Ignore all characters that aren't letters. Use integer variables to keep track of positions in the array.

(b) Revise the program to use pointers instead of integers to keep track of positions in the array.

W 3. Simplify Programming Project 1(b) by taking advantage of the fact that an array name can be used as a pointer.

4. Simplify Programming Project 2(b) by taking advantage of the fact that an array name can be used as a pointer.

5. Modify Programming Project 14 from Chapter 8 so that it uses a pointer instead of an integer to keep track of the current position in the array that contains the sentence.

6. Modify the `qsort.c` program of Section 9.6 so that `low`, `high`, and `middle` are pointers to array elements rather than integers. The `split` function will need to return a pointer, not an integer.

7. Modify the `maxmin.c` program of Section 11.4 so that the `max_min` function uses a pointer instead of an integer to keep track of the current position in the array.

13 Strings

It's difficult to extract sense from strings, but they're the only communication coin we can count on.

Although we've used `char` variables and arrays of `char` values in previous chapters, we still lack any convenient way to process a series of characters (a *string*, in C terminology). We'll remedy that defect in this chapter, which covers both string *constants* (or *literals*, as they're called in the C standard) and string *variables*, which can change during the execution of a program.

Section 13.1 explains the rules that govern string literals, including the rules for embedding escape sequences in string literals and for breaking long string literals. Section 13.2 then shows how to declare string variables, which are simply arrays of characters in which a special character—the null character—marks the end of a string. Section 13.3 describes ways to read and write strings. Section 13.4 shows how to write functions that process strings, and Section 13.5 covers some of the string-handling functions in the C library. Section 13.6 presents idioms that are often used when working with strings. Finally, Section 13.7 describes how to set up arrays whose elements are pointers to strings of different lengths. This section also explains how C uses such an array to supply command-line information to programs.

13.1 String Literals

A *string literal* is a sequence of characters enclosed within double quotes:

```
"When you come to a fork in the road, take it."
```

We first encountered string literals in Chapter 2; they often appear as format strings in calls of `printf` and `scanf`.

Escape Sequences in String Literals

escape sequences ➤7.3

String literals may contain the same escape sequences as character constants. We've used character escapes in `printf` and `scanf` format strings for some time. For example, we've seen that each `\n` character in the string

```
"Candy\nIs dandy\nBut liquor\nIs quicker.\n  --Ogden Nash\n"
```

causes the cursor to advance to the next line:

```
Candy
Is dandy
But liquor
Is quicker.
  --Ogden Nash
```

Although octal and hexadecimal escapes are also legal in string literals, they're not as common as character escapes.

Be careful when using octal and hexadecimal escape sequences in string literals. An octal escape ends after three digits or with the first non-octal character. For example, the string `"\1234"` contains two characters (`\123` and `4`), and the string `"\189"` contains three characters (`\1`, `8`, and `9`). A hexadecimal escape, on the other hand, isn't limited to three digits; it doesn't end until the first non-hex character. Consider what happens if a string contains the escape `\xfc`, which represents the character *ü* in the Latin1 character set, a common extension of ASCII. The string `"Z\xfcrich"` ("Zürich") has six characters (`Z`, `\xfc`, `r`, `i`, `c`, and `h`), but the string `"\xfcber"` (a failed attempt at "über") has only two (`\xfcbe` and `r`). Most compilers will object to the latter string, since hex escapes are usually limited to the range `\x0`–`\xff`.

Q&A

Continuing a String Literal

If we find that a string literal is too long to fit conveniently on a single line, C allows us to continue it on the next line, provided that we end the first line with a backslash character (\). No other characters may follow \ on the same line, other than the (invisible) new-line character at the end:

```
printf("When you come to a fork in the road, take it.  \
--Yogi Berra");
```

In general, the \ character can be used to join two or more lines of a program into a single line (a process that the C standard refers to as "splicing"). We'll see more examples of splicing in Section 14.3.

The \ technique has one drawback: the string must continue at the beginning of the next line, thereby wrecking the program's indented structure. There's a better way to deal with long string literals, thanks to the following rule: when two or more string literals are adjacent (separated only by white space), the compiler will

join them into a single string. This rule allows us to split a string literal over two or more lines:

```
printf("When you come to a fork in the road, take it.  "
       "--Yogi Berra");
```

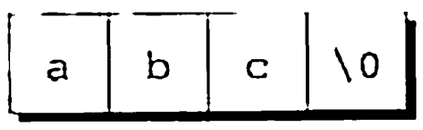
How String Literals Are Stored

We've used string literals often in calls of `printf` and `scanf`. But when we call `printf` and supply a string literal as an argument, what are we actually passing? To answer this question, we need to know how string literals are stored.

