5–48), which can be found on the Web.

Floating-Point Status Flags and Control Modes

Section 7.2 discussed some of the basic properties of IEEE Standard 754. Section 23.4, which covered the C99 additions to the `<math.h>` header, gave additional detail. Some of that discussion, particularly concerning exceptions and rounding directions, is directly relevant to the `<fenv.h>` header. Before we continue, let's review some of the material from Section 23.4 as well as define a few new terms.

A *floating-point status flag* is a system variable that's set when a floating-point exception is raised. In the IEEE standard, 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). Each exception has a corresponding status flag.

The `<fenv.h>` header declares a type named `fexcept_t` that's used for working with the floating-point status flags. An `fexcept_t` object represents the collective value of these flags. Although `fexcept_t` can simply be an integer type, with single bits representing individual flags, the C99 standard doesn't make this a requirement. Other alternatives exist, including the possibility that `fexcept_t` is a structure, with one member for each exception. This member could store additional information about the corresponding exception, such as the address of the floating-point instruction that caused the exception to be raised.

A *floating-point control mode* is a system variable that may be set by a program to change the future behavior of floating-point arithmetic. The IEEE standard requires a “directed-rounding” mode that controls the rounding direction when a number can't be represented exactly using a floating-point representation. There are four rounding directions: (1) *Round toward nearest*. Rounds to the nearest representable value. If a number falls halfway between two values, it's 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. Some implementations of the IEEE standard provide two additional control modes: a mode that controls rounding precision and a “trap enablement” mode that determines whether a floating-point processor will trap (or stop) when an exception is raised.

The term *floating-point environment* refers to the combination of floating-point status flags and control modes supported by a particular implementation. A value of type `fenv_t` represents an entire floating-point environment. The `fenv_t` type, like the `fexcept_t` type, is declared in `<fenv.h>`.

`<fenv.h>` Macros

The `<fenv.h>` header potentially defines the macros listed in Table 27.8. Only two of these macros (`FE_ALL_EXCEPT` and `FE_DFL_ENV`) are required, however. An implementation may define additional macros not listed in the table; the names of these macros must begin with `FE_` and an uppercase letter.

Table 27.8
<fenv.h> Macros

Name	Value	Description
FE_DIVBYZERO	Integer constant	Defined only if the corresponding floating-point exception is supported by the implementation. An implementation may define additional macros that represent floating-point exceptions.
FE_INEXACT	expressions whose	
FE_INVALID	bits do not overlap	
FE_OVERFLOW		
FE_UNDERFLOW		
FE_ALL_EXCEPT	See description	Bitwise <i>or</i> of all floating-point exception macros defined by the implementation. Has the value 0 if no such macros are defined.
FE_DOWNWARD	Integer constant	Defined only if the corresponding rounding direction can be retrieved and set via the <code>fgetround</code> and <code>fesetround</code> functions. An implementation may define additional macros that represent rounding directions.
FE_TONEAREST	expressions with	
FE_TOWARDZERO	distinct nonnegative values	
FE_UPWARD		
FE_DFL_ENV	A value of type <code>const fenv_t *</code>	Represents the default (program start-up) floating-point environment. An implementation may define additional macros that represent floating-point environments.

The FENV_ACCESS Pragma

#pragma directive ▶ 14.5

The `<fenv.h>` header provides a pragma named `FENV_ACCESS` that's used to notify the compiler of a program's intention to use the functions provided by this header. Knowing which portions of a program will use the capabilities of `<fenv.h>` is important for the compiler, because some common optimizations can't be performed if control modes don't have their customary settings or may change during program execution.

The `FENV_ACCESS` pragma has the following appearance:

```
#pragma STDC FENV_ACCESS on-off-switch
```

where *on-off-switch* is either `ON`, `OFF`, or `DEFAULT`. If the pragma is used with the value `ON`, it informs the compiler that the program might test floating-point status flags or alter a floating-point control mode. The value `OFF` indicates that flags won't be tested and default control modes are in effect. The meaning of `DEFAULT` is implementation-defined; it represents either `ON` or `OFF`.

The duration of the `FENV_ACCESS` pragma depends on where it's used in a program. When it appears at the top level of a source file, outside any external declarations, it remains in effect until the next `FENV_ACCESS` pragma or the end of the file. The only other place that an `FENV_ACCESS` pragma might appear is at the beginning of a compound statement (possibly the body of a function); in that case, the pragma remains in effect until the next `FENV_ACCESS` pragma (even one inside a nested compound statement) or the end of the compound statement. At the end of a compound statement, the state of the switch returns to its value before the compound statement was entered.

It's the programmer's responsibility to use the `FENV_ACCESS` pragma to indicate regions of a program in which low-level access to floating-point hardware

is needed. Undefined behavior occurs if a program tests floating-point status flags or runs under non-default control modes in a region for which the value of the pragma switch is OFF.

Typically, an FENV_ACCESS pragma that specifies the ON switch would be placed at the beginning of a function body:

```
void f(double x, double y)
{
    #pragma STDC FENV_ACCESS ON
    ...
}
```

The function *f* may test floating-point status flags or change control modes as needed. At the end of *f*'s body, the pragma switch will return to its previous state.

When a program goes from an FENV_ACCESS “off” region to an “on” region during execution, the floating-point status flags have unspecified values and the control modes have their default settings.

Floating-Point Exception Functions

```
int feclearexcept(int excepts);
int fegetexceptflag(fexcept_t *flagp, int excepts);
int feraiseexcept(int excepts);
int fesetexceptflag(const fexcept_t *flagp,
                    int excepts);
int fetestexcept(int excepts);
```

The *<fenv.h>* functions are divided into three groups. Functions in the first group deal with the floating-point status flags. Each of the five functions has an *int* parameter named *excepts*, which is the bitwise *or* of one or more of the floating-point exception macros (the first group of macros listed in Table 27.8). For example, the argument passed to one of these functions might be *FE_INVALID | FE_OVERFLOW | FE_UNDERFLOW*, to represent the combination of these three status flags. The argument may also be zero, to indicate that no flags are selected.

The *feclearexcept* function attempts to clear the floating-point exceptions represented by *excepts*. It returns zero if *excepts* is zero or if all specified exceptions were successfully cleared; otherwise, it returns a nonzero value.

The *fegetexceptflag* function attempts to retrieve the states of the floating-point status flags represented by *excepts*. This data is stored in the *fexcept_t* object pointed to by *flagp*. The *fegetexceptflag* function returns zero if the states of the status flags were successfully stored; otherwise, it returns a nonzero value.

The *feraiseexcept* function attempts to raise supported floating-point exceptions represented by *excepts*. It is implementation-defined whether *feraiseexcept* also raises the *inexact* floating-point exception whenever it

feclearexcept

fegetexceptflag

feraiseexcept

raises the *overflow* or *underflow* exception. (Implementations that conform to the IEEE standard will have this property.) `fraiseexcept` returns zero if `excepts` is zero or if all specified exceptions were successfully raised; otherwise, it returns a nonzero value.

`fesetexceptflag`

The `fesetexceptflag` function attempts to set the floating-point status flags represented by `excepts`. The states of the flags are stored in the `fexcept_t` object pointed to by `flagp`; this object must have been set by a previous call of `fegetexceptflag`. Moreover, the second argument in the prior call of `fegetexceptflag` must have included all floating-point exceptions represented by `excepts`. The `fesetexceptflag` function returns zero if `excepts` is zero or if all specified exceptions were successfully set; otherwise, it returns a nonzero value.

`fetestexcept`

The `fetestexcept` function tests only those floating-point status flags represented by `excepts`. It returns the bitwise *or* of the floating-point exception macros corresponding to the flags that are currently set. For example, if the value of `excepts` is `FE_INVALID | FE_OVERFLOW | FE_UNDERFLOW`, the `fetestexcept` function might return `FE_INVALID | FE_UNDERFLOW`, indicating that, of the exceptions represented by `FE_INVALID`, `FE_OVERFLOW`, and `FE_UNDERFLOW`, only the flags for `FE_INVALID` and `FE_UNDERFLOW` are currently set.

Rounding Functions

```
int fegetround(void);
int fesetround(int round);
```

The `fegetround` and `fesetround` functions are used to determine the rounding direction and modify it. Both functions rely on the rounding-direction macros (the third group in Table 27.8).

`fegetround`

The `fegetround` function returns the value of the rounding-direction macro that matches the current rounding direction. If the current rounding direction can't be determined or doesn't match any rounding-direction macro, `fegetround` returns a negative number.

`fesetround`

When passed the value of a rounding-direction macro, the `fesetround` function attempts to establish the corresponding rounding direction. If the call is successful, `fesetround` returns zero; otherwise, it returns a nonzero value.

Environment Functions

```
int fegetenv(fenv_t *envp);
int feholdexcept(fenv_t *envp);
int fesetenv(const fenv_t *envp);
int feupdateenv(const fenv_t *envp);
```

The last four functions in `<fenv.h>` deal with the entire floating-point environment, not just the status flags or control modes. Each function returns zero if it succeeds at the operation it was asked to perform. Otherwise, it returns a nonzero value.

`fegetenv`

The `fegetenv` function attempts to retrieve the current floating-point environment from the processor and store it in the object pointed to by `envp`.

`feholdexcept`

The `feholdexcept` function (1) stores the current floating-point environment in the object pointed to by `envp`, (2) clears the floating-point status flags, and (3) attempts to install a non-stop mode—if available—for all floating-point exceptions (so that future exceptions won’t cause a trap or stop).

`fesetenv`

The `fesetenv` function attempts to establish the floating-point environment represented by `envp`, which either points to a floating-point environment stored by a previous call of `fegetenv` or `feholdexcept`, or is equal to a floating-point environment macro such as `FE_DFL_ENV`. Unlike the `feupdateenv` function, `fesetenv` doesn’t raise any exceptions. If a call of `fegetenv` is used to save the current floating-point environment, then a later call of `fesetenv` can restore the environment to its previous state.

`feupdateenv`

The `feupdateenv` function attempts to (1) save the currently raised floating-point exceptions, (2) install the floating-point environment pointed to by `envp`, and (3) raise the saved exceptions. `envp` either points to a floating-point environment stored by a previous call of `fegetenv` or `feholdexcept`, or is equal to a floating-point environment macro such as `FE_DFL_ENV`.

Q & A

Q: If the `<inttypes.h>` header includes the `<stdint.h>` header, why do we need the `<stdint.h>` header at all? [p. 709]

A: The primary reason that `<stdint.h>` exists as a separate header is so that programs in a freestanding implementation may include it. (C99 requires conforming implementations—both hosted and freestanding—to provide the `<stdint.h>` header, but `<inttypes.h>` is required only for hosted implementations.) Even in a hosted environment, it may be advantageous to include `<stdint.h>` rather than `<inttypes.h>` to avoid defining all the macros that belong to the latter.

***Q:** There are three versions of the `modf` function in `<math.h>`, so why isn’t there a type-generic macro named `modf`? [p. 725]

A: Let’s take a look at the prototypes for the three versions of `modf`:

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

`modf` is unusual in that it has a pointer parameter, and the type of the pointer isn’t the same among the three versions of the function. (`frexp` and `remquo` have a

pointer parameter, but it always has `int *` type.) Having a type-generic macro for `modf` would pose some difficult problems. For example, the meaning of `modf(d, &f)`, where `d` has type `double` and `f` has type `float`, is unclear: are we calling the `modf` function or the `modff` function? Rather than develop a complicated set of rules for a single function (and probably taking into account that `modf` isn't a very popular function), the C99 committee chose not to provide a type-generic `modf` macro.

- Q:** When a `<tgmath.h>` macro is invoked with an integer argument, the double version of the corresponding function is called. Shouldn't the float version be called, according to the usual arithmetic conversions? [p. 725]

A: We're dealing with a macro, not a function, so the usual arithmetic conversions don't come into play. The C99 committee had to create a rule for determining which version of a function would be called when an integer argument is passed to a `<tgmath.h>` macro. Although the committee at one point considered having the `float` version called (for consistency with the usual arithmetic conversions), they eventually decided that choosing the `double` version was better. First, it's safer: converting an integer to `float` may cause a loss of accuracy, especially for integer types whose width is 32 bits or more. Second, it causes fewer surprises for the programmer. Suppose that `i` is an integer variable. If the `<tgmath.h>` header isn't included, the call `sin(i)` calls the `sin` function. On the other hand, if `<tgmath.h>` is included, the call `sin(i)` invokes the `sin` macro; because `i` is an integer, the preprocessor replaces the `sin` macro with the `sin` function, and the end result is the same.

- Q:** When a program invokes one of the type-generic macros in `<tgmath.h>`, how does the implementation determine which function to call? Is there a way for a macro to test the types of its arguments?

A: One unusual aspect of `<tgmath.h>` is that its macros need to be able to test the types of the arguments that are passed to them. C has no features for testing types, so it would normally be impossible to write such a macro. The `<tgmath.h>` macros rely on special facilities provided by a particular compiler to make such testing possible. We don't know what these facilities are, and they're not guaranteed to be portable from one compiler to another.

Exercises

Section 27.1

- (C99) Locate the declarations of the `intN_t` and `uintN_t` types in the `<stdint.h>` header installed on your system. Which values of `N` are supported?
- (C99) Write the parameterized macros `INT32_C(n)`, `UINT32_C(n)`, `INT64_C(n)`, and `UINT64_C(n)`, assuming that the `int` type and `long int` types are 32 bits wide and the `long long int` type is 64 bits wide. Hint: Use the `##` preprocessor operator to attach

a suffix to `n` containing a combination of L and/or U characters. (See Section 7.1 for a discussion of how to use the L and U suffixes with integer constants.)

Section 27.2

3. (C99) In each of the following statements, assume that the variable `i` has the indicated original type. Using macros from the `<inttypes.h>` header, modify each statement so that it will work correctly if the type of `i` is changed to the indicated new type.
- | | | |
|---------------------------------------|--|---------------------------------|
| (a) <code>printf("%d", i);</code> | Original type: <code>int</code> | New type: <code>int8_t</code> |
| (b) <code>printf("%12.4d", i);</code> | Original type: <code>int</code> | New type: <code>int32_t</code> |
| (c) <code>printf("%-6o", i);</code> | Original type: <code>unsigned int</code> | New type: <code>uint16_t</code> |
| (d) <code>printf("%#x", i);</code> | Original type: <code>unsigned int</code> | New type: <code>uint64_t</code> |

Section 27.5

4. (C99) Assume that the following variable declarations are in effect:

```
int i;
float f;
double d;
long double ld;
float complex fc;
double complex dc;
long double complex ldc;
```

Each of the following is an invocation of a macro in `<tgmath.h>`. Show what it will look like after preprocessing, when the macro has been replaced by a function from `<math.h>` or `<complex.h>`.

- (a) `tan(i)`
- (b) `fabs(f)`
- (c) `asin(d)`
- (d) `exp(ld)`
- (e) `log(fc)`
- (f) `acosh(dc)`
- (g) `nexttoward(d, ld)`
- (h) `remainder(f, i)`
- (i) `copysign(d, ld)`
- (j) `carg(i)`
- (k) `cimag(f)`
- (l) `conj(ldc)`

Programming Projects

- (C99) Make the following modifications to the `quadratic.c` program of Section 27.4:
 - Have the user enter the coefficients of the polynomial (the values of the variables `a`, `b`, and `c`).
 - Have the program test the discriminant before displaying the values of the roots. If the discriminant is negative, have the program display the roots in the same way as before. If it's nonnegative, have the program display the roots as real numbers (without an imaginary part). For example, if the quadratic equation is $x^2 + x - 2 = 0$, the output of the program would be

```
root1 = 1
root2 = -2
```

(c) Modify the program so that it displays a complex number with a negative imaginary part as $a - bi$ instead of $a + -bi$. For example, the output of the program with the original coefficients would be

```
root1 = -0.2 + 0.4i  
root2 = -0.2 - 0.4i
```

2. (C99) Write a program that converts a complex number in Cartesian coordinates to polar form. The user will enter a and b (the real and imaginary parts of the number); the program will display the values of r and θ .

3. (C99) Write a program that converts a complex number in polar coordinates to Cartesian form. After the user enters the values of r and θ , the program will display the number in the form $a + bi$, where

$$\begin{aligned}a &= r \cos \theta \\b &= r \sin \theta\end{aligned}$$

4. (C99) Write a program that displays the n th roots of unity when given a positive integer n . The n th roots of unity are given by the formula $e^{2\pi ik/n}$, where k is an integer between 0 and $n - 1$.

APPENDIX A

C Operators

Precedence	Name	Symbol(s)	Associativity
1	Array subscripting	[]	Left
1	Function call	()	Left
1	Structure and union member	. ->	Left
1	Increment (postfix)	++	Left
1	Decrement (postfix)	--	Left
2	Increment (prefix)	++	Right
2	Decrement (prefix)	--	Right
2	Address	&	Right
2	Indirection	*	Right
2	Unary plus	+	Right
2	Unary minus	-	Right
2	Bitwise complement	~	Right
2	Logical negation	!	Right
2	Size	sizeof	Right
3	Cast	()	Right
4	Multiplicative	* / %	Left
5	Additive	+ -	Left
6	Bitwise shift	<< >>	Left
7	Relational	< > <= >=	Left
8	Equality	== !=	Left
9	Bitwise <i>and</i>	&	Left
10	Bitwise exclusive <i>or</i>	^	Left
11	Bitwise inclusive <i>or</i>		Left
12	Logical <i>and</i>	&&	Left
13	Logical <i>or</i>		Left
14	Conditional	? :	Right
15	Assignment	= *= /= %= += -= <<= >>= &= ^= =	Right
16	Comma	,	Left

APPENDIX B

C99 versus C89

This appendix lists many of the most significant differences between C89 and C99. (The smaller differences are too numerous to mention here.) The headings indicate which chapter contains the primary discussion of each C99 feature. Some of the changes attributed to C99 actually occurred earlier, in Amendment 1 to the C89 standard; these changes are marked “Amendment 1.”

2 C Fundamentals

- // comments* C99 adds a second kind of comment, which begins with `//`.
- identifiers* C89 requires compilers to remember the first 31 characters of identifiers; in C99, the requirement is 63 characters. Only the first six characters of names with external linkage are significant in C89. Moreover, the case of letters may not matter. In C99, the first 31 characters are significant, and the case of letters is taken into account.
- keywords* Five keywords are new in C99: `inline`, `restrict`, `_Bool`, `_Complex`, and `_Imaginary`.
- returning from main* In C89, if a program reaches the end of the `main` function without executing a `return` statement, the value returned to the operating system is undefined. In C99, if `main` is declared to return an `int`, the program returns 0 to the operating system.

4 Expressions

- / and % operators* The C89 standard states that if either operand is negative, the result of an integer division can be rounded either up or down. Moreover, if `i` or `j` is negative, the sign of `i % j` depends on the implementation. In C99, the result of a division is always truncated toward zero and the value of `i % j` has the same sign as `i`.

5 Selection Statements

_Bool type C99 provides a Boolean type named `_Bool`; C89 has no Boolean type.

6 Loops

for statements In C99, the first expression in a `for` statement can be replaced by a declaration, allowing the statement to declare its own control variable(s).

7 Basic Types

long long integer types C99 provides two additional standard integer types, `long long int` and `unsigned long long int`.

extended integer types In addition to the standard integer types, C99 allows implementation-defined extended signed and unsigned integer types.

long long integer constants C99 provides a way to indicate that an integer constant has type `long long int` or `unsigned long long int`.

types of integer constants C99's rules for determining the type of an integer constant are different from those in C89.

hexadecimal floating constants C99 provides a way to write floating constants in hexadecimal.

implicit conversions The rules for implicit conversions in C99 are somewhat different from the rules in C89, primarily because of C99's additional basic types.

8 Arrays

designated initializers C99 supports designated initializers, which can be used to initialize arrays, structures, and unions.

variable-length arrays In C99, the length of an array may be specified by an expression that's not constant, provided that the array doesn't have static storage duration and its declaration doesn't contain an initializer.