In essence, C treats string literals as character arrays. When a C compiler encounters a string literal of length *n* in a program, it sets aside *n* + 1 bytes of memory for the string. This area of memory will contain the characters in the string, plus one extra character—the ***null character***—to mark the end of the string. The null character is a byte whose bits are all zero, so it's represented by the `\0` escape sequence.

Don't confuse the null character (`'\0'`) with the zero character (`'0'`). The null character has the code 0; the zero character has a different code (48 in ASCII).

For example, the string literal `"abc"` is stored as an array of four characters (`a`, `b`, `c`, and `\0`):



String literals may be empty: the string `""` is stored as a single null character:

Since a string literal is stored as an array, the compiler treats it as a pointer of type `char *`. Both `printf` and `scanf`, for example, expect a value of type `char *` as their first argument. Consider the following example:

```
printf("abc");
```

When `printf` is called, it's passed the address of `"abc"` (a pointer to where the letter `a` is stored in memory).

Operations on String Literals

In general, we can use a string literal wherever C allows a `char *` pointer. For example, a string literal can appear on the right side of an assignment:

```
char *p;

p = "abc";
```

This assignment doesn't copy the characters in `"abc"`; it merely makes `p` point to the first character of the string.

C allows pointers to be subscripted, so we can subscript string literals:

```
char ch;

ch = "abc"[1];
```

The new value of `ch` will be the letter `b`. The other possible subscripts are 0 (which would select the letter `a`), 2 (the letter `c`), and 3 (the null character). This property of string literals isn't used that much, but occasionally it's handy. Consider the following function, which converts a number between 0 and 15 into a character that represents the equivalent hex digit:

```
char digit_to_hex_char(int digit)
{
  return "0123456789ABCDEF"[digit];
}
```

Attempting to modify a string literal causes undefined behavior:

```
char *p = "abc";

*p = 'd';    /*** WRONG ***/
```

Q&A A program that tries to change a string literal may crash or behave erratically.

String Literals versus Character Constants

A string literal containing a single character isn't the same as a character constant. The string literal `"a"` is represented by a *pointer* to a memory location that contains the character `a` (followed by a null character). The character constant `'a'` is represented by an *integer* (the numerical code for the character).

Don't ever use a character when a string is required (or vice versa). The call

```
printf("\n");
```

is legal, because `printf` expects a pointer as its first argument. The following call isn't legal, however:

```
printf('\n');    /*** WRONG ***/
```

13.2 String Variables

Some programming languages provide a special `string` type for declaring string variables. C takes a different tack: any one-dimensional array of characters can be used to store a string, with the understanding that the string is terminated by a null character. This approach is simple, but has significant difficulties. It's sometimes hard to tell whether an array of characters is being used as a string. If we write our own string-handling functions, we've got to be careful that they deal properly with the null character. Also, there's no faster way to determine the length of a string than a character-by-character search for the null character.

Let's say that we need a variable capable of storing a string of up to 80 characters. Since the string will need a null character at the end, we'll declare the variable to be an array of 81 characters:

idiom

```
#define STR_LEN 80
...
char str[STR_LEN+1];
```

We defined `STR_LEN` to be `80` rather than `81`, thus emphasizing the fact that `str` can store strings of no more than 80 characters, and then added 1 to `STR_LEN` in the declaration of `str`. This a common practice among C programmers.

When declaring an array of characters that will be used to hold a string, always make the array one character longer than the string, because of the C convention that every string is terminated by a null character. Failing to leave room for the null character may cause unpredictable results when the program is executed, since functions in the C library assume that strings are null-terminated.

Declaring a character array to have length `STR_LEN + 1` doesn't mean that it will always contain a string of `STR_LEN` characters. The length of a string depends on the position of the terminating null character, not on the length of the array in which the string is stored. An array of `STR_LEN + 1` characters can hold strings of various lengths, ranging from the empty string to strings of length `STR_LEN`.

Initializing a String Variable

A string variable can be initialized at the same time it's declared:

```
char date1[8] = "June 14";
```

The compiler will put the characters from `"June 14"` in the `date1` array, then add a null character so that `date1` can be used as a string. Here's what `date1` will look like:

Although `"June 14"` appears to be a string literal, it's not. Instead, C views it as an abbreviation for an array initializer. In fact, we could have written

```
char date1[8] = {'J', 'u', 'n', 'e', ' ', '1', '4', '\0'};
```

I think you'll agree that the original version is easier to read.