9 Functions

no default return type If the return type of a function is omitted in C89, the function is presumed to return a value of type `int`. In C99, it's illegal to omit the return type of a function.

mixed declarations and statements In C89, declarations must precede statements within a block (including the body of a function). In C99, declarations and statements can be mixed, as long as each variable is declared prior to the first statement that uses the variable.

<i>declaration or definition required prior to function call</i>	C99 requires that either a declaration or a definition of a function be present prior to any call of the function. C89 doesn't have this requirement; if a function is called without a prior declaration or definition, the compiler assumes that the function returns an <code>int</code> value.
<i>variable-length array parameters</i>	C99 allows variable-length array parameters. In a function declaration, the <code>*</code> symbol may appear inside brackets to indicate a variable-length array parameter.
<i>static array parameters</i>	C99 allows the use of the word <code>static</code> in the declaration of an array parameter, indicating a minimum length for the first dimension of the array.
<i>compound literals</i>	C99 supports the use of compound literals, which allow the creation of unnamed array and structure values.
<i>declaration of main</i>	C99 allows <code>main</code> to be declared in an implementation-defined manner, with a return type other than <code>int</code> and/or parameters other than those specified by the standard.
<i>return statement without expression</i>	In C89, executing a <code>return</code> statement without an expression in a non-void function causes undefined behavior (but only if the program attempts to use the value returned by the function). In C99, such a statement is illegal.

14 The Preprocessor

<i>additional predefined macros</i>	C99 provides several new predefined macros.
<i>empty macro arguments</i>	C99 allows any or all of the arguments in a macro call to be empty, provided that the call contains the correct number of commas.
<i>macros with a variable number of arguments</i>	In C89, a macro must have a fixed number of arguments, if it has any at all. C99 allows macros that take an unlimited number of arguments.
<i><code>_func_</code> identifier</i>	In C99, the <code>_func_</code> identifier behaves like a string variable that stores the name of the currently executing function.
<i>standard pragmas</i>	In C89, there are no standard pragmas. C99 has three: <code>CX_LIMITED_RANGE</code> , <code>FENV_ACCESS</code> , and <code>FP_CONTRACT</code> .
<i><code>_Pragma</code> operator</i>	C99 provides the <code>_Pragma</code> operator, which is used in conjunction with the <code>#pragma</code> directive.

16 Structures, Unions, and Enumerations

<i>structure type compatibility</i>	In C89, structures defined in different files are compatible if their members have the same names and appear in the same order, with corresponding members having
-------------------------------------	---

compatible types. C99 also requires that either both structures have the same tag or neither has a tag.

trailing comma in enumerations In C99, the last constant in an enumeration may be followed by a comma.

17 Advanced Uses of Pointers

restricted pointers C99 has a new keyword, `restrict`, that can appear in the declaration of a pointer.

flexible array members C99 allows the last member of a structure to be an array of unspecified length.

18 Declarations

block scopes for selection and iteration statements In C99, selection statements (`if` and `switch`) and iteration statements (`while`, `do`, and `for`)—along with the “inner” statements that they control—are considered to be blocks.

array, structure, and union initializers In C89, a brace-enclosed initializer for an array, structure, or union must contain only constant expressions. In C99, this restriction applies only if the variable has static storage duration.

inline functions C99 allows functions to be declared `inline`.

21 The Standard Library

<stdbool.h> header The `<stdbool.h>` header, which defines the `bool`, `true`, and `false` macros, is new in C99.

22 Input/Output

...printf conversion specifications The conversion specifications for the `...printf` functions have undergone a number of changes in C99, with new length modifiers, new conversion specifiers, the ability to write infinity and NaN, and support for wide characters. Also, the `%le`, `%lE`, `%lf`, `%lg`, and `%lG` conversions are legal in C99; they caused undefined behavior in C89.

...scanf conversion specifications In C99, the conversion specifications for the `...scanf` functions have new length modifiers, new conversion specifiers, the ability to read infinity and NaN, and support for wide characters.

snprintf function C99 adds the `snprintf` function to the `<stdio.h>` header.

23 Library Support for Numbers and Character Data

additional macros in <float.h> header C99 adds the `DECIMAL_DIG` and `FLT_EVAL_METHOD` macros to the `<float.h>` header.

<i>additional macros in <limits.h> header</i>	In C99, the <code><limits.h></code> header contains three new macros that describe the characteristics of the <code>long long int</code> types.
<i>math_errhandling macro</i>	C99 gives implementations a choice of how to inform a program that an error has occurred in a mathematical function: via a value stored in <code>errno</code> , via a floating-point exception, or both. The value of the <code>math_errhandling</code> macro (defined in <code><math.h></code>) indicates how errors are signaled by a particular implementation.
<i>additional functions in <math.h> header</i>	C99 adds two new versions of most <code><math.h></code> functions, one for <code>float</code> and one for <code>long double</code> . C99 also adds a number of completely new functions and function-like macros to <code><math.h></code> .

24 Error Handling

EILSEQ macro C99 adds the `EILSEQ` macro to the `<errno.h>` header.

25 International Features

digraphs Digraphs, which are two-character symbols that can be used as substitutes for the `[`, `]`, `{`, `}`, `#`, and `##` tokens, are new in C99. (Amendment 1)

<iso646.h> header The `<iso646.h>` header, which defines macros that represent operators containing the characters `&`, `|`, `~`, `!`, and `^`, is new in C99. (Amendment 1)

universal character names Universal character names, which provide a way to embed UCS characters in the source code of a program, are new in C99.

<wchar.h> header The `<wchar.h>` header, which provides functions for wide-character input/output and wide string manipulation, is new in C99. (Amendment 1)

<wctype.h> header The `<wctype.h>` header, the wide-character version of `<ctype.h>`, is new in C99. `<wctype.h>` provides functions for classifying and changing the case of wide characters. (Amendment 1)

26 Miscellaneous Library Functions

va_copy macro C99 adds a function-like macro named `va_copy` to the `<stdarg.h>` header.

additional functions in <stdio.h> header C99 adds the `vsnprintf`, `vfscanf`, `vscanf`, and `vsscanf` functions to the `<stdio.h>` header.

additional functions in <stdlib.h> header C99 adds five numeric conversion functions, the `_Exit` function, and `long long` versions of the `abs` and `div` functions to the `<stdlib.h>` header.

additional strftime conversion specifiers C99 adds a number of new `strftime` conversion specifiers. It also allows the use of an `E` or `O` character to modify the meaning of certain conversion specifiers.

27 Additional C99 Support for Mathematics

- <stdint.h> header* The *<stdint.h>* header, which declares integer types with specified widths, is new in C99.
- <inttypes.h> header* The *<inttypes.h>* header, which provides macros that are useful for input/output of the integer types in *<stdint.h>*, is new in C99.
- complex types* C99 provides three complex types: `float _Complex`, `double _Complex`, and `long double _Complex`.
- <complex.h> header* The *<complex.h>* header, which provides functions that perform mathematical operations on complex numbers, is new in C99.
- <tgmath.h> header* The *<tgmath.h>* header, which provides type-generic macros that make it easier to call library functions in *<math.h>* and *<complex.h>*, is new in C99.
- <fenv.h> header* The *<fenv.h>* header, which gives programs access to floating-point status flags and control modes, is new in C99.

APPENDIX C

C89 versus K&R C

This appendix lists most of the significant differences between C89 and K&R C (the language described in the first edition of Kernighan and Ritchie's *The C Programming Language*). The headings indicate which chapter of this book discusses each C89 feature. This appendix doesn't address the C library, which has changed much over the years. For other (less important) differences between C89 and K&R C, consult Appendices A and C in the second edition of K&R.

Most of today's C compilers can handle all of C89, but this appendix is useful if you happen to encounter older programs that were originally written for pre-C89 compilers.

2 C Fundamentals

identifiers In K&R C, only the first eight characters of an identifier are significant.

keywords K&R C lacks the keywords `const`, `enum`, `signed`, `void`, and `volatile`. In K&R C, the word `entry` is a keyword.

4 Expressions

unary + K&R C doesn't support the unary + operator.

5 Selection Statements

switch In K&R C, the controlling expression (and case labels) in a `switch` statement must have type `int` after promotion. In C89, the expression and labels may be of any integral type, including `unsigned int` and `long int`.

7 Basic Types

<i>unsigned types</i>	K&R C provides only one unsigned type (<code>unsigned int</code>).
<i>signed</i>	K&R C doesn't support the <code>signed</code> type specifier.
<i>number suffixes</i>	K&R C doesn't support the U (or u) suffix to specify that an integer constant is unsigned, nor does it support the F (or f) suffix to indicate that a floating constant is to be stored as a <code>float</code> value instead of a <code>double</code> value. In K&R C, the L (or l) suffix can't be used with floating constants.
<i>long float</i>	K&R C allows the use of <code>long float</code> as a synonym for <code>double</code> ; this usage isn't legal in C89.
<i>long double</i>	K&R C doesn't support the <code>long double</code> type.
<i>escape sequences</i>	The escape sequences \a, \v, and \? don't exist in K&R C. Also, K&R C doesn't support hexadecimal escape sequences.
<i>size_t</i>	In K&R C, the <code>sizeof</code> operator returns a value of type <code>int</code> ; in C89, it returns a value of type <code>size_t</code> .
<i>usual arithmetic conversions</i>	K&R C requires that <code>float</code> operands be converted to <code>double</code> . Also, K&R C specifies that combining a shorter unsigned integer with a longer signed integer always produces an unsigned result.

9 Functions

<i>function definitions</i>	In a C89 function definition, the types of the parameters are included in the parameter list:
	<pre>double square(double x) { return x * x; }</pre>
	K&R C requires that the types of parameters be specified in separate lists:
	<pre>double square(x) double x; { return x * x; }</pre>
<i>function declarations</i>	A C89 function declaration (prototype) specifies the types of the function's parameters (and the names as well, if desired):
	<pre>double square(double x); double square(double); /* alternate form */ int rand(void); /* no parameters */</pre>

A K&R C function declaration omits all information about parameters:

```
double square();
int rand();
```

function calls

When a K&R C definition or declaration is used, the compiler doesn't check that the function is called with arguments of the proper number and type. Furthermore, the arguments aren't automatically converted to the types of the corresponding parameters. Instead, the integral promotions are performed, and float arguments are converted to double.

`void` K&R C doesn't support the `void` type.

12 Pointers and Arrays

pointer subtraction

Subtracting two pointers produces an `int` value in K&R C but a `ptrdiff_t` value in C89.

13 Strings

string literals

In K&R C, adjacent string literals aren't concatenated. Also, K&R C doesn't prohibit the modification of string literals.

string initialization

In K&R C, an initializer for a character array of length n is limited to $n - 1$ characters (leaving room for a null character at the end). C89 allows the initializer to have length n .

14 The Preprocessor

`#elif`, `#error`, `#pragma`

K&R C doesn't support the `#elif`, `#error`, and `#pragma` directives.

`#`, `##`, `defined`

K&R C doesn't support the `#`, `##`, and `defined` operators.

16 Structures, Unions, and Enumerations

structure and union members and tags

In C89, each structure and union has its own name space for members; structure and union tags are kept in a separate name space. K&R C uses a single name space for members and tags, so members can't have the same name (with some exceptions), and members and tags can't overlap.

whole-structure operations

K&R C doesn't allow structures to be assigned, passed as arguments, or returned by functions.

enumerations

K&R C doesn't support enumerations.

17 Advanced Uses of Pointers

<i>void *</i>	In C89, <code>void *</code> is used as a “generic” pointer type; for example, <code>malloc</code> returns a value of type <code>void *</code> . In K&R C, <code>char *</code> is used for this purpose.
<i>pointer mixing</i>	K&R C allows pointers of different types to be mixed in assignments and comparisons. In C89, pointers of type <code>void *</code> can be mixed with pointers of other types, but any other mixing isn’t allowed without casting. Similarly, K&R C allows the mixing of integers and pointers in assignments and comparisons; C89 requires casting.
<i>pointers to functions</i>	If <code>pf</code> is a pointer to a function, C89 permits using either <code>(*pf) (...)</code> or <code>pf (...)</code> to call the function. K&R C allows only <code>(*pf) (...)</code> .

18 Declarations

<i>const and volatile</i>	K&R C doesn’t support the <code>const</code> and <code>volatile</code> type qualifiers.
<i>initialization of arrays, structures, and unions</i>	K&R C doesn’t allow the initialization of automatic arrays and structures, nor does it allow initialization of unions (regardless of storage duration).

25 International Features

<i>wide characters</i>	K&R C doesn’t support wide character constants and wide string literals.
<i>trigraph sequences</i>	K&R C doesn’t support trigraph sequences.

26 Miscellaneous Library Functions

<i>variable arguments</i>	K&R C doesn’t provide a portable way to write functions with a variable number of arguments, and it lacks the <code>...</code> (ellipsis) notation.
---------------------------	---

APPENDIX D

Standard Library Functions

This appendix describes all library functions supported by C89 and C99.* When using this appendix, please keep the following points in mind:

- In the interest of brevity and clarity, I've omitted many details. Some functions (notably `printf` and `scanf` and their variants) are covered in depth elsewhere in the book, so their descriptions here are minimal. For more information about a function (including examples of how it's used), see the section(s) listed in italic at the lower right corner of the function's description.
- As in other parts of the book, italics are used to indicate C99 differences. The names and prototypes of functions that were added in C99 are shown in italics. Changes to C89 prototypes (the addition of the word `restrict` to the declaration of certain parameters) are also italicized.
- Function-like macros are included in this appendix (with the exception of the type-generic macros in `<tgmath.h>`). Each prototype for a macro is followed by the word *macro*.
- In C99, some `<math.h>` functions have three versions (one each for `float`, `double`, and `long double`). All three are grouped into a single entry, under the name of the `double` version. For example, there's only one entry (under `acos`) for the `acos`, `acosf`, and `acosl` functions. The name of each additional version (`acosf` and `acosl`, in this example) appears to the left of its prototype. The `<complex.h>` functions, which also come in three versions, are treated in a similar fashion.
- Most of the `<wchar.h>` functions are wide-character versions of functions found in other headers. Unless there's a significant difference in behavior, the

*This material is adapted from international standard ISO/IEC 9899:1999.

description of each wide-character function simply refers the reader to the corresponding function found elsewhere.

- If some aspect of a function's behavior is described as *implementation-defined*, that means that it depends on how the C library is implemented. The function will always behave consistently, but the results may vary from one system to another. (In other words, check the manual to see what happens.) *Undefined* behavior, on the other hand, is bad news: not only may the behavior vary between systems, but the program may act strangely or even crash.
- The descriptions of many `<math.h>` functions refer to the terms *domain error* and *range error*. The way in which these errors are indicated changed between C89 and C99. For the C89 treatment of these errors, see Section 23.3. For the C99 treatment, see Section 23.4.
- The behavior of the following functions is affected by the current locale:

<code><ctype.h></code>	All functions
<code><stdio.h></code>	Formatted input/output functions
<code><stdlib.h></code>	Multibyte/wide-character conversion functions, numeric conversion functions
<code><string.h></code>	<code>strcoll</code> , <code>strxfrm</code>
<code><time.h></code>	<code>strftime</code>
<code><wchar.h></code>	<code>wcscoll</code> , <code>wcsftime</code> , <code>wcsxfrm</code> , formatted input/output functions, numeric conversion functions, extended multibyte/wide-character conversion functions
<code><wctype.h></code>	All functions

The `isalpha` function, for example, usually checks whether a character lies between `a` and `z` or `A` and `Z`. In some locales, other characters are considered alphabetic as well.

abort *Abort Program*

<stdlib.h>

```
void abort(void);
```

Raises the `SIGABRT` signal. If the signal isn't caught (or if the signal handler returns), the program terminates abnormally and returns an implementation-defined code indicating unsuccessful termination. Whether output buffers are flushed, open streams are closed, or temporary files are removed is implementation-defined.

26.2

abs *Integer Absolute Value*

<stdlib.h>

```
int abs(int j);
```

Returns Absolute value of `j`. The behavior is undefined if the absolute value of `j` can't be represented.

26.2

acos *Arc Cosine*

<math.h>

```
double acos(double x);
```

```
acosf float acosf(float x);
```

```
acosl long double acosl(long double x);
```

Returns	Arc cosine of x ; the return value is in the range 0 to π . A domain error occurs if x isn't between -1 and $+1$.	23.3
acosh	<i>Arc Hyperbolic Cosine (C99)</i>	<math.h>
	double acosh(double x);	
acoshf	float acoshf(float x);	
acoshl	long double acoshl(long double x);	
Returns	Arc hyperbolic cosine of x ; the return value is in the range 0 to $+\infty$. A domain error occurs if x is less than 1 .	23.4
asctime	<i>Convert Broken-Down Time to String</i>	<time.h>
	char *asctime(const struct tm *timeptr);	
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 in the structure pointed to by <i>timeptr</i> .	
		26.3
asin	<i>Arc Sine</i>	<math.h>
	double asin(double x);	
asinf	float asinf(float x);	
asinl	long double asinl(long double x);	
Returns	Arc sine of x ; the return value is in the range $-\pi/2$ to $+\pi/2$. A domain error occurs if x isn't between -1 and $+1$.	23.3
asinh	<i>Arc Hyperbolic Sine (C99)</i>	<math.h>
	double asinh(double x);	
asinhf	float asinhf(float x);	
asinhl	long double asinhl(long double x);	
Returns	Arc hyperbolic sine of x .	23.4
assert	<i>Assert Truth of Expression</i>	<assert.h>
	void assert(scalar expression);	macro
	If the value of <i>expression</i> is nonzero, <i>assert</i> does nothing. If the value is zero, <i>assert</i> writes a message to <i>stderr</i> (specifying the text of <i>expression</i> , the name of the source file containing the <i>assert</i> , and the line number of the <i>assert</i>); it then terminates the program by calling <i>abort</i> . To disable <i>assert</i> , define the macro <i>NDEBUG</i> before including <assert.h>. C99 changes: The argument is allowed to have any scalar type; C89 specifies that the type is <i>int</i> . Also, C99 requires that the message written by <i>assert</i> include the name of the function in which the <i>assert</i> appears; C89 doesn't have this requirement.	24.1
atan	<i>Arc Tangent</i>	<math.h>
	double atan(double x);	
atanf	float atanf(float x);	

atanl	<i>long double atanl(long double x);</i>	
<i>Returns</i>	Arc tangent of x; the return value is in the range $-\pi/2$ to $+\pi/2$.	23.3
atan2	<i>Arc Tangent of Quotient</i>	<math.h>
	<i>double atan2(double y, double x);</i>	
atan2f	<i>float atan2f(float y, float x);</i>	
atan2l	<i>long double atan2l(long double y, long double x);</i>	
<i>Returns</i>	Arc tangent of y/x ; the return value is in the range $-\pi$ to $+\pi$. A domain error may occur if x and y are both zero.	23.3
atanh	<i>Arc Hyperbolic Tangent (C99)</i>	<math.h>
	<i>double atanh(double x);</i>	
atanhf	<i>float atanhf(float x);</i>	
atanhl	<i>long double atanh1l(long double x);</i>	
<i>Returns</i>	Arc hyperbolic tangent of x. A domain error occurs if x is not between -1 and +1. A range error may occur if x is equal to -1 or +1.	23.4
atexit	<i>Register Function to Be Called at Program Exit</i>	<stdlib.h>
	<i>int atexit(void (*func)(void));</i>	
	Registers the function pointed to by func as a termination function. The function will be called if the program terminates normally (via <code>return</code> or <code>exit</code> but not <code>abort</code>).	
<i>Returns</i>	Zero if successful, nonzero if unsuccessful (an implementation-dependent limit has been reached).	26.2
atof	<i>Convert String to Floating-Point Number</i>	<stdlib.h>
	<i>double atof(const char *nptr);</i>	
<i>Returns</i>	A double value corresponding to the longest initial part of the string pointed to by nptr that has the form of a floating-point number. Returns zero if no conversion could be performed. The function's behavior is undefined if the number can't be represented.	26.2
atoi	<i>Convert String to Integer</i>	<stdlib.h>
	<i>int atoi(const char *nptr);</i>	
<i>Returns</i>	An int value corresponding to the longest initial part of the string pointed to by nptr that has the form of an integer. Returns zero if no conversion could be performed. The function's behavior is undefined if the number can't be represented.	26.2
atol	<i>Convert String to Long Integer</i>	<stdlib.h>
	<i>long int atol(const char *nptr);</i>	
<i>Returns</i>	A long int value corresponding to the longest initial part of the string pointed to by nptr that has the form of an integer. Returns zero if no conversion	

could be performed. The function's behavior is undefined if the number can't be represented. 26.2

atoll *Convert String to Long Long Integer (C99)* <stdlib.h>

```
long long int atoll(const char *nptr);
```

Returns A long long int value corresponding to the longest initial part of the string pointed to by nptr that has the form of an integer. Returns zero if no conversion could be performed. The function's behavior is undefined if the number can't be represented. 26.2

bsearch *Binary Search* <stdlib.h>

```
void *bsearch(const void *key, const void *base,
              size_t memb, size_t size,
              int (*compar)(const void *,
                             const void *));
```

Searches for the value pointed to by key in the sorted array pointed to by base. The array has nmemb elements, each size bytes long. compar is a pointer to a comparison function. When passed pointers to the key and an array element, in that order, the comparison 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.