What if the initializer is too short to fill the string variable? In that case, the compiler adds extra null characters. Thus, after the declaration

```
char date2[9] = "June 14";
```

`date2` will have the following appearance:

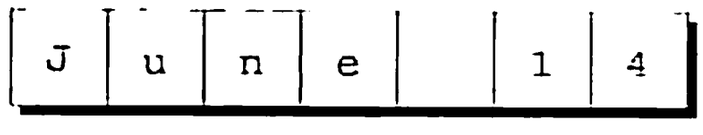
array initializers ➤8.1

This behavior is consistent with C's treatment of array initializers in general. When an array initializer is shorter than the array itself, the remaining elements are initialized to zero. By initializing the leftover elements of a character array to `\0`, the compiler is following the same rule.

What if the initializer is longer than the string variable? That's illegal for strings, just as it's illegal for other arrays. However, C does allow the initializer (not counting the null character) to have exactly the same length as the variable:

```
char date3[7] = "June 14";
```

There's no room for the null character, so the compiler makes no attempt to store one:



If you're planning to initialize a character array to contain a string, be sure that the length of the array is longer than the length of the initializer. Otherwise, the compiler will quietly omit the null character, making the array unusable as a string.

The declaration of a string variable may omit its length, in which case the compiler computes it:

```
char date4[] = "June 14";
```

The compiler sets aside eight characters for `date4`, enough to store the characters in `"June 14"` plus a null character. (The fact that the length of `date4` isn't specified doesn't mean that the array's length can be changed later. Once the program is compiled, the length of `date4` is fixed at eight.) Omitting the length of a string variable is especially useful if the initializer is long, since computing the length by hand is error-prone.

Character Arrays versus Character Pointers

Let's compare the declaration

```
char date[] = "June 14";
```

which declares `date` to be an *array*, with the similar-looking

```
char *date = "June 14";
```

which declares `date` to be a *pointer*. Thanks to the close relationship between arrays and pointers, we can use either version of `date` as a string. In particular, any function expecting to be passed a character array or character pointer will accept either version of `date` as an argument.

However, we must be careful not to make the mistake of thinking that the two versions of `date` are interchangeable. There are significant differences between the two:

- In the array version, the characters stored in `date` can be modified, like the elements of any array. In the pointer version, `date` points to a string literal, and we saw in Section 13.1 that string literals shouldn't be modified.
- In the array version, `date` is an array name. In the pointer version, `date` is a variable that can be made to point to other strings during program execution.

If we need a string that can be modified, it's our responsibility to set up an array of characters in which to store the string; declaring a pointer variable isn't enough. The declaration

```
char *p;
```

causes the compiler to set aside enough memory for a pointer variable; unfortunately, it doesn't allocate space for a string. (And how could it? We haven't indicated how long the string would be.) Before we can use `p` as a string, it must point to an array of characters. One possibility is to make `p` point to a string variable:

```
char str[STR_LEN+1], *p;

p = str;
```

`p` now points to the first character of `str`, so we can use `p` as a string. Another possibility is to make `p` point to a dynamically allocated string.

dynamically allocated strings ➤ 17.2

Using an uninitialized pointer variable as a string is a serious error. Consider the following example, which attempts to build the string "abc":

```
char *p;

p[0] = 'a';     /*** WRONG ***/
p[1] = 'b';     /*** WRONG ***/
p[2] = 'c';     /*** WRONG ***/
p[3] = '\0';    /*** WRONG ***/
```

Since p hasn't been initialized, we don't know where it's pointing. Using the pointer to write the characters a, b, c, and \0 into memory causes undefined behavior.

13.3 Reading and Writing Strings

Writing a string is easy using either the `printf` or `puts` functions. Reading a string is a bit harder, primarily because of the possibility that the input string may be longer than the string variable into which it's being stored. To read a string in a single step, we can use either `scanf` or `gets`. As an alternative, we can read strings one character at a time.

Writing Strings Using `printf` and `puts`

The `%s` conversion specification allows `printf` to write a string. Consider the following example:

```
char str[] = "Are we having fun yet?";

printf("%s\n", str);
```

The output will be

```
Are we having fun yet?
```

`printf` writes the characters in a string one by one until it encounters a null character. (If the null character is missing, `printf` continues past the end of the string until—eventually—it finds a null character somewhere in memory.)