Returns A pointer to an array element that tests equal to the key. Returns a null pointer if the key isn't found. 26.2

btowc *Convert Byte to Wide Character (C99)* <wchar.h>

```
wint_t btowc(int c);
```

Returns Wide-character representation of c. 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. 25.5

cabs *Complex Absolute Value (C99)* <complex.h>

```
double cabs(double complex z);
```

cabsf float cabsf(float complex z);

cabsl long double cabsl(long double complex z);

Returns Complex absolute value of z. 27.4

cacos *Complex Arc Cosine (C99)* <complex.h>

```
double complex cacos(double complex z);
```

cacosf float complex cacosf(float complex z);

cacosl long double complex cacosl(long double complex z);

Returns Complex arc cosine of z, with branch cuts outside the interval [-1, +1] along the real axis. The return value lies in a strip mathematically unbounded along the imaginary axis and in the interval [0, π] along the real axis. 27.4

cacosh	<i>Complex Arc Hyperbolic Cosine (C99)</i>	<complex.h>
	<i>double complex cacosh(double complex z);</i>	
cacoshf	<i>float complex cacoshf(float complex z);</i>	
cacoshl	<i>long double complex cacoshl(long double complex z);</i>	
<i>Returns</i>	Complex arc hyperbolic cosine of <i>z</i> , with a branch cut at values less than 1 along the real axis. The return value lies in a half-strip of nonnegative values along the real axis and in the interval $[-i\pi, +i\pi]$ along the imaginary axis.	27.4
calloc	<i>Allocate and Clear Memory Block</i>	<stdlib.h>
	<i>void *calloc(size_t nmemb, size_t size);</i>	
	Allocates a block of memory for an array with <i>nmemb</i> elements, each with <i>size</i> bytes. The block is cleared by setting all bits to zero.	
<i>Returns</i>	A pointer to the beginning of the block. Returns a null pointer if a block of the requested size can't be allocated.	17.3
carg	<i>Complex Argument (C99)</i>	<complex.h>
	<i>double carg(double complex z);</i>	
cargf	<i>float cargf(float complex z);</i>	
cargl	<i>long double cargl(long double complex z);</i>	
<i>Returns</i>	Argument (phase angle) of <i>z</i> , with a branch cut along the negative real axis. The return value lies in the interval $[-\pi, +\pi]$.	27.4
casin	<i>Complex Arc Sine (C99)</i>	<complex.h>
	<i>double complex casin(double complex z);</i>	
casinf	<i>float complex casinf(float complex z);</i>	
casinl	<i>long double complex casinl(long double complex z);</i>	
<i>Returns</i>	Complex arc sine of <i>z</i> , with branch cuts outside the interval $[-1, +1]$ along the real axis. The return value lies in a strip mathematically unbounded along the imaginary axis and in the interval $[-\pi/2, +\pi/2]$ along the real axis.	27.4
casinh	<i>Complex Arc Hyperbolic Sine (C99)</i>	<complex.h>
	<i>double complex casinh(double complex z);</i>	
casinhf	<i>float complex casinhf(float complex z);</i>	
casinhl	<i>long double complex casinhl(long double complex z);</i>	
<i>Returns</i>	Complex arc hyperbolic sine of <i>z</i> , with branch cuts outside the interval $[-i, +i]$ along the imaginary axis. The return value lies in a strip mathematically unbounded along the real axis and in the interval $[-i\pi/2, +i\pi/2]$ along the imaginary axis.	27.4
catan	<i>Complex Arc Tangent (C99)</i>	<complex.h>
	<i>double complex catan(double complex z);</i>	
catanf	<i>float complex catanf(float complex z);</i>	
catanl	<i>long double complex catanl(long double complex z);</i>	

Returns Complex arc tangent of z , with branch cuts outside the interval $[-i, +i]$ along the imaginary axis. The return value lies in a strip mathematically unbounded along the imaginary axis and in the interval $[-\pi/2, +\pi/2]$ along the real axis. 27.4

catanh *Complex Arc Hyperbolic Tangent (C99)* <complex.h>

double complex catanh(double complex z);

catanhf *float complex catanhf(float complex z);*

catanhl *long double complex catanhl(long double complex z);*

Returns Complex arc hyperbolic tangent of z , with branch cuts outside the interval $[-1, +1]$ along the real axis. The return value lies in a strip mathematically unbounded along the real axis and in the interval $[-i\pi/2, +i\pi/2]$ along the imaginary axis. 27.4

cbrt *Cube Root (C99)* <math.h>

double cbrt(double x);

cbrtf *float cbrtf(float x);*

cbrtl *long double cbrtl(long double x);*

Returns Real cube root of x . 23.4

ccos *Complex Cosine (C99)* <complex.h>

double complex ccos(double complex z);

ccosf *float complex ccosf(float complex z);*

ccosl *long double complex ccosl(long double complex z);*

Returns Complex cosine of z . 27.4

ccosh *Complex Hyperbolic Cosine (C99)* <complex.h>

double complex ccosh(double complex z);

ccoshf *float complex ccoshf(float complex z);*

ccoshl *long double complex ccoshl(long double complex z);*

Returns Complex hyperbolic cosine of z . 27.4

ceil *Ceiling* <math.h>

double ceil(double x);

ceilf *float ceilf(float x);*

ceill *long double ceill(long double x);*

Returns Smallest integer that is greater than or equal to x . 23.3

cexp *Complex Base-e Exponential (C99)* <complex.h>

double complex cexp(double complex z);

cexpf *float complex cexpf(float complex z);*

cexpl *long double complex cexpl(long double complex z);*

Returns Complex base- e exponential of z . 27.4

cimag *Imaginary Part of Complex Number (C99)* <complex.h>

double cimag(double complex z);

cimagf	<code>float cimagf(float complex z);</code>	
cimagl	<code>long double cimagl(long double complex z);</code>	
<i>Returns</i>	Imaginary part of z.	27.4
clearerr	<i>Clear Stream Error</i>	<code><stdio.h></code>
	<code>void clearerr(FILE *stream);</code>	
	Clears the end-of-file and error indicators for the stream pointed to by <code>stream</code> .	22.3
clock	<i>Processor Clock</i>	<code><time.h></code>
	<code>clock_t clock(void);</code>	
<i>Returns</i>	Elapsed processor time (measured in “clock ticks”) since the beginning of program execution. (To convert into seconds, divide by <code>CLOCKS_PER_SEC</code> .) Returns (<code>clock_t</code>) (-1) if the time is unavailable or can’t be represented.	26.3
clog	<i>Complex Natural Logarithm (C99)</i>	<code><complex.h></code>
	<code>double complex clog(double complex z);</code>	
clogf	<code>float complex clogf(float complex z);</code>	
clogl	<code>long double complex clogl(long double complex z);</code>	
<i>Returns</i>	Complex natural (base- <i>e</i>) logarithm of <i>z</i> , with a branch cut along the negative real axis. The return value lies in a strip mathematically unbounded along the real axis and in the interval $[-i\pi, +i\pi]$ along the imaginary axis.	27.4
conj	<i>Complex Conjugate (C99)</i>	<code><complex.h></code>
	<code>double complex conj(double complex z);</code>	
conjf	<code>float complex conjf(float complex z);</code>	
conjl	<code>long double complex conjl(long double complex z);</code>	
<i>Returns</i>	Complex conjugate of <i>z</i> .	27.4
copysign	<i>Copy Sign (C99)</i>	<code><math.h></code>
	<code>double copysign(double x, double y);</code>	
copysignf	<code>float copysignf(float x, float y);</code>	
copysignl	<code>long double copysignl(long double x, long double y);</code>	
<i>Returns</i>	A value with the magnitude of <i>x</i> and the sign of <i>y</i> .	23.4
cos	<i>Cosine</i>	<code><math.h></code>
	<code>double cos(double x);</code>	
cosf	<code>float cosf(float x);</code>	
cosl	<code>long double cosl(long double x);</code>	
<i>Returns</i>	Cosine of <i>x</i> (measured in radians).	23.3
cosh	<i>Hyperbolic Cosine</i>	<code><math.h></code>
	<code>double cosh(double x);</code>	
coshf	<code>float coshf(float x);</code>	

coshl long double coshl(long double x);**Returns** Hyperbolic cosine of x. A range error occurs if the magnitude of x is too large.

23.3

cpow Complex Power (C99)

<complex.h>

double complex cpow(double complex x,
double complex y);**cpowf** float complex cpowf(float complex x,
float complex y);**cpowl** long double complex cpowl(long double complex x,
long double complex y);**Returns** x raised to the power y, with a branch cut for the first parameter along the negative real axis.

27.4

cproj Complex Projection (C99)

<complex.h>

double complex cproj(double complex z);

cprojf float complex cprojf(float complex z);**cprojl** long double complex cprojl(long double complex z);**Returns** Projection of z onto the Riemann sphere. z is returned unless one of its parts is infinite, in which case the return value is INFINITY + I * copysign(0.0, cimag(z)).

27.4

creal Real Part of Complex Number (C99)

<complex.h>

double creal(double complex z);

crealf float crealf(float complex z);**creall** long double creall(long double complex z);**Returns** Real part of z.

27.4

csin Complex Sine (C99)

<complex.h>

double complex csin(double complex z);

csinf float complex csinf(float complex z);**csinl** long double complex csinl(long double complex z);**Returns** Complex sine of z.

27.4

csinh Complex Hyperbolic Sine (C99)

<complex.h>

double complex csinh(double complex z);

csinhf float complex csinhf(float complex z);**csinhl** long double complex csinhl(long double complex z);**Returns** Complex hyperbolic sine of z.

27.4

csqrt Complex Square Root (C99)

<complex.h>

double complex csqrt(double complex z);

csqrftf float complex csqrftf(float complex z);**csqrfl** long double complex csqrfl(long double complex z);

<i>Returns</i>	Complex square root of z , with a branch cut along the negative real axis. The return value lies in the right half-plane (including the imaginary axis).	27.4
ctan	<i>Complex Tangent (C99)</i>	<complex.h>
	double complex ctan(double complex z);	
ctanf	float complex ctanf(float complex z);	
ctanl	long double complex ctanl(long double complex z);	
<i>Returns</i>	Complex tangent of z .	27.4
ctanh	<i>Complex Hyperbolic Tangent (C99)</i>	<complex.h>
	double complex ctanh(double complex z);	
ctanhf	float complex ctanhf(float complex z);	
ctanhl	long double complex ctanhl(long double complex z);	
<i>Returns</i>	Complex hyperbolic tangent of z .	27.4
ctime	<i>Convert Calendar Time to String</i>	<time.h>
	char *ctime(const time_t *timer);	
<i>Returns</i>	A pointer to a string describing a local time equivalent to the calendar time pointed to by <i>timer</i> . Equivalent to asctime(localtime(timer)).	26.3
difftime	<i>Time Difference</i>	<time.h>
	double difftime(time_t <i>time1</i> , time_t <i>time0</i>);	
<i>Returns</i>	Difference between <i>time0</i> (the earlier time) and <i>time1</i> , measured in seconds.	26.3
div	<i>Integer Division</i>	<stdlib.h>
	div_t div(int <i>numer</i> , int <i>denom</i>);	
<i>Returns</i>	A <i>div_t</i> structure containing members named <i>quot</i> (the quotient when <i>numer</i> is divided by <i>denom</i>) and <i>rem</i> (the remainder). The behavior is undefined if either part of the result can't be represented.	26.2
erf	<i>Error Function (C99)</i>	<math.h>
	double erf(double <i>x</i>);	
erff	float erff(float <i>x</i>);	
erfl	long double erfl(long double <i>x</i>);	
<i>Returns</i>	<i>erf(x)</i> , where <i>erf</i> is the Gaussian error function.	23.4
erfc	<i>Complementary Error Function (C99)</i>	<math.h>
	double erfc(double <i>x</i>);	
erfcf	float erfcf(float <i>x</i>);	
erfc1	long double erfc1(long double <i>x</i>);	
<i>Returns</i>	<i>erfc(x) = 1 - erf(x)</i> , where <i>erf</i> is the Gaussian error function. A range error occurs if <i>x</i> is too large.	23.4

exit	<i>Exit from Program</i>	<stdlib.h>
	void exit(int status);	
	Calls all functions registered with atexit, flushes all output buffers, closes all open streams, removes any files created by tmpfile, and terminates the program. The value of status indicates whether the program terminated normally. The only portable values for status are 0 and EXIT_SUCCESS (both indicate successful termination) plus EXIT_FAILURE (unsuccessful termination).	
		9.5, 26.2
_Exit	<i>Exit from Program (C99)</i>	<stdlib.h>
	void _Exit(int status);	
	Causes normal program termination. Doesn't call functions registered with atexit or signal handlers registered with signal. The status returned is determined in the same way as for exit. Whether output buffers are flushed, open streams are closed, or temporary files are removed is implementation-defined.	
		26.2
exp	<i>Base-e Exponential</i>	<math.h>
	double exp(double x);	
expf	float expf(float x);	
expl	long double expl(long double x);	
<i>Returns</i>	e raised to the power x . A range error occurs if the magnitude of x is too large.	
		23.3
exp2	<i>Base-2 Exponential (C99)</i>	<math.h>
	double exp2(double x);	
exp2f	float exp2f(float x);	
exp2l	long double exp2l(long double x);	
<i>Returns</i>	2 raised to the power x . A range error occurs if the magnitude of x is too large.	
		23.4
expm1	<i>Base-e Exponential Minus 1 (C99)</i>	<math.h>
	double expm1(double x);	
expm1f	float expm1f(float x);	
expm1l	long double expm1l(long double x);	
<i>Returns</i>	e raised to the power x , minus 1. A range error occurs if x is too large.	
		23.4
fabs	<i>Floating Absolute Value</i>	<math.h>
	double fabs(double x);	
fabsf	float fabsf(float x);	
fabsl	long double fabsl(long double x);	
<i>Returns</i>	Absolute value of x .	
		23.3

fclose	<i>Close File</i>	<stdio.h>
	int fclose(FILE *stream);	
	Closes the stream pointed to by <i>stream</i> . Flushes any unwritten output remaining in the stream's buffer. Deallocates the buffer if it was allocated automatically.	
<i>Returns</i>	Zero if successful, EOF if an error was detected.	22.2
fdim	<i>Positive Difference (C99)</i>	<math.h>
	double fdim(double x, double y);	
fdimf	float fdimf(float x, float y);	
fdiml	long double fdiml(long double x, long double y);	
<i>Returns</i>	Positive difference of <i>x</i> and <i>y</i> :	
	$\begin{cases} x - y & \text{if } x > y \\ +0 & \text{if } x \leq y \end{cases}$	
	A range error may occur.	23.4
feclearexcept	<i>Clear Floating-Point Exceptions (C99)</i>	<fenv.h>
	int feclearexcept(int excepts);	
	Attempts to clear the floating-point exceptions represented by <i>excepts</i> .	
<i>Returns</i>	Zero if <i>excepts</i> is zero or if all specified exceptions were successfully cleared; otherwise, returns a nonzero value.	27.6
fegetenv	<i>Get Floating-Point Environment (C99)</i>	<fenv.h>
	int fegetenv(fenv_t *envp);	
	Attempts to store the current floating-point environment in the object pointed to by <i>envp</i> .	
<i>Returns</i>	Zero if the environment was successfully stored; otherwise, returns a nonzero value.	27.6
fegetexceptflag	<i>Get Floating-Point Exception Flags (C99)</i>	<fenv.h>
	int fegetexceptflag(fexcept_t *flagp, int excepts);	
	Attempts to retrieve the states of the floating-point status flags represented by <i>excepts</i> and store them in the object pointed to by <i>flagp</i> .	
<i>Returns</i>	Zero if the states of the status flags were successfully stored; otherwise, returns a nonzero value.	27.6
fegetround	<i>Get Floating-Point Rounding Direction (C99)</i>	<fenv.h>
	int fegetround(void);	
<i>Returns</i>	Value of the rounding-direction macro that represents the current rounding direction. Returns a negative value if the current rounding direction can't be determined or doesn't match any rounding-direction macro.	27.6

feholdexcept	<i>Save Floating-Point Environment (C99)</i>	<fenv.h>
	<i>int feholdexcept(fenv_t *envp);</i>	
	Saves the current floating-point environment in the object pointed to by <i>envp</i> , clears the floating-point status flags, and attempts to install a non-stop mode for all floating-point exceptions.	
<i>Returns</i>	Zero if non-stop floating-point exception handling was successfully installed; otherwise, returns a nonzero value.	27.6
feof	<i>Test for End-of-File</i>	<stdio.h>
	<i>int feof(FILE *stream);</i>	
<i>Returns</i>	A nonzero value if the end-of-file indicator is set for the stream pointed to by <i>stream</i> ; otherwise, returns zero.	22.3
feraiseexcept	<i>Raise Floating-Point Exceptions (C99)</i>	<fenv.h>
	<i>int feraiseexcept(int excepts);</i>	
	Attempts to raise supported floating-point exceptions represented by <i>excepts</i> .	
<i>Returns</i>	Zero if <i>excepts</i> is zero or if all specified exceptions were successfully raised; otherwise, returns a nonzero value.	27.6
ferror	<i>Test for File Error</i>	<stdio.h>
	<i>int ferror(FILE *stream);</i>	
<i>Returns</i>	A nonzero value if the error indicator is set for the stream pointed to by <i>stream</i> ; otherwise, returns zero.	22.3
fesetenv	<i>Set Floating-Point Environment (C99)</i>	<fenv.h>
	<i>int fesetenv(const fenv_t *envp);</i>	
	Attempts to establish the floating-point environment represented by the object pointed to by <i>envp</i> .	
<i>Returns</i>	Zero if the environment was successfully established; otherwise, returns a nonzero value.	27.6
fesetexceptflag	<i>Set Floating-Point Exception Flags (C99)</i>	<fenv.h>
	<i>int fesetexceptflag(const fexcept_t *flagp, int excepts);</i>	
	Attempts to set the floating-point status flags represented by <i>excepts</i> to the states stored in the object pointed to by <i>flagp</i> .	
<i>Returns</i>	Zero if <i>excepts</i> is zero or if all specified exceptions were successfully set; otherwise, returns a nonzero value.	27.6
fesetround	<i>Set Floating-Point Rounding Direction (C99)</i>	<fenv.h>
	<i>int fesetround(int round);</i>	

	Attempts to establish the rounding direction represented by round.	
Returns	Zero if the requested rounding direction was established; otherwise, returns a non-zero value.	27.6
fetestexcept	<i>Test Floating-Point Exception Flags (C99)</i> <code>int fetestexcept(int excepts);</code>	<fenv.h>
Returns	Bitwise <i>or</i> of the floating-point exception macros corresponding to the currently set flags for the exceptions represented by excepts.	27.6
feupdateenv	<i>Update Floating-Point Environment (C99)</i> <code>int feupdateenv(const fenv_t *envp);</code> Attempts to save the currently raised floating-point exceptions, install the floating-point environment represented by the object pointed to by envp, and then raise the saved exceptions.	<fenv.h>
Returns	Zero if all actions were successfully carried out; otherwise, returns a nonzero value.	27.6
fflush	<i>Flush File Buffer</i> <code>int fflush(FILE *stream);</code> Writes any unwritten data in the buffer associated with stream, which points to a stream that was opened for output or updating. If stream is a null pointer, fflush flushes all streams that have unwritten data stored in a buffer.	<stdio.h>
Returns	Zero if successful, EOF if a write error occurs.	22.2
fgetc	<i>Read Character from File</i> <code>int fgetc(FILE *stream);</code> Reads a character from the stream pointed to by stream.	<stdio.h>
Returns	Character read from the stream. If fgetc encounters the end of the stream, it sets the stream's end-of-file indicator and returns EOF. If a read error occurs, fgetc sets the stream's error indicator and returns EOF.	22.4
fgetpos	<i>Get File Position</i> <code>int fgetpos(FILE * restrict stream, fpos_t * restrict pos);</code> Stores the current position of the stream pointed to by stream in the object pointed to by pos.	<stdio.h>
Returns	Zero if successful. If the call fails, returns a nonzero value and stores an implementation-defined positive value in errno.	22.7
fgets	<i>Read String from File</i> <code>char *fgets(char * restrict s, int n, FILE * restrict stream);</code>	<stdio.h>

Reads characters from the stream pointed to by `stream` and stores them in the array pointed to by `s`. Reading stops at the first new-line character (which is stored in the string), when $n - 1$ characters have been read, or at end-of-file. `fgets` appends a null character to the string.

Returns `s` (a pointer to the array in which the input is stored). Returns a null pointer if a read error occurs or `fgets` encounters the end of the stream before it has stored any characters. 22.5

fgetwc *Read Wide Character from File (C99)* <wchar.h>
`wint_t fgetwc(FILE *stream);`
 Wide-character version of `fgetc`. 25.5

fgetws *Read Wide String from File (C99)* <wchar.h>
`wchar_t *fgetws(wchar_t * restrict s, int n,
FILE * restrict stream);`
 Wide-character version of `fgets`. 25.5

floor *Floor* <math.h>
`double floor(double x);`
floorf `float floorf(float x);`
floorl `long double floorl(long double x);`
Returns Largest integer that is less than or equal to `x`. 23.3

fma *Floating Multiply-Add (C99)* <math.h>
`double fma(double x, double y, double z);`
fmaf `float fmaf(float x, float y, float z);`
fmal `long double fmal(long double x, long double y,
long double z);`

Returns $(x \times y) + z$. The result is rounded only once, using the rounding mode corresponding to `FLT_ROUNDS`. A range error may occur. 23.4

fmax *Floating Maximum (C99)* <math.h>
`double fmax(double x, double y);`
fmaxf `float fmaxf(float x, float y);`
fmaxl `long double fmaxl(long double x, long double y);`
Returns Maximum of `x` and `y`. If one argument is a NaN and the other is numeric, the numeric value is returned. 23.4

fmin *Floating Minimum (C99)* <math.h>
`double fmin(double x, double y);`
fminf `float fminf(float x, float y);`
fminl `long double fminl(long double x, long double y);`
Returns Minimum of `x` and `y`. If one argument is a NaN and the other is numeric, the numeric value is returned. 23.4

fmod	<i>Floating Modulus</i>	<math.h>
	double fmod(double x, double y);	
fmodf	float fmodf(float x, float y);	
fmodl	long double fmodl(long double x, long double y);	
<i>Returns</i>	Remainder when x is divided by y. If y is zero, either a domain error occurs or zero is returned.	23.3
fopen	<i>Open File</i>	<stdio.h>
	FILE *fopen(const char * restrict filename, const char * restrict mode);	
	Opens the file whose name is pointed to by <i>filename</i> and associates it with a stream. <i>mode</i> specifies the mode in which the file is to be opened. Clears the error and end-of-file indicators for the stream.	
<i>Returns</i>	A file pointer to be used when performing subsequent operations on the file. Returns a null pointer if the file can't be opened.	22.2
fpclassify	<i>Floating-Point Classification (C99)</i>	<math.h>
	int fpclassify(real-floating x);	macro
<i>Returns</i>	Either FP_INFINITE, FP_NAN, FP_NORMAL, FP_SUBNORMAL, or FP_ZERO, depending on whether x is infinity, not a number, normal, subnormal, or zero, respectively.	23.4
fprintf	<i>Formatted File Write</i>	<stdio.h>
	int fprintf(FILE * restrict stream, const char * restrict format, ...);	
	Writes output to the stream pointed to by <i>stream</i> . The string pointed to by <i>format</i> specifies how subsequent arguments will be displayed.	
<i>Returns</i>	Number of characters written. Returns a negative value if an error occurs.	22.3
fputc	<i>Write Character to File</i>	<stdio.h>
	int fputc(int c, FILE *stream);	
	Writes the character <i>c</i> to the stream pointed to by <i>stream</i> .	
<i>Returns</i>	<i>c</i> (the character written). If a write error occurs, <i>fputc</i> sets the stream's error indicator and returns EOF.	22.4
fputs	<i>Write String to File</i>	<stdio.h>
	int fputs(const char * restrict s, FILE * restrict stream);	
	Writes the string pointed to by <i>s</i> to the stream pointed to by <i>stream</i> .	
<i>Returns</i>	A nonnegative value if successful. Returns EOF if a write error occurs.	22.5

fputwc	<i>Write Wide Character to File (C99)</i>	<wchar.h>
	<i>wint_t fputwc(wchar_t c, FILE *stream);</i>	
	Wide-character version of fputc.	25.5
fputws	<i>Write Wide String to File (C99)</i>	<wchar.h>
	<i>int fputws(const wchar_t * restrict s,</i> <i> FILE * restrict stream);</i>	
	Wide-character version of fputs.	25.5
fread	<i>Read Block from File</i>	<stdio.h>
	<i>size_t fread(void * restrict ptr, size_t size,</i> <i> size_t nmemb, FILE * restrict stream);</i>	
	Attempts to read nmemb elements, each size bytes long, from the stream pointed to by stream and store them in the array pointed to by ptr.	
<i>Returns</i>	Number of elements actually read. This number will be less than nmemb if fread encounters end-of-file or a read error occurs. Returns zero if either nmemb or size is zero.	22.6
free	<i>Free Memory Block</i>	<stdlib.h>
	<i>void free(void *ptr);</i>	
	Releases the memory block pointed to by ptr. (If ptr is a null pointer, the call has no effect.) The block must have been allocated by a call of calloc, malloc, or realloc.	17.4
freopen	<i>Reopen File</i>	<stdio.h>
	<i>FILE *freopen(const char * restrict filename,</i> <i> const char * restrict mode,</i> <i> FILE * restrict stream);</i>	
	Closes the file associated with stream, then opens the file whose name is pointed to by filename and associates it with stream. The mode parameter has the same meaning as in a call of fopen. <i>C99 change:</i> If filename is a null pointer, freopen attempts to change the stream's mode to that specified by mode.	
<i>Returns</i>	Value of stream if the operation succeeds. Returns a null pointer if the file can't be opened.	22.2
frexp	<i>Split into Fraction and Exponent</i>	<math.h>
	<i>double frexp(double value, int *exp);</i>	
frexpf	<i>float frexpf(float value, int *exp);</i>	
frexpl	<i>long double frexpl(long double value, int *exp);</i>	
	Splits value into a fractional part <i>f</i> and an exponent <i>n</i> in such a way that <i>value = f × 2ⁿ</i>	

<i>f</i>	<i>f</i> is normalized so that either $0.5 \leq f < 1$ or $f = 0$. Stores <i>n</i> in the object pointed to by <i>exp</i> .	
<i>Returns</i>	<i>f</i> , the fractional part of value.	23.3
fscanf	<i>Formatted File Read</i>	<stdio.h>
	<pre>int fscanf(FILE * restrict stream, const char * restrict format, ...);</pre>	
	Reads input items from the stream pointed to by <i>stream</i> . The string pointed to by <i>format</i> specifies the format of the items to be read. The arguments that follow <i>format</i> point to objects in which the items are to be stored.	
<i>Returns</i>	Number of input items successfully read and stored. Returns EOF if an input failure occurs before any items can be read.	22.3
fseek	<i>File Seek</i>	<stdio.h>
	<pre>int fseek(FILE *stream, long int offset, int whence);</pre>	
	Changes the file position indicator for the stream pointed to by <i>stream</i> . If <i>whence</i> is SEEK_SET, the new position is the beginning of the file plus <i>offset</i> bytes. If <i>whence</i> is SEEK_CUR, the new position is the current position plus <i>offset</i> bytes. If <i>whence</i> is SEEK_END, the new position is the end of the file plus <i>offset</i> bytes. The value of <i>offset</i> may be negative. For text streams, either <i>offset</i> must be zero or <i>whence</i> must be SEEK_SET and <i>offset</i> a value obtained by a previous call of ftell. For binary streams, fseek may not support calls in which <i>whence</i> is SEEK_END.	
<i>Returns</i>	Zero if the operation is successful, nonzero otherwise.	22.7
fsetpos	<i>Set File Position</i>	<stdio.h>
	<pre>int fsetpos(FILE *stream, const fpos_t *pos);</pre>	
	Sets the file position indicator for the stream pointed to by <i>stream</i> according to the value pointed to by <i>pos</i> (obtained from a previous call of fgetpos).	
<i>Returns</i>	Zero if successful. If the call fails, returns a nonzero value and stores an implementation-defined positive value in errno.	22.7
ftell	<i>Determine File Position</i>	<stdio.h>
	<pre>long int ftell(FILE *stream);</pre>	
<i>Returns</i>	Current file position indicator for the stream pointed to by <i>stream</i> . If the call fails, returns -1L and stores an implementation-defined positive value in errno.	
		22.7
fwide	<i>Get and Set Stream Orientation (C99)</i>	<wchar.h>
	<pre>int fwide(FILE *stream, int mode);</pre>	
	Determines the current orientation of a stream and, if desired, attempts to set its orientation. If <i>mode</i> is greater than zero, fwide tries to make the stream wide-oriented if it has no orientation. If <i>mode</i> is less than zero, it tries to make the	

stream byte-oriented if it has no orientation. If mode is zero, the orientation is not changed.

Returns A positive value if the stream has wide orientation after the call, a negative value if it has byte orientation, or zero if it has no orientation. 25.5

fwprintf *Wide-Character Formatted File Write (C99)* <wchar.h>
*int fwprintf(FILE * restrict stream,
 const wchar_t * restrict format, ...);*
 Wide-character version of fprintf. 25.5

fwrite *Write Block to File* <stdio.h>
*size_t fwrite(const void * restrict ptr, size_t size,
 size_t nmemb, FILE * restrict stream);*
 Writes nmemb elements, each size bytes long, from the array pointed to by ptr to the stream pointed to by stream.
Returns Number of elements actually written. This number will be less than nmemb if a write error occurs. In C99, returns zero if either nmemb or size is zero. 22.6

fwscanf *Wide-Character Formatted File Read (C99)* <wchar.h>
*int fwscanf(FILE * restrict stream,
 const wchar_t * restrict format, ...);*
 Wide-character version of fscanf. 25.5

getc *Read Character from File* <stdio.h>
*int getc(FILE *stream);*
 Reads a character from the stream pointed to by stream. Note: getc is normally implemented as a macro; it may evaluate stream more than once.
Returns Character read from the stream. If getc encounters the end of the stream, it sets the stream's end-of-file indicator and returns EOF. If a read error occurs, getc sets the stream's error indicator and returns EOF. 22.4

getchar *Read Character* <stdio.h>
int getchar(void);
 Reads a character from the stdin stream. Note: getchar is normally implemented as a macro.
Returns Character read from the stream. If getchar encounters the end of the stream, it sets the stream's end-of-file indicator and returns EOF. If a read error occurs, getchar sets the stream's error indicator and returns EOF. 7.3, 22.4

getenv *Get Environment String* <stdlib.h>
*char *getenv(const char *name);*
 Searches the operating system's environment list to see if any string matches the one pointed to by name.

Returns A pointer to the string associated with the matching name. Returns a null pointer if no match is found. 26.2

gets *Read String*

<stdio.h>

```
char *gets(char *s);
```

Reads characters from the `stdin` stream and stores them in the array pointed to by `s`. Reading stops at the first new-line character (which is discarded) or at end-of-file. `gets` appends a null character to the string.

Returns `s` (a pointer to the array in which the input is stored). Returns a null pointer if a read error occurs or `gets` encounters the end of the stream before it has stored any characters. 13.3, 22.5

getwc *Read Wide Character from File (C99)*

<wchar.h>

```
wint_t getwc(FILE *stream);
```

Wide-character version of `getc`.

25.5

getwchar *Read Wide Character (C99)*

<wchar.h>

```
wint_t getwchar(void);
```

Wide-character version of `getchar`.

25.5

gmtime *Convert Calendar Time to Broken-Down UTC Time*

<time.h>

```
struct tm *gmtime(const time_t *timer);
```

Returns A pointer to a structure containing a broken-down UTC time equivalent to the calendar time pointed to by `timer`. Returns a null pointer if the calendar time can't be converted to UTC. 26.3

hypot *Hypotenuse (C99)*

<math.h>

```
double hypot(double x, double y);
```

```
float hypotf(float x, float y);
```

```
long double hypotl(long double x, long double y);
```

Returns $\sqrt{x^2 + y^2}$ (the hypotenuse of a right triangle with legs `x` and `y`). A range error may occur. 23.4

ilogb *Unbiased Exponent (C99)*

<math.h>

```
int ilogb(double x);
```

```
int ilogbf(float x);
```

```
int ilogbl(long double x);
```

Returns Exponent of `x` as a signed integer; equivalent to calling the corresponding `logb` function and casting the returned value to type `int`. Returns `FP_ILOGB0` if `x` is zero, `INT_MAX` if `x` is infinite, and `FP_ILOGBNAN` if `x` is a NaN; a domain error or range error may occur in these cases. 23.4

imaxabs *Greatest-Width Integer Absolute Value (C99)*

<inttypes.h>

```
intmax_t imaxabs(intmax_t j);
```

<i>Returns</i>	Absolute value of <i>j</i> . The behavior is undefined if the absolute value of <i>j</i> can't be represented.	27.2
imaxdiv	<i>Greatest-Width Integer Division (C99)</i> <i>imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);</i>	<inttypes.h>
<i>Returns</i>	A structure of type <i>imaxdiv_t</i> containing members named <i>quot</i> (the quotient when <i>numer</i> is divided by <i>denom</i>) and <i>rem</i> (the remainder). The behavior is undefined if either part of the result can't be represented.	27.2
isalnum	<i>Test for Alphanumeric</i> <i>int isalnum(int c);</i>	<ctype.h>
<i>Returns</i>	A nonzero value if <i>c</i> is alphanumeric and zero otherwise. (<i>c</i> is alphanumeric if either <i>isalpha(c)</i> or <i>isdigit(c)</i> is true.)	23.5
isalpha	<i>Test for Alphabetic</i> <i>int isalpha(int c);</i>	<ctype.h>
<i>Returns</i>	A nonzero value if <i>c</i> is alphabetic and zero otherwise. In the "C" locale, <i>c</i> is alphabetic if either <i>islower(c)</i> or <i>isupper(c)</i> is true.	23.5
isblank	<i>Test for Blank (C99)</i> <i>int isblank(int c);</i>	<ctype.h>
<i>Returns</i>	A nonzero value if <i>c</i> is a blank character that is used to separate words within a line of text. In the "C" locale, the blank characters are space (' ') and horizontal tab ('\t').	23.5
iscntrl	<i>Test for Control Character</i> <i>int iscntrl(int c);</i>	<ctype.h>
<i>Returns</i>	A nonzero value if <i>c</i> is a control character and zero otherwise.	23.5
isdigit	<i>Test for Digit</i> <i>int isdigit(int c);</i>	<ctype.h>
<i>Returns</i>	A nonzero value if <i>c</i> is a decimal digit and zero otherwise.	23.5
isfinite	<i>Test for Finite Number (C99)</i> <i>int isfinite(real-floating x);</i>	<math.h> macro
<i>Returns</i>	A nonzero value if <i>x</i> is finite (zero, subnormal, or normal, but not infinite or NaN) and zero otherwise.	23.4
isgraph	<i>Test for Graphical Character</i> <i>int isgraph(int c);</i>	<ctype.h>
<i>Returns</i>	A nonzero value if <i>c</i> is a printing character (except a space) and zero otherwise.	23.5
isgreater	<i>Test for Greater Than (C99)</i> <i>int isgreater(real-floating x, real-floating y);</i>	<math.h> macro

<i>Returns</i>	($x > y$). Unlike the $>$ operator, <code>isgreater</code> doesn't raise the <i>invalid</i> floating-point exception if one or both of the arguments is a NaN.	23.4
<code>isgreaterequal</code>	<i>Test for Greater Than or Equal (C99)</i>	<code><math.h></code>
	<code>int isgreaterequal(real-floating x, real-floating y);</code>	macro
<i>Returns</i>	($x \geq y$). Unlike the \geq operator, <code>isgreaterequal</code> doesn't raise the <i>invalid</i> floating-point exception if one or both of the arguments is a NaN.	23.4
<code>isinf</code>	<i>Test for Infinity (C99)</i>	<code><math.h></code>
	<code>int isinf(real-floating x);</code>	macro
<i>Returns</i>	A nonzero value if x is infinity (positive or negative) and zero otherwise.	23.4
<code>isless</code>	<i>Test for Less Than (C99)</i>	<code><math.h></code>
	<code>int isless(real-floating x, real-floating y);</code>	macro
<i>Returns</i>	($x < y$). Unlike the $<$ operator, <code>isless</code> doesn't raise the <i>invalid</i> floating-point exception if one or both of the arguments is a NaN.	23.4
<code>islessequal</code>	<i>Test for Less Than or Equal (C99)</i>	<code><math.h></code>
	<code>int islessequal(real-floating x, real-floating y);</code>	macro
<i>Returns</i>	($x \leq y$). Unlike the \leq operator, <code>islessequal</code> doesn't raise the <i>invalid</i> floating-point exception if one or both of the arguments is a NaN.	23.4
<code>islessgreater</code>	<i>Test for Less Than or Greater Than (C99)</i>	<code><math.h></code>
	<code>int islessgreater(real-floating x, real-floating y);</code>	macro
<i>Returns</i>	($x < y \mid\mid x > y$). Unlike this expression, <code>islessgreater</code> doesn't raise the <i>invalid</i> floating-point exception if one or both of the arguments is a NaN; also, x and y are evaluated only once.	23.4
<code>islower</code>	<i>Test for Lower-Case Letter</i>	<code><ctype.h></code>
	<code>int islower(int c);</code>	
<i>Returns</i>	A nonzero value if c is a lower-case letter and zero otherwise.	23.5
<code>isnan</code>	<i>Test for NaN (C99)</i>	<code><math.h></code>
	<code>int isnan(real-floating x);</code>	macro
<i>Returns</i>	A nonzero value if x is a NaN value and zero otherwise.	23.4
<code>isnormal</code>	<i>Test for Normal Number (C99)</i>	<code><math.h></code>
	<code>int isnormal(real-floating x);</code>	macro
<i>Returns</i>	A nonzero value if x has a normal value (not zero, subnormal, infinite, or NaN) and zero otherwise.	23.4
<code>isprint</code>	<i>Test for Printing Character</i>	<code><ctype.h></code>
	<code>int isprint(int c);</code>	

Returns A nonzero value if *c* is a printing character (including a space) and zero otherwise.