To print just part of a string, we can use the conversion specification `%.`*p*`s`, where *p* is the number of characters to be displayed. The statement

```
printf("%.6s\n", str);
```

will print

```
Are we
```

A string, like a number, can be printed within a field. The %*m*s conversion will display a string in a field of size *m*. (A string with more than *m* characters will be printed in full, not truncated.) If the string has fewer than *m* characters, it will be right-justified within the field. To force left justification instead, we can put a minus sign in front of *m*. The *m* and *p* values can be used in combination: a conversion specification of the form %*m*.*p*s causes the first *p* characters of a string to be displayed in a field of size *m*.

`printf` isn't the only function that can write strings. The C library also provides `puts`, which is used in the following way:

```
puts(str);
```

`puts` has only one argument (the string to be printed). After writing the string, `puts` always writes an additional new-line character, thus advancing to the beginning of the next output line.

Reading Strings Using `scanf` and `gets`

The `%s` conversion specification allows `scanf` to read a string into a character array:

```
scanf("%s", str);
```

There's no need to put the & operator in front of `str` in the call of `scanf`; like any array name, `str` is treated as a pointer when passed to a function.

white-space characters ➤ 3.2

When `scanf` is called, it skips white space, then reads characters and stores them in `str` until it encounters a white-space character. `scanf` always stores a null character at the end of the string.

A string read using `scanf` will never contain white space. Consequently, `scanf` won't usually read a full line of input; a new-line character will cause `scanf` to stop reading, but so will a space or tab character. To read an entire line of input at a time, we can use `gets`. Like `scanf`, the `gets` function reads input characters into an array, then stores a null character. In other respects, however, `gets` is somewhat different from `scanf`:

- `gets` doesn't skip white space before starting to read the string (`scanf` does).
- `gets` reads until it finds a new-line character (`scanf` stops at any white-space character). Incidentally, `gets` discards the new-line character instead of storing it in the array; the null character takes its place.

To see the difference between `scanf` and `gets`, consider the following program fragment:

```
char sentence[SENT_LEN+1];

printf("Enter a sentence:\n");
scanf("%s", sentence);
```

Suppose that after the prompt

```
Enter a sentence:
```

the user enters the line

```
  To C, or not to C: that is the question.
```

`scanf` will store the string `"To"` in `sentence`. The next call of `scanf` will resume reading the line at the space after the word `To`.

Now suppose that we replace `scanf` by `gets`:

```
gets(sentence);
```

When the user enters the same input as before, `gets` will store the string

```
"  To C, or not to C: that is the question."
```

in `sentence`.

As they read characters into an array, `scanf` and `gets` have no way to detect when it's full. Consequently, they may store characters past the end of the array, causing undefined behavior. `scanf` can be made safer by using the conversion specification `%ns` instead of `%s`, where *n* is an integer indicating the maximum number of characters to be stored. `gets`, unfortunately, is inherently unsafe: `fgets` is a much better alternative.

fgets function ➤22.5

Reading Strings Character by Character

Since both `scanf` and `gets` are risky and insufficiently flexible for many applications, C programmers often write their own input functions. By reading strings one character at a time, these functions provide a greater degree of control than the standard input functions.

If we decide to design our own input function, we'll need to consider the following issues:

- Should the function skip white space before beginning to store the string?
- What character causes the function to stop reading: a new-line character, any white-space character, or some other character? Is this character stored in the string or discarded?
- What should the function do if the input string is too long to store: discard the extra characters or leave them for the next input operation?

Suppose we need a function that doesn't skip white-space characters, stops reading at the first new-line character (which isn't stored in the string), and discards extra characters. The function might have the following prototype:

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

`str` represents the array into which we'll store the input, and `n` is the maximum number of characters to be read. If the input line contains more than `n` characters, `read_line` will discard the additional characters. We'll have `read_line` return the number of characters it actually stores in `str` (a number anywhere from 0 to `n`). We may not always need `read_line`'s return value, but it doesn't hurt to have it available.

getchar function ➤7.3

`read_line` consists primarily of a loop that calls `getchar` to read a character and then stores the character in `str`, provided that there's room left. The loop terminates when the new-line character is read. (Strictly speaking, we should also have the loop terminate if `getchar` should fail to read a character, but we'll ignore that complication for now.) Here's the complete definition of `read_line`:

Q&A

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

  while ((ch = getchar()) != '\n')
    if (i < n)
      str[i++] = ch;
  str[i] = '\0';        /* terminates string */
  return i;             /* number of characters stored */
}
```

Note that `ch` has `int` type rather than `char` type, because `getchar` returns the character that it reads as an `int` value.

Before returning, `read_line` puts a null character at the end of the string. Standard functions such as `scanf` and `gets` automatically put a null character at the end of an input string; if we're writing our own input function, however, we must take on that responsibility.

13.4 Accessing the Characters in a String

Since strings are stored as arrays, we can use subscripting to access the characters in a string. To process every character in a string `s`, for example, we can set up a loop that increments a counter `i` and selects characters via the expression `s[i]`.

Suppose that we need a function that counts the number of spaces in a string. Using array subscripting, we might write the function in the following way:

```
int count_spaces(const char s[])
{
  int count = 0, i;

  for (i = 0; s[i] != '\0'; i++)
    if (s[i] == ' ')
      count++;
  return count;
}
```

I've included `const` in the declaration of `s` to indicate that `count_spaces` doesn't change the array that `s` represents. If `s` were not a string, the function would need a second argument specifying the length of the array. Since `s` is a string, however, `count_spaces` can determine where it ends by testing for the null character.

Many C programmers wouldn't write `count_spaces` as we have. Instead, they'd use a pointer to keep track of the current position within the string. As we saw in Section 12.2, this technique is always available for processing arrays, but it proves to be especially convenient for working with strings.

Let's rewrite the `count_spaces` function using pointer arithmetic instead of array subscripting. We'll eliminate the variable `i` and use `s` itself to keep track of our position in the string. By incrementing `s` repeatedly, `count_spaces` can step through each character in the string. Here's our new version of the function:

```
int count_spaces(const char *s)
{
  int count = 0;

  for (; *s != '\0'; s++)
    if (*s == ' ')
      count++;
  return count;
}
```

Note that `const` doesn't prevent `count_spaces` from modifying `s`; it's there to prevent the function from modifying what `s` points to. And since `s` is a copy of the pointer that's passed to `count_spaces`, incrementing `s` doesn't affect the original pointer.

The `count_spaces` example raises some questions about how to write string functions:

- ***Is it better to use array operations or pointer operations to access the characters in a string?*** We're free to use whichever is more convenient; we can even mix the two. In the second version of `count_spaces`, treating `s` as a pointer simplifies the function slightly by removing the need for the variable `i`. Traditionally, C programmers lean toward using pointer operations for processing strings.
- ***Should a string parameter be declared as an array or as a pointer?*** The two versions of `count_spaces` illustrate the options: the first version declares `s` to be an array; the second declares `s` to be a pointer. Actually, there's no difference between the two declarations—recall from Section 12.3 that the compiler treats an array parameter as though it had been declared as a pointer.
- ***Does the form of the parameter (`s[]` or `*s`) affect what can be supplied as an argument?*** No. When `count_spaces` is called, the argument could be an array name, a pointer variable, or a string literal—`count_spaces` can't tell the difference.

13.5 Using the C String Library

Some programming languages provide operators that can copy strings, compare strings, concatenate strings, select substrings, and the like. C's operators, in contrast, are essentially useless for working with strings. Strings are treated as arrays in C, so they're restricted in the same ways as arrays—in particular, they can't be copied or compared using operators.

Direct attempts to copy or compare strings will fail. For example, suppose that `str1` and `str2` have been declared as follows:

```
char str1[10], str2[10];
```

Copying a string into a character array using the = operator is not possible:

```
str1 = "abc";   /*** WRONG ***/
str2 = str1;    /*** WRONG ***/
```

We saw in Section 12.3 that using an array name as the left operand of = is illegal. *Initializing* a character array using = is legal, though:

```
char str1[10] = "abc";
```

In the context of a declaration, = is not the assignment operator.

Attempting to compare strings using a relational or equality operator is legal but won't produce the desired result:

```
if (str1 == str2) ...   /*** WRONG ***/
```

This statement compares `str1` and `str2` as *pointers*; it doesn't compare the contents of the two arrays. Since `str1` and `str2` have different addresses, the expression `str1 == str2` must have the value 0.

Fortunately, all is not lost: the C library provides a rich set of functions for performing operations on strings. Prototypes for these functions reside in the `<string.h>` header, so programs that need string operations should contain the following line:

`<string.h>` header ➤23.6