23.5

ispunct *Test for Punctuation Character* <ctype.h>

```
int ispunct(int c);
```

Returns A nonzero value if *c* is a punctuation character and zero otherwise. All printing characters except the space (' ') and the alphanumeric characters are considered punctuation. *C99 change:* In the "C" locale, all printing characters except those for which isspace or isalnum is true are considered punctuation.

23.5

isspace *Test for White-Space Character* <ctype.h>

```
int isspace(int c);
```

Returns A nonzero value if *c* is a white-space character and zero otherwise. In the "C" locale, the white-space characters are space (' '), form feed ('\f'), new-line ('\n'), carriage return ('\r'), horizontal tab ('\t'), and vertical tab ('\v').

23.5

isunordered *Test for Unordered (C99)* <math.h>

```
int isunordered(real-floating x, real-floating y);
```

macro

Returns 1 if *x* and *y* are unordered (at least one is a NaN) and 0 otherwise.

23.4

isupper *Test for Upper-Case Letter* <ctype.h>

```
int isupper(int c);
```

Returns A nonzero value if *c* is an upper-case letter and zero otherwise.

23.5

iswalnum *Test for Alphanumeric Wide Character (C99)* <wctype.h>

```
int iswalnum(wint_t wc);
```

Returns A nonzero value if *wc* is alphanumeric and zero otherwise. (*wc* is alphanumeric if either iswalpha (*wc*) or iswdigit (*wc*) is true.)

25.6

iswalpha *Test for Alphabetic Wide Character (C99)* <wctype.h>

```
int iswalpha(wint_t wc);
```

Returns A nonzero value if *wc* is alphabetic and zero otherwise. (*wc* is alphabetic if iswupper (*wc*) or iswlower (*wc*) is true, or if *wc* is one of a locale-specific set of alphabetic wide characters for which none of iswcntrl, iswdigit, iswpunct, or iswspace is true.)

25.6

iswblank *Test for Blank Wide Character (C99)* <wctype.h>

```
int iswblank(wint_t wc);
```

Returns A nonzero value if *wc* is a standard blank wide character or one of a locale-specific set of wide characters for which iswspace is true and that are used to separate words within a line of text. In the "C" locale, iswblank returns true only for the standard blank characters: space (L' ') and horizontal tab (L'\t').

25.6

<i>iswcntrl</i>	<i>Test for Control Wide Character (C99)</i>	<wctype.h>
	<i>int iswcntrl(wint_t wc);</i>	
<i>Returns</i>	A nonzero value if <i>wc</i> is a control wide character and zero otherwise.	25.6
<i>iswctype</i>	<i>Test Type of Wide Character (C99)</i>	<wctype.h>
	<i>int iswctype(wint_t wc, wctype_t desc);</i>	
<i>Returns</i>	A nonzero value if the wide character <i>wc</i> has the property described by <i>desc</i> . (<i>desc</i> must be a value returned by a call of <i>wctype</i> ; the current setting of the LC_CTYPE category must be the same during both calls.) Returns zero otherwise.	25.6
<i>iswdigit</i>	<i>Test for Digit Wide Character (C99)</i>	<wctype.h>
	<i>int iswdigit(wint_t wc);</i>	
<i>Returns</i>	A nonzero value if <i>wc</i> corresponds to a decimal digit and zero otherwise.	25.6
<i>iswgraph</i>	<i>Test for Graphical Wide Character (C99)</i>	<wctype.h>
	<i>int iswgraph(wint_t wc);</i>	
<i>Returns</i>	A nonzero value if <i>iswprint(wc)</i> is true and <i>iswspace(wc)</i> is false. Returns zero otherwise.	25.6
<i>iswlower</i>	<i>Test for Lower-Case Wide Character (C99)</i>	<wctype.h>
	<i>int iswlower(wint_t wc);</i>	
<i>Returns</i>	A nonzero value if <i>wc</i> corresponds to a lower-case letter or is one of a locale-specific set of wide characters for which none of <i>iswcntrl</i> , <i>iswdigit</i> , <i>iswpunct</i> , or <i>iswspace</i> is true. Returns zero otherwise.	25.6
<i>iswprint</i>	<i>Test for Printing Wide Character (C99)</i>	<wctype.h>
	<i>int iswprint(wint_t wc);</i>	
<i>Returns</i>	A nonzero value if <i>wc</i> is a printing wide character and zero otherwise.	25.6
<i>iswpunct</i>	<i>Test for Punctuation Wide Character (C99)</i>	<wctype.h>
	<i>int iswpunct(wint_t wc);</i>	
<i>Returns</i>	A nonzero value if <i>wc</i> is a printing wide character that is one of a locale-specific set of punctuation wide characters for which neither <i>iswspace</i> nor <i>iswalnum</i> is true. Returns zero otherwise.	25.6
<i>iswspace</i>	<i>Test for White-Space Wide Character (C99)</i>	<wctype.h>
	<i>int iswspace(wint_t wc);</i>	
<i>Returns</i>	A nonzero value if <i>wc</i> is one of a locale-specific set of white-space wide characters for which none of <i>iswalnum</i> , <i>iswgraph</i> , or <i>iswpunct</i> is true. Returns zero otherwise.	25.6

iswupper	<i>Test for Upper-Case Wide Character (C99)</i>	<wctype.h>
	int iswupper(wint_t wc);	
<i>Returns</i>	A nonzero value if wc corresponds to an upper-case letter or is one of a locale-specific set of wide characters for which none of iswcntrl, iswdigit, iswpunct, or iswspace is true. Returns zero otherwise.	25.6
iswxdigit	<i>Test for Hexadecimal-Digit Wide Character (C99)</i>	<wctype.h>
	int iswxdigit(wint_t wc);	
<i>Returns</i>	A nonzero value if wc corresponds to a hexadecimal digit (0–9, a–f, A–F) and zero otherwise.	25.6
isxdigit	<i>Test for Hexadecimal Digit</i>	<ctype.h>
	int isxdigit(int c);	
<i>Returns</i>	A nonzero value if c is a hexadecimal digit (0–9, a–f, A–F) and zero otherwise.	23.5
labs	<i>Long Integer Absolute Value</i>	<stdlib.h>
	long int labs(long int j);	
<i>Returns</i>	Absolute value of j. The behavior is undefined if the absolute value of j can't be represented.	26.2
ldexp	<i>Combine Fraction and Exponent</i>	<math.h>
	double ldexp(double x, int exp);	
ldexpf	float ldexpf(float x, int exp);	
ldexpl	long double ldexpl(long double x, int exp);	
<i>Returns</i>	$x \times 2^{\text{exp}}$. A range error may occur.	23.3
ldiv	<i>Long Integer Division</i>	<stdlib.h>
	ldiv_t ldiv(long int numer, long int denom);	
<i>Returns</i>	An ldiv_t structure containing members named quot (the quotient when numer is divided by denom) and rem (the remainder). The behavior is undefined if either part of the result can't be represented.	26.2
lgamma	<i>Logarithm of Gamma Function (C99)</i>	<math.h>
	double lgamma(double x);	
lgammaf	float lgammaf(float x);	
lgammal	long double lgammal(long double x);	
<i>Returns</i>	$\ln(\Gamma(x))$, where Γ is the gamma function. A range error occurs if x is too large and may occur if x is a negative integer or zero.	23.4
llabs	<i>Long Long Integer Absolute Value (C99)</i>	<stdlib.h>
	long long int llabs(long long int j);	

<i>Returns</i>	Absolute value of <i>j</i> . The behavior is undefined if the absolute value of <i>j</i> can't be represented.	26.2
lldiv	<i>Long Long Integer Division (C99)</i>	<stdlib.h>
	<i>lldiv_t lldiv(long long int numer, long long int denom);</i>	
<i>Returns</i>	An <i>lldiv_t</i> structure containing members named <i>quot</i> (the quotient when <i>numer</i> is divided by <i>denom</i>) and <i>rem</i> (the remainder). The behavior is undefined if either part of the result can't be represented.	26.2
llrint	<i>Round to Long Long Integer Using Current Direction (C99)</i>	<math.h>
	<i>long long int llrint(double x); long long int llrintf(float x); long long int llrintl(long double x);</i>	
<i>Returns</i>	<i>x</i> rounded to the nearest integer using the current rounding direction. If the rounded value is outside the range of the <i>long long int</i> type, the result is unspecified and a domain or range error may occur.	23.4
llround	<i>Round to Nearest Long Long Integer (C99)</i>	<math.h>
	<i>long long int llround(double x); long long int llroundf(float x); long long int llroundl(long double x);</i>	
<i>Returns</i>	<i>x</i> rounded to the nearest integer, with halfway cases rounded away from zero. If the rounded value is outside the range of the <i>long long int</i> type, the result is unspecified and a domain or range error may occur.	23.4
localeconv	<i>Get Locale Conventions</i>	<locale.h>
	<i>struct lconv *localeconv(void);</i>	
<i>Returns</i>	A pointer to a structure containing information about the current locale.	25.1
localtime	<i>Convert Calendar Time to Broken-Down Local Time</i>	<time.h>
	<i>struct tm *localtime(const time_t *timer);</i>	
<i>Returns</i>	A pointer to a structure containing a broken-down local time equivalent to the calendar time pointed to by <i>timer</i> . Returns a null pointer if the calendar time can't be converted to local time.	26.3
log	<i>Natural Logarithm</i>	<math.h>
	<i>double log(double x); float logf(float x); long double logl(long double x);</i>	
<i>Returns</i>	Logarithm of <i>x</i> to the base <i>e</i> . A domain error occurs if <i>x</i> is negative. A range error may occur if <i>x</i> is zero.	23.3

log10	<i>Common Logarithm</i>	<math.h>
	double log10(double x);	
log10f	float log10f(float x);	
log10l	long double log10l(long double x);	
<i>Returns</i>	Logarithm of x to the base 10. A domain error occurs if x is negative. A range error may occur if x is zero.	23.3
log1p	<i>Natural Logarithm of 1 Plus Argument (C99)</i>	<math.h>
	double log1p(double x);	
log1pf	float log1pf(float x);	
log1pl	long double log1pl(long double x);	
<i>Returns</i>	Logarithm of 1 + x to the base e. A domain error occurs if x is less than -1. A range error may occur if x is equal to -1.	23.4
log2	<i>Base-2 Logarithm (C99)</i>	<math.h>
	double log2(double x);	
log2f	float log2f(float x);	
log2l	long double log2l(long double x);	
<i>Returns</i>	Logarithm of x to the base 2. A domain error occurs if x is negative. A range error may occur if x is zero.	23.4
logb	<i>Radix-Independent Exponent (C99)</i>	<math.h>
	double logb(double x);	
logbf	float logbf(float x);	
logbl	long double logbl(long double x);	
<i>Returns</i>	$\log_r(x)$, where r is the radix of floating-point arithmetic (defined by the macro FLT_RADIX, which typically has the value 2). A domain error or range error may occur if x is zero.	23.4
longjmp	<i>Nonlocal Jump</i>	<setjmp.h>
	void longjmp(jmp_buf env, int val);	
	Restores the environment stored in env and returns from the call of setjmp that originally saved env. If val is nonzero, it will be setjmp's return value; if val is 0, setjmp returns 1.	
		24.4
lrint	<i>Round to Long Integer Using Current Direction (C99)</i>	<math.h>
	long int lrint(double x);	
lrintf	long int lrintf(float x);	
lrintl	long int lrintl(long double x);	
<i>Returns</i>	x rounded to the nearest integer using the current rounding direction. If the rounded value is outside the range of the long int type, the result is unspecified and a domain or range error may occur.	23.4

lround	<i>Round to Nearest Long Integer (C99)</i>	<math.h>
	long int lround(double x);	
lroundf	long int lroundf(float x);	
lroundl	long int lroundl(long double x);	
<i>Returns</i>	x rounded to the nearest integer, with halfway cases rounded away from zero. If the rounded value is outside the range of the long int type, the result is unspecified and a domain or range error may occur.	23.4
malloc	<i>Allocate Memory Block</i>	<stdlib.h>
	void *malloc(size_t size);	
	Allocates a block of memory with size bytes. The block is not cleared.	
<i>Returns</i>	A pointer to the beginning of the block. Returns a null pointer if a block of the requested size can't be allocated.	17.2
mblen	<i>Length of Multibyte Character</i>	<stdlib.h>
	int mblen(const char *s, size_t n);	
<i>Returns</i>	If s is a null pointer, returns a nonzero or zero value, depending on whether or not multibyte characters have state-dependent encodings. If s points to a null character, returns zero. Otherwise, returns the number of bytes in the multibyte character pointed to by s; returns -1 if the next n or fewer bytes don't form a valid multibyte character.	25.2
mbrlen	<i>Length of Multibyte Character – Restartable (C99)</i>	<wchar.h>
	size_t mbrlen(const char * restrict s, size_t n, mbstate_t * restrict ps);	
	Determines the number of bytes in the array pointed to by s that are required to complete a multibyte character. ps should point to an object of type mbstate_t that contains the current conversion state. A call of mbrlen is equivalent to mbrtowc(NULL, s, n, ps)	
	except that if ps is a null pointer, the address of an internal object is used instead.	
<i>Returns</i>	See mbrtowc.	25.5
mbrtowc	<i>Convert Multibyte Character to Wide Character – Restartable (C99)</i>	<wchar.h>
	size_t mbrtowc(wchar_t * restrict pwc, const char * restrict s, size_t n, mbstate_t * restrict ps);	
	If s is a null pointer, a call of mbrtowc is equivalent to mbrtowc(NULL, "", 1, ps)	
	Otherwise, mbrtowc examines up to n bytes in the array pointed to by s to see if	

they complete a valid multibyte character. If so, the multibyte character is converted into a wide character. If `pwc` isn't a null pointer, the wide character is stored in the object pointed to by `pwc`. The value of `ps` should be a pointer to an object of type `mbstate_t` that contains the current conversion state. If `ps` is a null pointer, `mbrtowc` uses an internal object to store the conversion state. If the result of the conversion is the null wide character, the `mbstate_t` object used during the call is left in the initial conversion state.

Returns 0 if the conversion produces a null wide character. 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. Returns `(size_t) (-2)` if the `n` bytes pointed to by `s` weren't enough to complete a multibyte character. Returns `(size_t) (-1)` and stores `EILSEQ` in `errno` if an encoding error occurs.

25.5

mbsinit *Test for Initial Conversion State (C99)* `<wchar.h>`

```
int mbsinit(const mbstate_t *ps);
```

Returns A nonzero value if `ps` is a null pointer or it points to an `mbstate_t` object that describes an initial conversion state; otherwise, returns zero.

25.5

mbsrtowcs *Convert Multibyte String to Wide String – Restartable (C99)* `<wchar.h>`

```
size_t mbsrtowcs(wchar_t * restrict dst,
                  const char ** restrict src,
                  size_t len, mbstate_t * restrict ps);
```

Converts a sequence of multibyte characters from the array indirectly pointed to by `src` into a sequence of corresponding wide characters. `ps` should point to an object of type `mbstate_t` that contains the current conversion state. If the argument corresponding to `ps` is a null pointer, `mbsrtowcs` uses an internal object to store the conversion state. If `dst` isn't a null pointer, the converted characters are stored in the array that it points to. Conversion continues up to and including a terminating null character, which is also stored. Conversion stops earlier if a sequence of bytes is encountered that doesn't form a valid multibyte character or—if `dst` isn't a null pointer—when `len` wide characters have been stored in the array. If `dst` isn't a null pointer, the object pointed to by `src` is assigned either a null pointer (if a terminating null character was reached) or the address just past the last multibyte character converted (if any). If the conversion ends at a null character and if `dst` isn't a null pointer, the resulting state is the initial conversion state.

Returns Number of multibyte characters successfully converted, not including any terminating null character. Returns `(size_t) (-1)` and stores `EILSEQ` in `errno` if an invalid multibyte character is encountered.

25.5

mbstowcs *Convert Multibyte String to Wide String* `<stdlib.h>`

```
size_t mbstowcs(wchar_t * restrict pwcs,
                 const char * restrict s, size_t n);
```

Converts the sequence of multibyte characters pointed to by *s* into a sequence of wide characters, storing at most *n* wide characters in the array pointed to by *pwc*. Conversion ends if a null character is encountered; it is converted into a null wide character.

Returns Number of array elements modified, not including the null wide character, if any. Returns (*size_t*) (-1) if an invalid multibyte character is encountered. 25.2

mbtowc *Convert Multibyte Character to Wide Character* <stdlib.h>

```
int mbtowc(wchar_t * restrict pwc,
           const char * restrict s, size_t n);
```

If *s* isn't a null pointer, converts the multibyte character pointed to by *s* into a wide character; at most *n* bytes will be examined. If the multibyte character is valid and *pwc* isn't a null pointer, stores the value of the wide character in the object pointed to by *pwc*.

Returns If *s* is a null pointer, returns a nonzero or zero value, depending on whether or not multibyte characters have state-dependent encodings. If *s* points to a null character, returns zero. Otherwise, returns the number of bytes in the multibyte character pointed to by *s*; returns -1 if the next *n* or fewer bytes don't form a valid multibyte character. 25.2

memchr *Search Memory Block for Character* <string.h>

```
void *memchr(const void *s, int c, size_t n);
```

Returns A pointer to the first occurrence of the character *c* among the first *n* characters of the object pointed to by *s*. Returns a null pointer if *c* isn't found. 23.6

memcmp *Compare Memory Blocks* <string.h>

```
int memcmp(const void *s1, const void *s2, size_t n);
```

Returns A negative, zero, or positive integer, depending on whether the first *n* characters of the object pointed to by *s1* are less than, equal to, or greater than the first *n* characters of the object pointed to by *s2*. 23.6

memcpy *Copy Memory Block* <string.h>

```
void *memcpy(void * restrict s1,
            const void * restrict s2, size_t n);
```

Copies *n* characters from the object pointed to by *s2* into the object pointed to by *s1*. The behavior is undefined if the objects overlap.

Returns *s1* (a pointer to the destination). 23.6

memmove *Copy Memory Block* <string.h>

```
void *memmove(void *s1, const void *s2, size_t n);
```

Copies *n* characters from the object pointed to by *s2* into the object pointed to by *s1*. Will work properly if the objects overlap.

Returns *s1* (a pointer to the destination). 23.6

memset	<i>Initialize Memory Block</i>	<string.h>
	void *memset(void *s, int c, size_t n);	
	Stores c in each of the first n characters of the object pointed to by s.	
<i>Returns</i>	s (a pointer to the object).	23.6
mktimē	<i>Convert Broken-Down Local Time to Calendar Time</i>	<time.h>
	time_t mktimē(struct tm *timeptr);	
	Converts a broken-down local time (stored in the structure pointed to by timeptr) into a calendar time. The members of the structure aren't required to be within their legal ranges; also, the values of tm_wday (day of the week) and tm_yday (day of the year) are ignored. mktimē stores values in tm_wday and tm_yday after adjusting the other members to bring them into their proper ranges.	
<i>Returns</i>	A calendar time corresponding to the structure pointed to by timeptr. Returns (time_t) (-1) if the calendar time can't be represented.	26.3
modf	<i>Split into Integer and Fractional Parts</i>	<math.h>
	double modf(double value, double *iptr);	
modff	float modff(float value, float *iptr);	
modfl	long double modfl(long double value, long double *iptr);	
	Splits value into integer and fractional parts; stores the integer part in the object pointed to by iptr.	
<i>Returns</i>	Fractional part of value.	23.3
nan	<i>Create NaN (C99)</i>	<math.h>
	double nan(const char *tagp);	
nanf	float nanf(const char *tagp);	
nanl	long double nanl(const char *tagp);	
<i>Returns</i>	A "quiet" NaN whose binary pattern is determined by the string pointed to by tagp. Returns zero if quiet NaNs aren't supported.	23.4
nearbyint	<i>Round to Integral Value Using Current Direction (C99)</i>	<math.h>
	double nearbyint(double x);	
nearbyintf	float nearbyintf(float x);	
nearbyintl	long double nearbyintl(long double x);	
<i>Returns</i>	x rounded to an integer (in floating-point format) using the current rounding direction. Doesn't raise the <i>inexact</i> floating-point exception.	23.4
nextafter	<i>Next Number After (C99)</i>	<math.h>
	double nextafter(double x, double y);	
nextafterf	float nextafterf(float x, float y);	
nextafterl	long double nextafterl(long double x, long double y);	

Returns	Next representable value after x in the direction of y . Returns the value just before x if $y < x$ or the value just after x if $x < y$. Returns y if x equals y . A range error may occur if the magnitude of x is the largest representable finite value and the result is infinite or not representable.	23.4
nexttoward	<i>Next Number Toward (C99)</i>	<math.h>
	<i>double nexttoward(double x, long double y);</i> <i>float nexttowardf(float x, long double y);</i> <i>long double nexttowardl(long double x, long double y);</i>	
nexttowardf		
nexttowardl		
Returns	Next representable value after x in the direction of y (see <i>nextafter</i>). Returns y converted to the function's type if x equals y .	23.4
perror	<i>Print Error Message</i>	<stdio.h>
	<i>void perror(const char *s);</i>	
	Writes the following message to the <i>stderr</i> stream: <i>string : error-message</i>	
	<i>string</i> is the string pointed to by <i>s</i> and <i>error-message</i> is an implementation-defined message that matches the one returned by the call <i>strerror(errno)</i> .	24.2
pow	<i>Power</i>	<math.h>
	<i>double pow(double x, double y);</i> <i>float powf(float x, float y);</i> <i>long double powl(long double x, long double y);</i>	
powf		
powl		
Returns	x raised to the power y . A domain or range error may occur in certain cases, which vary between C89 and C99.	23.3
printf	<i>Formatted Write</i>	<stdio.h>
	<i>int printf(const char * restrict format, ...);</i>	
	Writes output to the <i>stdout</i> stream. The string pointed to by <i>format</i> specifies how subsequent arguments will be displayed.	
Returns	Number of characters written. Returns a negative value if an error occurs.	3.1, 22.3
putc	<i>Write Character to File</i>	<stdio.h>
	<i>int putc(int c, FILE *stream);</i>	
	Writes the character <i>c</i> to the stream pointed to by <i>stream</i> . <i>Note:</i> <i>putc</i> is normally implemented as a macro; it may evaluate <i>stream</i> more than once.	
Returns	<i>c</i> (the character written). If a write error occurs, <i>putc</i> sets the stream's error indicator and returns EOF.	22.4
putchar	<i>Write Character</i>	<stdio.h>
	<i>int putchar(int c);</i>	
	Writes the character <i>c</i> to the <i>stdout</i> stream. <i>Note:</i> <i>putchar</i> is normally implemented as a macro.	

Returns	c (the character written). If a write error occurs, putchar sets the stream's error indicator and returns EOF.	7.3, 22.4
puts	<i>Write String</i> int puts(const char *s); Writes the string pointed to by s to the stdout stream, then writes a new-line character.	<stdio.h>
Returns	A nonnegative value if successful. Returns EOF if a write error occurs.	13.3, 22.5
putwc	<i>Write Wide Character to File (C99)</i> wint_t putwc(wchar_t c, FILE *stream); Wide-character version of putc.	<wchar.h> 25.5
putwchar	<i>Write Wide Character (C99)</i> wint_t putwchar(wchar_t c); Wide-character version of putchar.	<wchar.h> 25.5
qsort	<i>Sort Array</i> void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *)); Sorts the array pointed to by base. The array has nmemb elements, each size bytes long. compar is a pointer to a comparison function. When passed pointers to two array elements, the comparison function must return a negative, zero, or positive integer, depending on whether the first array element is less than, equal to, or greater than the second.	<stdlib.h> 17.7, 26.2
raise	<i>Raise Signal</i> int raise(int sig); Raises the signal whose number is sig.	<signal.h>
Returns	Zero if successful, nonzero otherwise.	24.3
rand	<i>Generate Pseudo-Random Number</i> int rand(void); Returns A pseudo-random integer between 0 and RAND_MAX (inclusive).	<stdlib.h> 26.2
realloc	<i>Resize Memory Block</i> void *realloc(void *ptr, size_t size); ptr is assumed to point to a block of memory previously obtained from calloc, malloc, or realloc. realloc allocates a block of size bytes, copying the contents of the old block if necessary.	<stdlib.h> 17.3
Returns	A pointer to the beginning of the new memory block. Returns a null pointer if a block of the requested size can't be allocated.	

remainder	<i>Remainder (C99)</i>	<math.h>
	<i>double remainder(double x, double y);</i>	
remainderf	<i>float remainderf(float x, float y);</i>	
remainderl	<i>long double remainderl(long double x, long double y);</i>	
<i>Returns</i>	<i>x - ny</i> , where <i>n</i> is the integer nearest the exact value of <i>x/y</i> . (If <i>x/y</i> is halfway between two integers, <i>n</i> is even.) If <i>x - ny</i> = 0, the return value has the same sign as <i>x</i> . If <i>y</i> is zero, either a domain error occurs or zero is returned.	23.4
remove	<i>Remove File</i>	<stdio.h>
	<i>int remove(const char *filename);</i>	
	Deletes the file whose name is pointed to by <i>filename</i> .	
<i>Returns</i>	Zero if successful, nonzero otherwise.	22.2
remquo	<i>Remainder and Quotient (C99)</i>	<math.h>
	<i>double remquo(double x, double y, int *quo);</i>	
remquof	<i>float remquof(float x, float y, int *quo);</i>	
remquol	<i>long double remquol(long double x, long double y, int *quo);</i>	
	Computes both the remainder and the quotient when <i>x</i> is divided by <i>y</i> . The object pointed to by <i>quo</i> is modified so that it contains <i>n</i> low-order bits of the integer quotient $ x/y $, where <i>n</i> is implementation-defined but must be at least three. The value stored in this object will be negative if $x/y < 0$.	
<i>Returns</i>	Same value as the corresponding <i>remainder</i> function. If <i>y</i> is zero, either a domain error occurs or zero is returned.	23.4
rename	<i>Rename File</i>	<stdio.h>
	<i>int rename(const char *old, const char *new);</i>	
	Changes the name of a file. <i>old</i> and <i>new</i> point to strings containing the old name and new name, respectively.	
<i>Returns</i>	Zero if the renaming is successful. Returns a nonzero value if the operation fails (perhaps because the old file is currently open).	22.2
rewind	<i>Rewind File</i>	<stdio.h>
	<i>void rewind(FILE *stream);</i>	
	Sets the file position indicator for the stream pointed to by <i>stream</i> to the beginning of the file. Clears the error and end-of-file indicators for the stream.	22.7
rint	<i>Round to Integral Value Using Current Direction (C99)</i>	<math.h>
	<i>double rint(double x);</i>	
rintf	<i>float rintf(float x);</i>	
rintl	<i>long double rintl(long double x);</i>	
<i>Returns</i>	<i>x</i> rounded to an integer (in floating-point format) using the current rounding direc-	

tion. May raise the *inexact* floating-point exception if the result has a different value than *x*. 23.4

round	<i>Round to Nearest Integral Value (C99)</i>	<math.h>
	<i>double round(double x);</i>	
roundf	<i>float roundf(float x);</i>	
roundl	<i>long double roundl(long double x);</i>	
<i>Returns</i>	<i>x</i> rounded to the nearest integer (in floating-point format). Halfway cases are rounded away from zero. 23.4	
scalbln	<i>Scale Floating-Point Number Using Long Integer (C99)</i>	<math.h>
	<i>double scalbln(double x, long int n);</i>	
scalblnf	<i>float scalblnf(float x, long int n);</i>	
scalblnl	<i>long double scalblnl(long double x, long int n);</i>	
<i>Returns</i>	<i>x</i> × FLT_RADIX ^{<i>n</i>} , computed in an efficient way. A range error may occur. 23.4	
scalbn	<i>Scale Floating-Point Number Using Integer (C99)</i>	<math.h>
	<i>double scalbn(double x, int n);</i>	
scalbnf	<i>float scalbnf(float x, int n);</i>	
scalbnl	<i>long double scalbnl(long double x, int n);</i>	
<i>Returns</i>	<i>x</i> × FLT_RADIX ^{<i>n</i>} , computed in an efficient way. A range error may occur. 23.4	
scanf	<i>Formatted Read</i>	<stdio.h>
	<i>int scanf(const char * restrict format, ...);</i>	
	Reads input items from the <i>stdin</i> stream. The string pointed to by <i>format</i> specifies the format of the items to be read. The arguments that follow <i>format</i> point to objects in which the items are to be stored.	
<i>Returns</i>	Number of input items successfully read and stored. Returns EOF if an input failure occurs before any items can be read. 3.2, 22.3	
setbuf	<i>Set Buffer</i>	<stdio.h>
	<i>void setbuf(FILE * restrict stream, char * restrict buf);</i>	
	If <i>buf</i> isn't a null pointer, a call of <i>setbuf</i> is equivalent to:	
	<i>(void) setvbuf(stream, buf, _IOFBF, BUFSIZ);</i>	
	Otherwise, it's equivalent to:	
	<i>(void) setvbuf(stream, NULL, _IONBF, 0);</i> 22.2	
setjmp	<i>Prepare for Nonlocal Jump</i>	<setjmp.h>
	<i>int setjmp(jmp_buf env);</i>	macro
	Stores the current environment in <i>env</i> for use in a later call of <i>longjmp</i> .	
<i>Returns</i>	Zero when called directly. Returns a nonzero value when returning from a call of <i>longjmp</i> . 24.4	

setlocale	<i>Set Locale</i>	<locale.h>
	char *setlocale(int category, const char *locale);	
	Sets a portion of the program's locale. <i>category</i> indicates which portion is affected. <i>locale</i> points to a string representing the new locale.	
<i>Returns</i>	If <i>locale</i> is a null pointer, returns a pointer to the string associated with <i>category</i> for the current locale. Otherwise, returns a pointer to the string associated with <i>category</i> for the new locale. Returns a null pointer if the operation fails.	25.1
setvbuf	<i>Set Buffer</i>	<stdio.h>
	int setvbuf(FILE * restrict stream, char * restrict buf, int mode, size_t size);	
	Changes the buffering of the stream pointed to by <i>stream</i> . The value of <i>mode</i> can be either <i>_IOFBF</i> (full buffering), <i>_IOLBF</i> (line buffering), or <i>_IONBF</i> (no buffering). If <i>buf</i> is a null pointer, a buffer is automatically allocated if needed. Otherwise, <i>buf</i> points to a memory block that can be used as the buffer; <i>size</i> is the number of bytes in the block. <i>Note:</i> <i>setvbuf</i> must be called after the stream is opened but before any other operations are performed on it.	
<i>Returns</i>	Zero if the operation is successful. Returns a nonzero value if <i>mode</i> is invalid or the request can't be honored.	22.2
signal	<i>Install Signal Handler</i>	<signal.h>
	void (*signal(int sig, void (*func)(int)))(int);	
	Installs the function pointed to by <i>func</i> as the handler for the signal whose number is <i>sig</i> . Passing <i>SIG_DFL</i> as the second argument causes default handling for the signal; passing <i>SIG_IGN</i> causes the signal to be ignored.	
<i>Returns</i>	A pointer to the previous handler for this signal; returns <i>SIG_ERR</i> and stores a positive value in <i>errno</i> if the handler can't be installed.	24.3
signbit	<i>Sign Bit (C99)</i>	<math.h>
	int signbit(real-floating x);	macro
<i>Returns</i>	A nonzero value if the sign of <i>x</i> is negative and zero otherwise. The value of <i>x</i> may be any number, including infinity and NaN.	23.4
sin	<i>Sine</i>	<math.h>
	double sin(double x);	
sinf	float sinf(float x);	
sinl	long double sinl(long double x);	
<i>Returns</i>	Sine of <i>x</i> (measured in radians).	23.3
sinh	<i>Hyperbolic Sine</i>	<math.h>
	double sinh(double x);	

sinhf float *sinhf*(float *x*);
sinhl long double *sinhl*(long double *x*);

Returns Hyperbolic sine of *x*. A range error occurs if the magnitude of *x* is too large. 23.3

snprintf *Bounded Formatted String Write (C99)* <stdio.h>

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

Equivalent to *fprintf*, but stores characters in the array pointed to by *s* instead of writing them to a stream. No more than *n* – 1 characters will be written to the array. The string pointed to by *format* specifies how subsequent arguments will be displayed. Stores a null character in the array at the end of output.

Returns Number of characters that would have been stored in the array (not including the null character) had there been no length restriction. Returns a negative value if an encoding error occurs. 22.8

sprintf *Formatted String Write* <stdio.h>

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

Equivalent to *fprintf*, but stores characters in the array pointed to by *s* instead of writing them to a stream. The string pointed to by *format* specifies how subsequent arguments will be displayed. Stores a null character in the array at the end of output.

Returns Number of characters stored in the array, not including the null character. In C99, returns a negative value if an encoding error occurs. 22.8

sqrt *Square Root* <math.h>

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

Returns Nonnegative square root of *x*. A domain error occurs if *x* is negative. 23.3

rand *Seed Pseudo-Random Number Generator* <stdlib.h>

```
void rand(unsigned int seed);
```

Uses *seed* to initialize the sequence of pseudo-random numbers produced by calling *rand*. 26.2

sscanf *Formatted String Read* <stdio.h>

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

Equivalent to *fscanf*, but reads characters from the string pointed to by *s* instead of reading them from a stream. The string pointed to by *format* specifies the format of the items to be read. The arguments that follow *format* point to objects in which the items are to be stored.

<i>Returns</i>	Number of input items successfully read and stored. Returns EOF if an input failure occurs before any items could be read.	22.8
strcat	<i>String Concatenation</i>	<string.h>
	char *strcat(char * restrict s1, const char * restrict s2);	
	Appends characters from the string pointed to by s2 to the string pointed to by s1.	
<i>Returns</i>	s1 (a pointer to the concatenated string).	13.5, 23.6
strchr	<i>Search String for Character</i>	<string.h>
	char *strchr(const char *s, int c);	
<i>Returns</i>	A pointer to the first occurrence of the character c in the string pointed to by s. Returns a null pointer if c isn't found.	23.6
strcmp	<i>String Comparison</i>	<string.h>
	int strcmp(const char *s1, const char *s2);	
<i>Returns</i>	A negative, zero, or positive integer, depending on whether the string pointed to by s1 is less than, equal to, or greater than the string pointed to by s2.	13.5, 23.6
strcoll	<i>String Comparison Using Locale-Specific Collating Sequence</i>	<string.h>
	int strcoll(const char *s1, const char *s2);	
<i>Returns</i>	A negative, zero, or positive integer, depending on whether the string pointed to by s1 is less than, equal to, or greater than the string pointed to by s2. The comparison is performed according to the rules of the current locale's LC_COLLATE category.	23.6
strcpy	<i>String Copy</i>	<string.h>
	char *strcpy(char * restrict s1, const char * restrict s2);	
	Copies the string pointed to by s2 into the array pointed to by s1.	
<i>Returns</i>	s1 (a pointer to the destination).	13.5, 23.6
strcspn	<i>Search String for Initial Span of Characters Not in Set</i>	<string.h>
	size_t strcspn(const char *s1, const char *s2);	
<i>Returns</i>	Length of the longest initial segment of the string pointed to by s1 that doesn't contain any character in the string pointed to by s2.	23.6
strerror	<i>Convert Error Number to String</i>	<string.h>
	char *strerror(int errnum);	
<i>Returns</i>	A pointer to a string containing an error message corresponding to the value of errnum.	24.2

strftime	<i>Write Formatted Date and Time to String</i>	<time.h>
	<pre>size_t strftime(char * restrict s, size_t maxsize, const char * restrict format, const struct tm * restrict timeptr);</pre>	
	Stores characters in the array pointed to by <i>s</i> under control of the string pointed to by <i>format</i> . The format string may contain ordinary characters, which are copied unchanged, and conversion specifiers, which are replaced by values from the structure pointed to by <i>timeptr</i> . The <i>maxsize</i> parameter limits the number of characters (including the null character) that can be stored.	
<i>Returns</i>	Number of characters stored (not including the terminating null character). Returns zero if the number of characters to be stored (including the null character) exceeds <i>maxsize</i> .	26.3
strlen	<i>String Length</i>	<string.h>
	<pre>size_t strlen(const char *s);</pre>	
<i>Returns</i>	Length of the string pointed to by <i>s</i> , not including the null character.	13.5, 23.6
strncat	<i>Bounded String Concatenation</i>	<string.h>
	<pre>char *strncat(char * restrict s1, const char * restrict s2, size_t n);</pre>	
	Appends characters from the array pointed to by <i>s2</i> to the string pointed to by <i>s1</i> . Copying stops when a null character is encountered or <i>n</i> characters have been copied.	
<i>Returns</i>	<i>s1</i> (a pointer to the concatenated string).	13.5, 23.6
strcmp	<i>Bounded String Comparison</i>	<string.h>
	<pre>int strcmp(const char *s1, const char *s2, size_t n);</pre>	
<i>Returns</i>	A negative, zero, or positive integer, depending on whether the first <i>n</i> characters of the array pointed to by <i>s1</i> are less than, equal to, or greater than the first <i>n</i> characters of the array pointed to by <i>s2</i> . Comparison stops if a null character is encountered in either array.	23.6
strncpy	<i>Bounded String Copy</i>	<string.h>
	<pre>char *strncpy(char * restrict s1, const char * restrict s2, size_t n);</pre>	
	Copies the first <i>n</i> characters of the array pointed to by <i>s2</i> into the array pointed to by <i>s1</i> . If it encounters a null character in the array pointed to by <i>s2</i> , <i>strncpy</i> adds null characters to the array pointed to by <i>s1</i> until a total of <i>n</i> characters have been written.	
<i>Returns</i>	<i>s1</i> (a pointer to the destination).	13.5, 23.6

strpbrk	<i>Search String for One of a Set of Characters</i>	<string.h>
	char *strpbrk(const char *s1, const char *s2);	
<i>Returns</i>	A pointer to the leftmost character in the string pointed to by s1 that matches any character in the string pointed to by s2. Returns a null pointer if no match is found.	23.6
strrchr	<i>Search String in Reverse for Character</i>	<string.h>
	char *strrchr(const char *s, int c);	
<i>Returns</i>	A pointer to the last occurrence of the character c in the string pointed to by s. Returns a null pointer if c isn't found.	23.6
strspn	<i>Search String for Initial Span of Characters in Set</i>	<string.h>
	size_t strspn(const char *s1, const char *s2);	
<i>Returns</i>	Length of the longest initial segment in the string pointed to by s1 that consists entirely of characters in the string pointed to by s2.	23.6
strstr	<i>Search String for Substring</i>	<string.h>
	char *strstr(const char *s1, const char *s2);	
<i>Returns</i>	A pointer to the first occurrence in the string pointed to by s1 of the sequence of characters in the string pointed to by s2. Returns a null pointer if no match is found.	23.6
strtod	<i>Convert String to Double</i>	<stdlib.h>
	double strtod(const char * restrict nptr, char ** restrict endptr);	
	Skips white-space characters in the string pointed to by nptr, then converts subsequent characters into a double value. If endptr isn't a null pointer, strtod modifies the object pointed to by endptr so that it points to the first leftover character. If no double value is found, or if it has the wrong form, strtod stores nptr in the object pointed to by endptr. If the number is too large or small to represent, it stores ERANGE in errno. <i>C99 changes:</i> The string pointed to by nptr may contain a hexadecimal floating-point number, infinity, or NaN. Whether ERANGE is stored in errno when the number is too small to represent is implementation-defined.	
<i>Returns</i>	The converted number. Returns zero if no conversion could be performed. If the number is too large to represent, returns plus or minus HUGE_VAL, depending on the number's sign. Returns zero if the number is too small to represent. <i>C99 change:</i> If the number is too small to represent, strtod returns a value whose magnitude is no greater than the smallest normalized positive double.	26.2
strtod	<i>Convert String to Float (C99)</i>	<stdlib.h>
	float strtod(const char * restrict nptr, char ** restrict endptr);	

`strtof` is identical to `strtod`, except that it converts a string to a float value.