```
#include <string.h>
```

Most of the functions declared in `<string.h>` require at least one string as an argument. String parameters are declared to have type `char *`, allowing the argument to be a character array, a variable of type `char *`, or a string literal—all are suitable as strings. Watch out for string parameters that aren't declared `const`, however. Such a parameter may be modified when the function is called, so the corresponding argument shouldn't be a string literal.

There are many functions in <string.h>; I'll cover a few of the most basic. In subsequent examples, assume that str1 and str2 are character arrays used as strings.

The strcpy (String Copy) Function

The strcpy function has the following prototype in <string.h>:

```
char *strcpy(char *s1, const char *s2);
```

strcpy copies the string s2 into the string s1. (To be precise, we should say "strcpy copies the string pointed to by s2 into the array pointed to by s1.") That is, strcpy copies characters from s2 to s1 up to (and including) the first null character in s2. strcpy returns s1 (a pointer to the destination string). The string pointed to by s2 isn't modified, so it's declared const.

The existence of strcpy compensates for the fact that we can't use the assignment operator to copy strings. For example, suppose that we want to store the string "abcd" in str2. We can't use the assignment

```
str2 = "abcd";              /*** WRONG ***/
```

because str2 is an array name and can't appear on the left side of an assignment. Instead, we can call strcpy:

```
strcpy(str2, "abcd");   /* str2 now contains "abcd" */
```

Similarly, we can't assign str2 to str1 directly, but we can call strcpy:

```
strcpy(str1, str2);     /* str1 now contains "abcd" */
```

Most of the time, we'll discard the value that strcpy returns. On occasion, though, it can be useful to call strcpy as part of a larger expression in order to use its return value. For example, we could chain together a series of strcpy calls:

```
strcpy(str1, strcpy(str2, "abcd"));
  /* both str1 and str2 now contain "abcd" */
```

In the call strcpy(str1, str2), strcpy has no way to check that the string pointed to by str2 will actually fit in the array pointed to by str1. Suppose that str1 points to an array of length n. If the string that str2 points to has no more than $n - 1$ characters, then the copy will succeed. But if str2 points to a longer string, undefined behavior occurs. (Since strcpy always copies up to the first null character, it will continue copying past the end of the array that str1 points to.)

strncpy function ➤23.6

Calling the strncpy function is a safer, albeit slower, way to copy a string. strncpy is similar to strcpy but has a third argument that limits the number of characters that will be copied. To copy str2 into str1, we could use the following call of strncpy:

```
strncpy(str1, str2, sizeof(str1));
```

As long as `str1` is large enough to hold the string stored in `str2` (including the null character), the copy will be done correctly. `strncpy` itself isn't without danger, though. For one thing, it will leave the string in `str1` without a terminating null character if the length of the string stored in `str2` is greater than or equal to the size of the `str1` array. Here's a safer way to use `strncpy`:

```
strncpy(str1, str2, sizeof(str1) - 1);
str1[sizeof(str1)-1] = '\0';
```

The second statement guarantees that `str1` is always null-terminated, even if `strncpy` fails to copy a null character from `str2`.

The `strlen` (String Length) Function

The `strlen` function has the following prototype:

```
size_t strlen(const char *s);
```

size_t type ➤7.6

`size_t`, which is defined in the C library, is a `typedef` name that represents one of C's unsigned integer types. Unless we're dealing with extremely long strings, this technicality need not concern us—we can simply treat the return value of `strlen` as an integer.

`strlen` returns the length of a string `s`: the number of characters in `s` up to, but not including, the first null character. Here are a few examples:

```
int len;

len = strlen("abc");    /* len is now 3 */
len = strlen("");       /* len is now 0 */
strcpy(str1, "abc");
len = strlen(str1);     /* len is now 3 */
```

The last example illustrates an important point. When given an array as its argument, `strlen` doesn't measure the length of the array itself; instead, it returns the length of the string stored in the array.

The `strcat` (String Concatenation) Function

The `strcat` function has the following prototype:

```
char *strcat(char *s1, const char *s2);
```

`strcat` appends the contents of the string `s2` to the end of the string `s1`; it returns `s1` (a pointer to the resulting string).

Here are some examples of `strcat` in action:

```
strcpy(str1, "abc");
strcat(str1, "def");   /* str1 now contains "abcdef" */
```

```
strcpy(str1, "abc");
strcpy(str2, "def");
strcat(str1, str2);     /* str1 now contains "abcdef" */
```

As with `strcpy`, the value returned by `strcat` is normally discarded. The following example shows how the return value might be used:

```
strcpy(str1, "abc");
strcpy(str2, "def");
strcat(str1, strcat(str2, "ghi"));
  /* str1 now contains "abcdefghi"; str2 contains "defghi" */
```

The effect of the call `strcat(str1, str2)` is undefined if the array pointed to by `str1` isn't long enough to accommodate the additional characters from `str2`. Consider the following example:

```
char str1[6] = "abc";

strcat(str1, "def");    /*** WRONG ***/
```

`strcat` will attempt to add the characters `d`, `e`, `f`, and `\0` to the end of the string already stored in `str1`. Unfortunately, `str1` is limited to six characters, causing `strcat` to write past the end of the array.

strncat function ➤23.6

The `strncat` function is a safer but slower version of `strcat`. Like `strncpy`, it has a third argument that limits the number of characters it will copy. Here's what a call might look like:

```
strncat(str1, str2, sizeof(str1) - strlen(str1) - 1);
```

`strncat` will terminate `str1` with a null character, which isn't included in the third argument (the number of characters to be copied). In the example, 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.

The `strcmp` (String Comparison) Function

The `strcmp` function has the following prototype:

```
int strcmp(const char *s1, const char *s2);
```

`strcmp` compares the strings `s1` and `s2`, returning a value less than, equal to, or greater than 0, depending on whether `s1` is less than, equal to, or greater than `s2`. For example, to see if `str1` is less than `str2`, we'd write

```
if (strcmp(str1, str2) < 0)     /* is str1 < str2? */
  ...
```

To test whether `str1` is less than or equal to `str2`, we'd write

```
if (strcmp(str1, str2) <= 0)    /* is str1 <= str2? */
  ...
```

By choosing the proper relational operator (<, <=, >, >=) or equality operator (==, !=), we can test any possible relationship between `str1` and `str2`.

`strcmp` compares strings based on their lexicographic ordering, which resembles the way words are arranged in a dictionary. More precisely, `strcmp` considers `s1` to be less than `s2` if either one of the following conditions is satisfied:

- The first *i* characters of `s1` and `s2` match, but the (*i*+1)st character of `s1` is less than the (*i*+1)st character of `s2`. For example, `"abc"` is less than `"bcd"`, and `"abd"` is less than `"abe"`.
- All characters of `s1` match `s2`, but `s1` is shorter than `s2`. For example, `"abc"` is less than `"abcd"`.

As it compares characters from two strings, `strcmp` looks at the numerical codes that represent the characters. Some knowledge of the underlying character set is helpful in order to predict what `strcmp` will do. For example, here are a few important properties of the ASCII character set:

ASCII character set ➤*Appendix E*

- The characters in each of the sequences A–Z, a–z, and 0–9 have consecutive codes.
- All upper-case letters are less than all lower-case letters. (In ASCII, codes between 65 and 90 represent upper-case letters; codes between 97 and 122 represent lower-case letters.)
- Digits are less than letters. (Codes between 48 and 57 represent digits.)
- Spaces are less than all printing characters. (The space character has the value 32 in ASCII.)

PROGRAM Printing a One-Month Reminder List

To illustrate the use of the C string library, we'll now develop a program that prints a one-month list of daily reminders. The user will enter a series of reminders, with each prefixed by a day of the month. When the user enters 0 instead of a valid day, the program will print a list of all reminders entered, sorted by day. Here's what a session with the program will look like:

```
Enter day and reminder: 24 Susan's birthday
Enter day and reminder: 5 6:00 - Dinner with Marge and Russ
Enter day and reminder: 26 Movie - "Chinatown"
Enter day and reminder: 7 10:30 - Dental appointment
Enter day and reminder: 12 Movie - "Dazed and Confused"
Enter day and reminder: 5 Saturday class
Enter day and reminder: 12 Saturday class
Enter day and reminder: 0
```

```
Day Reminder
  5 Saturday class
  5 6:00 - Dinner with Marge and Russ
  7 10:30 - Dental appointment
 12 Saturday class
 12 Movie - "Dazed and Confused"
 24 Susan's birthday
 26 Movie - "Chinatown"
```

The overall strategy isn't very complicated: we'll have the program read a series of day-and-reminder combinations, storing them in order (sorted by day), and then display them. To read the days, we'll use `scanf`; to read the reminders, we'll use the `read_line` function of Section 13.3.

We'll store the 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 will search the array to determine where the day belongs, using `strcmp` to do comparisons. It will then use `strcpy` to move all strings *below* that point down one position. Finally, the program will copy the day into the array and call `strcat` to append the reminder to the day. (The day and the reminder have been kept separate up to this point.).

Of course, there are always a few minor complications. For example, we want the days to be right-justified in a two-character field, so that their ones digits will line up. There are many ways to handle the problem. I've chosen to have the program use `scanf` to read the day into an integer variable, then call `sprintf` to convert the day back into string form. `sprintf` is a library function that's similar to `printf`, except that it writes output into a string. The call

`sprintf` function ➤22.8