Returns The converted number. Returns zero if no conversion could be performed. If the number is too large to represent, returns plus or minus `HUGE_VALF`, depending on the number's sign. If the number is too small to represent, returns a value whose magnitude is no greater than the smallest normalized positive float. 26.2

strtoimax *Convert String to Greatest-Width Integer (C99)* <inttypes.h>

```
intmax_t strtoimax(const char * restrict nptr,
                    char ** restrict endptr, int base);
```

`strtoimax` is identical to `strtol`, except that it converts a string to a value of type `intmax_t` (the widest signed integer type).

Returns The converted number. Returns zero if no conversion could be performed. If the number can't be represented, returns `INTMAX_MAX` or `INTMAX_MIN`, depending on the number's sign. 27.2

strtok *Search String for Token* <string.h>

```
char *strtok(char * restrict s1,
             const char * restrict s2);
```

Searches the string pointed to by `s1` for a "token" consisting of characters not in the string pointed to by `s2`. If a token exists, the character following it is changed to a null character. If `s1` is a null pointer, a search begun by the most recent call of `strtok` is continued; the search begins immediately after the null character at the end of the previous token.

Returns A pointer to the first character of the token. Returns a null pointer if no token could be found. 23.6

strtol *Convert String to Long Integer* <stdlib.h>

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

Skips white-space characters in the string pointed to by `nptr`, then converts subsequent characters into a `long int` value. If `base` is between 2 and 36, it is used as the radix of the number. If `base` is zero, the number is assumed to be decimal unless it begins with 0 (octal) or with 0x or 0X (hexadecimal). If `endptr` isn't a null pointer, `strtol` modifies the object pointed to by `endptr` so that it points to the first leftover character. If no `long int` value is found, or if it has the wrong form, `strtol` stores `nptr` in the object pointed to by `endptr`. If the number can't be represented, it stores `ERANGE` in `errno`.

Returns The converted number. Returns zero if no conversion could be performed. If the number can't be represented, returns `LONG_MAX` or `LONG_MIN`, depending on the number's sign. 26.2

strtold *Convert String to Long Double (C99)* <stdlib.h>

```
long double strtold(const char * restrict nptr,
                     char ** restrict endptr);
```

`strtold` is identical to `strtod`, except that it converts a string to a long double value.

Returns The converted number. Returns zero if no conversion could be performed. If the number is too large to represent, returns plus or minus `HUGE_VAL`, depending on the number's sign. If the number is too small to represent, returns a value whose magnitude is no greater than the smallest normalized positive long double. 26.2

strtol *Convert String to Long Long Integer (C99)* <stdlib.h>

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

`strtol` is identical to `strtol`, except that it converts a string to a long long int value.

Returns The converted number. Returns zero if no conversion could be performed. If the number can't be represented, returns `LLONG_MAX` or `LLONG_MIN`, depending on the number's sign. 26.2

strtoul *Convert String to Unsigned Long Integer* <stdlib.h>

```
unsigned long int strtoul(const char * restrict nptr,
                          char ** restrict endptr,
                          int base);
```

`strtoul` is identical to `strtol`, except that it converts a string to an unsigned long int value.

Returns The converted number. Returns zero if no conversion could be performed. If the number can't be represented, returns `ULONG_MAX`. 26.2

strtoull *Convert String to Unsigned Long Long Integer (C99)* <stdlib.h>

```
unsigned long long int strtoull(
    const char * restrict nptr,
    char ** restrict endptr, int base);
```

`strtoull` is identical to `strtol`, except that it converts a string to an unsigned long long int value.

Returns The converted number. Returns zero if no conversion could be performed. If the number can't be represented, returns `ULLONG_MAX`. 26.2

strtoumax *Convert String to Unsigned Greatest-Width Integer (C99)* <inttypes.h>

```
uintmax_t strtoumax(const char * restrict nptr,
                     char ** restrict endptr,
                     int base);
```

`strtoumax` is identical to `strtol`, except that it converts a string to a value of type `uintmax_t` (the widest unsigned integer type).

Returns The converted number. Returns zero if no conversion could be performed. If the number can't be represented, returns `UINTMAX_MAX`. 27.2

strxfrm	<i>Transform String</i>	<string.h>
	<pre>size_t strxfrm(char * restrict s1, const char * restrict s2, size_t n);</pre>	
	Transforms the string pointed to by <i>s2</i> , placing the first <i>n</i> characters of the result—including the null character—in the array pointed to by <i>s1</i> . Calling <i>strcmp</i> with two transformed strings should produce the same outcome (negative, zero, or positive) as calling <i>strcoll</i> with the original strings. If <i>n</i> is zero, <i>s1</i> is allowed to be a null pointer.	
<i>Returns</i>	Length of the transformed string. If this value is <i>n</i> or more, the contents of the array pointed to by <i>s1</i> are indeterminate.	23.6
swprintf	<i>Wide-Character Formatted String Write (C99)</i>	<wchar.h>
	<pre>int swprintf(wchar_t * restrict s, size_t n, const wchar_t * restrict format, ...);</pre>	
	Equivalent to <i>fwprintf</i> , but stores wide characters in the array pointed to by <i>s</i> instead of writing them to a stream. The string pointed to by <i>format</i> specifies how subsequent arguments will be displayed. No more than <i>n</i> wide characters will be written to the array, including a terminating null wide character.	
<i>Returns</i>	Number of wide characters stored in the array, not including the null wide character. Returns a negative value if an encoding error occurs or the number of wide characters to be written is <i>n</i> or more.	25.5
swscanf	<i>Wide-Character Formatted String Read (C99)</i>	<wchar.h>
	<pre>int swscanf(const wchar_t * restrict s, const wchar_t * restrict format, ...);</pre>	
	Wide-character version of <i>sscanf</i> .	25.5
system	<i>Perform Operating-System Command</i>	<stdlib.h>
	<pre>int system(const char *string);</pre>	
	Passes the string pointed to by <i>string</i> to the operating system's command processor (shell) to be executed. Program termination may occur as a result of executing the command.	
<i>Returns</i>	If <i>string</i> is a null pointer, returns a nonzero value if a command processor is available. If <i>string</i> isn't a null pointer, <i>system</i> returns an implementation-defined value (if it returns at all).	26.2
tan	<i>Tangent</i>	<math.h>
	<pre>double tan(double x);</pre>	
tanf	<pre>float tanf(float x);</pre>	
tanl	<pre>long double tanl(long double x);</pre>	
<i>Returns</i>	Tangent of <i>x</i> (measured in radians).	23.3

tanh	<i>Hyperbolic Tangent</i>	<math.h>
	double tanh(double x);	
tanhf	float tanhf(float x);	
tanhl	long double tanhl(long double x);	
<i>Returns</i>	Hyperbolic tangent of x.	23.3
tgamma	<i>Gamma Function (C99)</i>	<math.h>
	double tgamma(double x);	
tgammaf	float tgammaf(float x);	
tgammal	long double tgammal(long double x);	
<i>Returns</i>	$\Gamma(x)$, where Γ is the gamma function. A domain error or range error may occur if x is a negative integer or zero. A range error may occur if the magnitude of x is too large or too small.	23.4
time	<i>Current Time</i>	<time.h>
	time_t time(time_t *timer);	
<i>Returns</i>	Current calendar time. Returns (time_t) (-1) if the calendar time isn't available. If timer isn't a null pointer, also stores the return value in the object pointed to by timer.	26.3
tmpfile	<i>Create Temporary File</i>	<stdio.h>
	FILE *tmpfile(void);	
	Creates a temporary file that will automatically be removed when it's closed or the program ends. Opens the file in "wb+" mode.	
<i>Returns</i>	A file pointer to be used when performing subsequent operations on the file. Returns a null pointer if a temporary file can't be created.	22.2
tmpnam	<i>Generate Temporary File Name</i>	<stdio.h>
	char *tmpnam(char *s);	
	Generates a name for a temporary file. If s is a null pointer, tmpnam stores the file name in a static object. Otherwise, it copies the file name into the character array pointed to by s. (The array must be long enough to store L_tmpnam characters.)	
<i>Returns</i>	A pointer to the file name. Returns a null pointer if a file name can't be generated.	22.2
tolower	<i>Convert to Lower Case</i>	<ctype.h>
	int tolower(int c);	
<i>Returns</i>	If c is an upper-case letter, returns the corresponding lower-case letter. If c isn't an upper-case letter, returns c unchanged.	23.5
toupper	<i>Convert to Upper Case</i>	<ctype.h>
	int toupper(int c);	

Returns If *c* is a lower-case letter, returns the corresponding upper-case letter. If *c* isn't a lower-case letter, returns *c* unchanged. 23.5

towctrans *Transliterate Wide Character (C99)* <wctype.h>

wint_t towctrans(wint_t wc, wctrans_t desc);

Returns Mapped value of *wc* using the mapping described by *desc*. (*desc* must be a value returned by a call of *wctrans*; the current setting of the *LC_CTYPE* category must be the same during both calls.) 25.6

towlower *Convert Wide Character to Lower Case (C99)* <wctype.h>

wint_t towlower(wint_t wc);

Returns If *iswupper(wc)* is true, returns a corresponding wide character for which *iswlower* is true in the current locale, if such a character exists. Otherwise, returns *wc* unchanged. 25.6

towupper *Convert Wide Character to Upper Case (C99)* <wctype.h>

wint_t towupper(wint_t wc);

Returns If *iswlower(wc)* is true, returns a corresponding wide character for which *iswupper* is true in the current locale, if such a character exists. Otherwise, returns *wc* unchanged. 25.6

trunc *Truncate to Nearest Integral Value (C99)* <math.h>

double trunc(double x);

truncf *float truncf(float x);*

truncl *long double truncl(long double x);*

Returns *x* rounded to the integer (in floating-point format) nearest to it but no larger in magnitude. 23.4

ungetc *Unread Character* <stdio.h>

*int ungetc(int c, FILE *stream);*

Pushes the character *c* back onto the stream pointed to by *stream* and clears the stream's end-of-file indicator. The number of characters that can be pushed back by consecutive calls of *ungetc* varies; only the first call is guaranteed to succeed. Calling a file positioning function (*fseek*, *fsetpos*, or *rewind*) causes the pushed-back character(s) to be lost.

Returns *c* (the pushed-back character). Returns *EOF* if an attempt is made to push back *EOF* or to push back too many characters without a read or file positioning operation. 22.4

ungetwc *Unread Wide Character (C99)* <wchar.h>

*wint_t ungetwc(wint_t c, FILE *stream);*

Wide-character version of *ungetc*. 25.5

va_arg	<i>Fetch Argument from Variable Argument List</i>	<stdarg.h>
	<i>type va_arg(va_list ap, type);</i>	<i>macro</i>
	Fetches an argument in the variable argument list associated with ap, then modifies ap so that the next use of va_arg fetches the following argument. ap must have been initialized by va_start (or va_copy in C99) prior to the first use of va_arg.	
<i>Returns</i>	Value of the argument, assuming that its type (after the default argument promotions have been applied) is compatible with <i>type</i> .	26.1
va_copy	<i>Copy Variable Argument List (C99)</i>	<stdarg.h>
	<i>void va_copy(va_list dest, va_list src);</i>	<i>macro</i>
	Copies src into dest. The value of dest will be the same as if va_start had been applied to dest followed by the same sequence of va_arg applications that was used to reach the present state of src.	26.1
va_end	<i>End Processing of Variable Argument List</i>	<stdarg.h>
	<i>void va_end(va_list ap);</i>	<i>macro</i>
	Ends the processing of the variable argument list associated with ap.	26.1
va_start	<i>Start Processing of Variable Argument List</i>	<stdarg.h>
	<i>void va_start(va_list ap, parmN);</i>	<i>macro</i>
	Must be invoked before accessing a variable argument list. Initializes ap for later use by va_arg and va_end. <i>parmN</i> is the name of the last ordinary parameter (the one followed by , . . .).	26.1
vfprintf	<i>Formatted File Write Using Variable Argument List</i>	<stdio.h>
	<i>int vfprintf(FILE * restrict stream, const char * restrict format, va_list arg);</i>	
	Equivalent to fprintf with the variable argument list replaced by arg.	
<i>Returns</i>	Number of characters written. Returns a negative value if an error occurs.	26.1
vfscanf	<i>Formatted File Read Using Variable Argument List (C99)</i>	<stdio.h>
	<i>int vfscanf(FILE * restrict stream, const char * restrict format, va_list arg);</i>	
	Equivalent to fscanf with the variable argument list replaced by arg.	
<i>Returns</i>	Number of input items successfully read and stored. Returns EOF if an input failure occurs before any items can be read.	26.1
vfwprintf	<i>Wide-Character Formatted File Write Using Variable Argument List (C99)</i>	<wchar.h>

```
int vfwprintf(FILE * restrict stream,
               const wchar_t * restrict format,
               va_list arg);
```

Wide-character version of vfprintf.

25.5

vfwscanf	<i>Wide-Character Formatted File Read Using Variable Argument List (C99)</i>	<wchar.h>
-----------------	--	-----------

```
int vfwscanf(FILE * restrict stream,
              const wchar_t * restrict format,
              va_list arg);
```

Wide-character version of vfscanf.

25.5

vprintf	<i>Formatted Write Using Variable Argument List</i>	<stdio.h>
----------------	---	-----------

```
int vprintf(const char * restrict format, va_list arg);
```

Equivalent to printf with the variable argument list replaced by arg.

<i>Returns</i>	Number of characters written. Returns a negative value if an error occurs.	26.1
----------------	--	------

vscanf	<i>Formatted Read Using Variable Argument List (C99)</i>	<stdio.h>
---------------	--	-----------

```
int vscanf(const char * restrict format, va_list arg);
```

Equivalent to scanf with the variable argument list replaced by arg.

<i>Returns</i>	Number of input items successfully read and stored. Returns EOF if an input failure occurs before any items can be read.	26.1
----------------	--	------

vsnprintf	<i>Bounded Formatted String Write Using Variable Argument List (C99)</i>	<stdio.h>
------------------	--	-----------

```
int vsnprintf(char * restrict s, size_t n,
              const char * restrict format,
              va_list arg);
```

Equivalent to sprintf with the variable argument list replaced by arg.

<i>Returns</i>	Number of characters that would have been stored in the array pointed to by s (not including the null character) had there been no length restriction. Returns a negative value if an encoding error occurs.	26.1
----------------	--	------

vsprintf	<i>Formatted String Write Using Variable Argument List</i>	<stdio.h>
-----------------	--	-----------

```
int vsprintf(char * restrict s,
            const char * restrict format,
            va_list arg);
```

Equivalent to sprintf with the variable argument list replaced by arg.

<i>Returns</i>	Number of characters stored in the array pointed to by s, not including the null character. In C99, returns a negative value if an encoding error occurs.	26.1
----------------	---	------

vsscanf	<i>Formatted String Read Using Variable Argument List (C99)</i>	<stdio.h>
	<pre>int vsscanf(const char * restrict s, const char * restrict format, va_list arg);</pre>	
	Equivalent to sscanf with the variable argument list replaced by arg.	
<i>Returns</i>	Number of input items successfully read and stored. Returns EOF if an input failure occurs before any items can be read.	26.1
vswprintf	<i>Wide-Character Formatted String Write Using Variable Argument List (C99)</i>	<wchar.h>
	<pre>int vswprintf(wchar_t * restrict s, size_t n, const wchar_t * restrict format, va_list arg);</pre>	
	Equivalent to swprintf with the variable argument list replaced by arg.	
<i>Returns</i>	Number of wide characters stored in the array pointed to by s, not including the null wide character. Returns a negative value if an encoding error occurs or the number of wide characters to be written is n or more.	25.5
vswscanf	<i>Wide-Character Formatted String Read Using Variable Argument List (C99)</i>	<wchar.h>
	<pre>int vswscanf(const wchar_t * restrict s, const wchar_t * restrict format, va_list arg);</pre>	
	Wide-character version of vsscanf.	25.5
vwprintf	<i>Wide-Character Formatted Write Using Variable Argument List (C99)</i>	<wchar.h>
	<pre>int vwprintf(const wchar_t * restrict format, va_list arg);</pre>	
	Wide-character version of vprintf.	25.5
vwscanf	<i>Wide-Character Formatted Read Using Variable Argument List (C99)</i>	<wchar.h>
	<pre>int vwscanf(const wchar_t * restrict format, va_list arg);</pre>	
	Wide-character version of vscanf.	25.5
wcrtomb	<i>Convert Wide Character to Multibyte Character – Restartable (C99)</i>	<wchar.h>
	<pre>size_t wcrtomb(char * restrict s, wchar_t wc, mbstate_t * restrict ps);</pre>	
	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 (possibly including shift sequences), which it stores in the array pointed to by s. The value of ps should be a pointer to an object of type mbstate_t that contains the current conversion state. If ps is a null pointer, wcrtomb uses an internal object to store the conversion state. If wc is a null wide character, wcrtomb stores a null byte, preceded by a shift sequence if necessary to restore the initial shift state, and the mbstate_t object used during the call is left in the initial conversion state.

Returns Number of bytes stored in the array, including shift sequences. If wc isn't a valid wide character, returns (size_t) (-1) and stores EILSEQ in errno. 25.5

wcscat	<i>Wide-String Concatenation (C99)</i>	<wchar.h>
	<i>wchar_t *wcscat(wchar_t * restrict s1, const wchar_t * restrict s2);</i>	
	Wide-character version of strcat.	25.5
wcschr	<i>Search Wide String for Character (C99)</i>	<wchar.h>
	<i>wchar_t *wcschr(const wchar_t *s, wchar_t c);</i>	
	Wide-character version of strchr.	25.5
wcsncmp	<i>Wide-String Comparison (C99)</i>	<wchar.h>
	<i>int wcsncmp(const wchar_t *s1, const wchar_t *s2);</i>	
	Wide-character version of strcmp.	25.5
wcscoll	<i>Wide-String Comparison Using Locale-Specific Collating Sequence (C99)</i>	<wchar.h>
	<i>int wcscoll(const wchar_t *s1, const wchar_t *s2);</i>	
	Wide-character version of strcoll.	25.5
wcscpy	<i>Wide-String Copy (C99)</i>	<wchar.h>
	<i>wchar_t *wcscpy(wchar_t * restrict s1, const wchar_t * restrict s2);</i>	
	Wide-character version of strcpy.	25.5
wcscspn	<i>Search Wide String for Initial Span of Characters Not in Set (C99)</i>	<wchar.h>
	<i>size_t wcscspn(const wchar_t *s1, const wchar_t *s2);</i>	
	Wide-character version of strcspn.	25.5
wcsftime	<i>Write Formatted Date and Time to Wide String (C99)</i>	<wchar.h>
	<i>size_t wcsftime(wchar_t * restrict s, size_t maxsize, const wchar_t * restrict format, const struct tm * restrict timeptr);</i>	
	Wide-character version of strftime.	25.5

wcslen	<i>Wide-String Length (C99)</i>	<wchar.h>
	<code>size_t wcslen(const wchar_t *s);</code>	
	Wide-character version of <code>strlen</code> .	25.5
wcsncat	<i>Bounded Wide-String Concatenation (C99)</i>	<wchar.h>
	<code>wchar_t *wcsncat(wchar_t * restrict s1,</code> <code> const wchar_t * restrict s2,</code> <code> size_t n);</code>	
	Wide-character version of <code>strncat</code> .	25.5
wcsncmp	<i>Bounded Wide-String Comparison (C99)</i>	<wchar.h>
	<code>int wcsncmp(const wchar_t *s1, const wchar_t *s2,</code> <code> size_t n);</code>	
	Wide-character version of <code>strncmp</code> .	25.5
wcsncpy	<i>Bounded Wide-String Copy (C99)</i>	<wchar.h>
	<code>wchar_t *wcsncpy(wchar_t * restrict s1,</code> <code> const wchar_t * restrict s2,</code> <code> size_t n);</code>	
	Wide-character version of <code>strncpy</code> .	25.5
wcspbrk	<i>Search Wide String for One of a Set of Characters (C99)</i>	<wchar.h>
	<code>wchar_t *wcspbrk(const wchar_t *s1,</code> <code> const wchar_t *s2);</code>	
	Wide-character version of <code>strpbrk</code> .	25.5
wcsrchr	<i>Search Wide String in Reverse for Character (C99)</i>	<wchar.h>
	<code>wchar_t *wcsrchr(const wchar_t *s, wchar_t c);</code>	
	Wide-character version of <code>strrchr</code> .	25.5
wcsrtombs	<i>Convert Wide String to Multibyte String – Restartable (C99)</i>	<wchar.h>
	<code>size_t wcsrtombs(char * restrict dst,</code> <code> const wchar_t ** restrict src,</code> <code> size_t len,</code> <code> mbstate_t * restrict ps);</code>	
	Converts a sequence of wide characters from the array indirectly pointed to by <code>src</code> into a sequence of corresponding multibyte characters that begins in the conversion state described by the object pointed to by <code>ps</code> . If <code>ps</code> is a null pointer, <code>wcsrtombs</code> uses an internal object to store the conversion state. If <code>dst</code> isn't a null pointer, the converted characters are then stored in the array pointed to by <code>dst</code> . Conversion continues up to and including a terminating null wide character, which is also stored. Conversion stops earlier if a wide character is reached that doesn't correspond to a valid multibyte character or—if <code>dst</code> isn't a null pointer—	

when the next multibyte character would exceed the limit of *len* total bytes to be stored in the array pointed to by *dst*. If *dst* isn't a null pointer, the object pointed to by *src* is assigned either a null pointer (if a terminating null wide character was reached) or the address just past the last wide character converted (if any). If the conversion ends at a null wide character, the resulting state is the initial conversion state.

Returns Number of bytes in the resulting multibyte character sequence, not including any terminating null character. Returns (*size_t*) (-1) and stores EILSEQ in *errno* if a wide character is encountered that doesn't correspond to a valid multibyte character. 25.5

wcsspn *Search Wide String for Initial Span of Characters in Set (C99)* <wchar.h>
*size_t wcsspn(const wchar_t *s1, const wchar_t *s2);*
 Wide-character version of *strspn*. 25.5

wcsstr *Search Wide String for Substring (C99)* <wchar.h>
*wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2);*
 Wide-character version of *strstr*. 25.5

wcstod *Convert Wide String to Double (C99)* <wchar.h>
*double wcstod(const wchar_t * restrict nptr,
 wchar_t ** restrict endptr);*
 Wide-character version of *strtod*. 25.5

wcstof *Convert Wide String to Float (C99)* <wchar.h>
*float wcstof(const wchar_t * restrict nptr,
 wchar_t ** restrict endptr);*
 Wide-character version of *strtod*. 25.5

wcstoiimax *Convert Wide String to Greatest-Width Integer (C99)* <inttypes.h>
*intmax_t wcstoiimax(const wchar_t * restrict nptr,
 wchar_t ** restrict endptr,
 int base);*
 Wide-character version of *strtoiimax*. 27.2

wcstok *Search Wide String for Token (C99)* <wchar.h>
*wchar_t *wcstok(wchar_t * restrict s1,
 const wchar_t * restrict s2,
 wchar_t ** restrict ptr);*

Searches the wide string pointed to by *s1* for a "token" consisting of wide characters not in the wide string pointed to by *s2*. If a token exists, the character following it is changed to a null wide character. If *s1* is a null pointer, a search begun by a previous call of *wcstok* is continued; the search begins immediately after the null wide character at the end of the previous token. *ptr* points to an object of

type `wchar_t *` that `wcstok` modifies to keep track of its progress. If `s1` is a null pointer, this object must be the same one used in a previous call of `wcstok`; it determines which wide string is to be searched and where the search is to begin.

Returns A pointer to the first wide character of the token. Returns a null pointer if no token could be found. 25.5

`wcstol` *Convert Wide String to Long Integer (C99)* `<wchar.h>`
`long int wcstol(const wchar_t * restrict nptr,`
`wchar_t ** restrict endptr, int base);`
 Wide-character version of `strtol`. 25.5

`wcstold` *Convert Wide String to Long Double (C99)* `<wchar.h>`
`long double wcstold(const wchar_t * restrict nptr,`
`wchar_t ** restrict endptr);`
 Wide-character version of `strtold`. 25.5

`wcstoll` *Convert Wide String to Long Long Integer (C99)* `<wchar.h>`
`long long int wcstoll(const wchar_t * restrict nptr,`
`wchar_t ** restrict endptr,`
`int base);`
 Wide-character version of `strtoll`. 25.5

`wcstombs` *Convert Wide String to Multibyte String* `<stdlib.h>`
`size_t wcstombs(char * restrict s,`
`const wchar_t * restrict pwcs,`
`size_t n);`
 Converts a sequence of wide characters into corresponding multibyte characters. `pwcs` points to an array containing the wide characters. The multibyte characters are stored in the array pointed to by `s`. Conversion ends if a null character is stored or if storing a multibyte character would exceed the limit of `n` bytes.

Returns Number of bytes stored, not including the terminating null character, if any. Returns `(size_t) (-1)` if a wide character is encountered that doesn't correspond to a valid multibyte character. 25.2

`wcstoul` *Convert Wide String to Unsigned Long Integer (C99)* `<wchar.h>`
`unsigned long int wcstoul(`
`const wchar_t * restrict nptr,`
`wchar_t ** restrict endptr, int base);`
 Wide-character version of `strtoul`. 25.5

`wcstoull` *Convert Wide String to Unsigned Long Long Integer (C99)* `<wchar.h>`
`unsigned long long int wcstoull(`
`const wchar_t * restrict nptr,`
`wchar_t ** restrict endptr, int base);`
 Wide-character version of `strtoull`. 25.5

wcstoumax	<i>Convert Wide String to Unsigned Greatest-Width Integer</i> <inttypes.h> (C99)	
	<pre>uintmax_t wcstoumax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);</pre>	
	Wide-character version of strtoumax.	27.2
wcsxfrm	<i>Transform Wide String</i> (C99)	<wchar.h>
	<pre>size_t wcsxfrm(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);</pre>	
	Wide-character version of strxfrm.	25.5
wctob	<i>Convert Wide Character to Byte</i> (C99)	<wchar.h>
	<pre>int wctob(wint_t c);</pre>	
<i>Returns</i>	Single-byte representation of c as an unsigned char converted to int. Returns EOF if c doesn't correspond to one multibyte character in the initial shift state.	
		25.5
wctomb	<i>Convert Wide Character to Multibyte Character</i>	<stdlib.h>
	<pre>int wctomb(char *s, wchar_t wc);</pre>	
	Converts the wide character stored in wc into a multibyte character. If s isn't a null pointer, stores the result in the array that s points to.	
<i>Returns</i>	If s is a null pointer, returns a nonzero or zero value, depending on whether or not multibyte characters have state-dependent encodings. Otherwise, returns the number of bytes in the multibyte character that corresponds to wc; returns -1 if wc doesn't correspond to a valid multibyte character.	25.2
wctrans	<i>Define Wide-Character Mapping</i> (C99)	<wctype.h>
	<pre>wctrans_t wctrans(const char *property);</pre>	
<i>Returns</i>	If property identifies a valid mapping of wide characters according to the LC_CTYPE category of the current locale, returns a nonzero value that can be used as the second argument to the towctrans function; otherwise, returns zero.	
		25.6
wctype	<i>Define Wide-Character Class</i> (C99)	<wctype.h>
	<pre>wctype_t wctype(const char *property);</pre>	
<i>Returns</i>	If property identifies a valid class of wide characters according to the LC_CTYPE category of the current locale, returns a nonzero value that can be used as the second argument to the iswctype function; otherwise, returns zero.	25.6
wmemchr	<i>Search Wide-Character Memory Block for Character</i> (C99)	<wchar.h>
	<pre>wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);</pre>	
	Wide-character version of memchr.	25.5

wmemcmp	<i>Compare Wide-Character Memory Blocks (C99)</i>	<wchar.h>
	<i>int wmemcmp(const wchar_t * s1, const wchar_t * s2, size_t n);</i>	
	Wide-character version of memcmp.	25.5
wmemcpy	<i>Copy Wide-Character Memory Block (C99)</i>	<wchar.h>
	<i>wchar_t *wmemcpy(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);</i>	
	Wide-character version of memcpy.	25.5
wmemmove	<i>Copy Wide-Character Memory Block (C99)</i>	<wchar.h>
	<i>wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);</i>	
	Wide-character version of memmove.	25.5
wmemset	<i>Initialize Wide-Character Memory Block (C99)</i>	<wchar.h>
	<i>wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);</i>	
	Wide-character version of memset.	25.5
wprintf	<i>Wide-Character Formatted Write (C99)</i>	<wchar.h>
	<i>int wprintf(const wchar_t * restrict format, ...);</i>	
	Wide-character version of printf.	25.5
wscanf	<i>Wide-Character Formatted Read (C99)</i>	<wchar.h>
	<i>int wscanf(const wchar_t * restrict format, ...);</i>	
	Wide-character version of scanf.	25.5

APPENDIX E

ASCII Character Set

Escape Sequence								
Decimal	Oct	Hex	Char	Character				
0	\0	\x00		<i>nul</i>	32	!	64	@
1	\1	\x01		<i>soh</i> (^A)	33	"	65	A
2	\2	\x02		<i>stx</i> (^B)	34	#	66	B
3	\3	\x03		<i>etx</i> (^C)	35	\$	67	C
4	\4	\x04		<i>eot</i> (^D)	36	%	68	D
5	\5	\x05		<i>enq</i> (^E)	37	&	69	E
6	\6	\x06		<i>ack</i> (^F)	38	'	70	F
7	\7	\x07	\a	<i>bel</i> (^G)	39	(71	G
8	\10	\x08	\b	<i>bs</i> (^H)	40)	72	H
9	\11	\x09	\t	<i>ht</i> (^I)	41	*	73	I
10	\12	\x0a	\n	<i>lf</i> (^J)	42	+	74	J
11	\13	\x0b	\v	<i>vt</i> (^K)	43	,	75	K
12	\14	\x0c	\f	<i>ff</i> (^L)	44	-	76	L
13	\15	\x0d	\r	<i>cr</i> (^M)	45	.	77	M
14	\16	\x0e		<i>so</i> (^N)	46	/	78	N
15	\17	\x0f		<i>si</i> (^O)	47	0	79	O
16	\20	\x10		<i>dle</i> (^P)	48	1	80	P
17	\21	\x11		<i>dc1</i> (^Q)	49	2	81	Q
18	\22	\x12		<i>dc2</i> (^R)	50	3	82	R
19	\23	\x13		<i>dc3</i> (^S)	51	4	83	S
20	\24	\x14		<i>dc4</i> (^T)	52	5	84	T
21	\25	\x15		<i>nak</i> (^U)	53	6	85	U
22	\26	\x16		<i>syn</i> (^V)	54	7	86	V
23	\27	\x17		<i>etb</i> (^W)	55	8	87	W
24	\30	\x18		<i>can</i> (^X)	56	9	88	X
25	\31	\x19		<i>em</i> (^Y)	57	:	89	Y
26	\32	\x1a		<i>sub</i> (^Z)	58	;	90	Z
27	\33	\x1b		<i>esc</i>	59	<	91	[
28	\34	\x1c		<i>fs</i>	60	=	92	\
29	\35	\x1d		<i>gs</i>	61	>	93]
30	\36	\x1e		<i>rs</i>	62	?	94	^
31	\37	\x1f		<i>us</i>	63	—	95	~
								del

BIBLIOGRAPHY

The best book on programming for the layman is "Alice in Wonderland"; but that's because it's the best book on anything for the layman.

C Programming

Feuer, A. R., *The C Puzzle Book*, Revised Printing, Addison-Wesley, Reading, Mass., 1999. Contains numerous “puzzles”—small C programs whose output the reader is asked to predict. The book shows the correct output of each program and provides a detailed explanation of how it works. Good for testing your C knowledge and reviewing the fine points of the language.

Harbison, S. P., III, and G. L. Steele, Jr., *C: A Reference Manual*, Fifth Edition, Prentice-Hall, Upper Saddle River, N.J., 2002. The ultimate C reference—essential reading for the would-be C expert. Covers both C89 and C99 in considerable detail, with frequent discussions of implementation differences found in C compilers. Not a tutorial—assumes that the reader is already well versed in C.

Kernighan, B. W., and D. M. Ritchie, *The C Programming Language*, Second Edition, Prentice-Hall, Englewood Cliffs, N.J., 1988. The original C book, affectionately known as K&R or simply “the White Book.” Includes both a tutorial and a complete C reference manual. The second edition reflects the changes made in C89.

Koenig, A., *C Traps and Pitfalls*, Addison-Wesley, Reading, Mass., 1989. An excellent compendium of common (and some not-so-common) C pitfalls. Forewarned is forearmed.

Plauger, P. J., *The Standard C Library*, Prentice-Hall, Englewood Cliffs, N.J., 1992. Not only explains all aspects of the C89 standard library, but provides complete source code! There’s no better way to learn the library than to study this book. Even if your interest in the library is minimal, the book is worth getting just for the opportunity to study C code written by a master.

- Ritchie, D. M., The development of the C programming language, in *History of Programming Languages II*, edited by T. J. Bergin, Jr., and R. G. Gibson, Jr., Addison-Wesley, Reading, Mass., 1996, pages 671–687. A brief history of C written by the language's designer for the Second ACM SIGPLAN History of Programming Languages Conference, which was held in 1993. The article is followed by transcripts of Ritchie's presentation at the conference and the question-and-answer session with the audience.
- Ritchie, D. M., S. C. Johnson, M. E. Lesk, and B. W. Kernighan, UNIX time-sharing system: the C programming language, *Bell System Technical Journal* 57, 6 (July–August 1978), 1991–2019. A famous article that discusses the origins of C and describes the language as it looked in 1978.
- Rosler, L., The UNIX system: the evolution of C—past and future, *AT&T Bell Laboratories Technical Journal* 63, 8 (October 1984), 1685–1699. Traces the evolution of C from 1978 to 1984 and beyond.
- Summit, S., *C Programming FAQs: Frequently Asked Questions*, Addison-Wesley, Reading, Mass., 1996. An expanded version of the FAQ list that has appeared for years in the Usenet *comp.lang.c* newsgroup.
- van der Linden, P., *Expert C Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1994. Written by one of the C wizards at Sun Microsystems, this book manages to entertain and inform in equal amounts. With its profusion of anecdotes and jokes, it makes learning the fine points of C seem almost fun.

UNIX Programming

Rochkind, M. J., *Advanced UNIX Programming*, Second Edition, Addison-Wesley, Boston, Mass., 2004. Covers UNIX system calls in considerable detail. This book, along with the one by Stevens and Rago, is a must-have for C programmers who use the UNIX operating system or one of its variants.

Stevens, W. R., and S. A. Rago, *Advanced Programming in the UNIX Environment*, Second Edition, Addison-Wesley, Upper Saddle River, N.J., 2005. An excellent follow-up to this book for programmers working under the UNIX operating system. Focuses on using UNIX system calls, including standard C library functions as well as functions that are specific to UNIX.

Programming in General

Bentley, J., *Programming Pearls*, Second Edition, Addison-Wesley, Reading, Mass., 2000. This updated version of Bentley's classic programming book emphasizes writing efficient programs, but touches on other topics that are crucial for the professional programmer. The author's light touch makes the book as enjoyable to read as it is informative.

Kernighan, B. W., and R. Pike, *The Practice of Programming*, Addison-Wesley, Reading, Mass., 1999. Read this book for advice on programming style, choosing the right algorithm, testing and debugging, and writing portable programs. Examples are drawn from C, C++, and Java.

McConnell, S., *Code Complete*, Second Edition, Microsoft Press, Redmond, Wash., 2004. Tries to bridge the gap between programming theory and practice by providing down-to-earth coding advice based on proven research. Includes plenty of examples in a variety of programming languages. Highly recommended.

Raymond, E. S., ed., *The New Hacker's Dictionary*, Third Edition, MIT Press, Cambridge, Mass., 1996. Explains much of the jargon that programmers use, and it's great fun to read as well.

Web Resources

ANSI eStandards Store (webstore.ansi.org). The C99 standard (ISO/IEC 9899:1999) can be purchased at this site. Each set of corrections to the standard (known as a Technical Corrigendum) can be downloaded for free.

comp.lang.c Frequently Asked Questions (c-faq.com). Steve Summit's FAQ list for the *comp.lang.c* newsgroup is a must-read for any C programmer.

Dinkumware (www.dinkumware.com). Dinkumware is owned by P. J. Plauger, the acknowledged master of the C and C++ standard libraries. The web site includes a handy C99 library reference, among other things.

Google Groups (groups.google.com). One of the best ways to find answers to programming questions is to search the Usenet newsgroups using the Google Groups search engine. If you have a question, it's likely that someone else has already asked the question on a newsgroup and the answer has been posted. Groups of particular interest to C programmers include *alt.comp.lang.learn.c-c++* (for C and C++ beginners), *comp.lang.c* (the primary C language group), and *comp.std.c* (devoted to discussion of the C standard).

International Obfuscated C Code Contest (www.ioccc.org). Home of an annual contest in which participants vie to see who can write the most obscure C programs.

ISO/IEC JTC1/SC22/WG14 (www.open-std.org/jtc1/sc22/wg14/). The official web site of WG14, the international working group that created the C99 standard and is responsible for updating it. Of particular interest among the many documents available at the site is the rationale for C99, which explains the reasons for the changes made in the standard.

Lysator (www.lysator.liu.se/c/). A collection of links to C-related web sites maintained by Lysator, an academic computer society located at Sweden's Linköping University.

K.N.KING

C PROGRAMMING

A Modern Approach

SECOND EDITION

A clear, complete, and engaging presentation of the C programming language—now with coverage of both C89 and C99

The first edition of *C Programming: A Modern Approach* was a hit with students and faculty alike because of its clarity and comprehensiveness as well as its trademark Q&A sections. King's spiral approach made the first edition accessible to a broad range of readers, from beginners to more advanced students. The first edition was used at over 225 colleges, making it one of the leading C textbooks of the last ten years.

FEATURES OF THE SECOND EDITION

- Complete coverage of both the C89 standard and the C99 standard, with all C99 changes clearly marked
- Includes a quick reference to all C89 and C99 library functions
- Expanded coverage of GCC
- New coverage of abstract data types
- Updated to reflect today's CPUs and operating systems
- Nearly 500 exercises and programming projects—60% more than in the first edition
- Source code and solutions to selected exercises and programming projects for students, available at the author's website (knking.com)
- Password-protected instructor site (also at knking.com) containing solutions to the remaining exercises and projects, plus PowerPoint presentations for most chapters

"I thoroughly enjoyed reading the second edition of *C Programming* and I look forward to using it in future courses."

— Karen Reid, Senior Lecturer, Department of Computer Science, University of Toronto

"The second edition of King's *C Programming* improves on an already impressive base, and is the book I recommend to anyone who wants to learn C."

— Peter Seebach, moderator, *comp.lang.c.moderated*

"I assign *C Programming* to first-year engineering students. It is concise, clear, accessible to the beginner, and yet also covers all aspects of the language."

— Professor Markus Bussmann, Department of Mechanical and Industrial Engineering, University of Toronto

K. N. KING (Ph.D., University of California, Berkeley) is an associate professor of computer science at Georgia State University. He is also the author of *Modula-2: A Complete Guide* and *Java Programming: From the Beginning*.

ISBN 978-0-393-97950-3



9 0000 >
9 780393 979503

EAN

Cover design by Joan Greenfield

Cover art: László Moholy-Nagy. AXI. Westfälisches
Landesmuseum, Münster, Westphalia, Germany
Photo: Erich Lessing / Art Resource, NY
Copyright: © ARS, NY



W. W. NORTON
NEW YORK • LONDON

www.wwnorton.com