```
sprintf(day_str, "%2d", day);
```

writes the value of `day` into `day_str`. Since `sprintf` automatically adds a null character when it's through writing, `day_str` will contain a properly null-terminated string.

Another complication is making sure that the user doesn't enter more than two digits. We'll use the following call of `scanf` for this purpose:

```
scanf("%2d", &day);
```

The number 2 between `%` and `d` tells `scanf` to stop reading after two digits, even if the input has more digits.

With those details out of the way, here's the program:

remind.c

```
/* Prints a one-month reminder list */

#include <stdio.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][MSG_LEN+3];
  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--)
      strcpy(reminders[j], reminders[j-1]);

    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;
}
```

Although `remind.c` is useful for demonstrating the `strcpy`, `strcat`, and `strcmp` functions, it lacks something as a practical reminder program. There are

obviously a number of improvements needed, ranging from minor tweaks to major enhancements (such as saving the reminders in a file when the program terminates). We'll discuss several improvements in the programming projects at the end of this chapter and in later chapters.

13.6 String Idioms

Functions that manipulate strings are a particularly rich source of idioms. In this section, we'll explore some of the most famous idioms by using them to write the `strlen` and `strcat` functions. You'll never have to write these functions, of course, since they're part of the standard library, but you may have to write functions that are similar.

The concise style I'll use in this section is popular with many C programmers. You should master this style even if you don't plan to use it in your own programs, since you're likely to encounter it in code written by others.

One last note before we get started. If you want to try out any of the versions of `strlen` and `strcat` in this section, be sure to alter the name of the function (changing `strlen` to `my_strlen`, for example). As Section 21.1 explains, we're not allowed to write a function that has the same name as a standard library function, even when we don't include the header to which the function belongs. In fact, all names that begin with `str` and a lower-case letter are reserved (to allow functions to be added to the `<string.h>` header in future versions of the C standard).

Searching for the End of a String

Many string operations require searching for the end of a string. The `strlen` function is a prime example. The following version of `strlen` searches its string argument to find the end, using a variable to keep track of the string's length:

```
size_t strlen(const char *s)
{
  size_t n;

  for (n = 0; *s != '\0'; s++)
    n++;
  return n;
}
```

As the pointer `s` moves across the string from left to right, the variable `n` keeps track of how many characters have been seen so far. When `s` finally points to a null character, `n` contains the length of the string.

Let's see if we can condense the function. First, we'll move the initialization of `n` to its declaration:

```
size_t strlen(const char *s)
{
  size_t n = 0;

  for (; *s != '\0'; s++)
    n++;
  return n;
}
```

Next, we notice that the condition `*s != '\0'` is the same as `*s != 0`, because the integer value of the null character is 0. But testing `*s != 0` is the same as testing `*s`; both are true if `*s` isn't equal to 0. These observations lead to our next version of `strlen`:

```
size_t strlen(const char *s)
{
  size_t n = 0;

  for (; *s; s++)
    n++;
  return n;
}
```

But, as we saw in Section 12.2, it's possible to increment `s` and test `*s` in the same expression:

```
size_t strlen(const char *s)
{
  size_t n = 0;

  for (; *s++;)
    n++;
  return n;
}
```

Replacing the `for` statement with a `while` statement, we arrive at the following version of `strlen`:

```
size_t strlen(const char *s)
{
  size_t n = 0;

  while (*s++)
    n++;
  return n;
}
```

Although we've condensed `strlen` quite a bit, it's likely that we haven't increased its speed. Here's a version that *does* run faster, at least with some compilers:

```
size_t strlen(const char *s)
{
  const char *p = s;
```

```
  while (*s)
    s++;
  return s - p;
}
```

This version of `strlen` computes the length of the string by locating the position of the null character, then subtracting from it the position of the first character in the string. The improvement in speed comes from not having to increment `n` inside the `while` loop. Note the appearance of the word `const` in the declaration of `p`, by the way; without it, the compiler would notice that assigning `s` to `p` places the string that `s` points to at risk.

The statement

idiom
```
while (*s)
  s++;
```

and the related

idiom
```
while (*s++)
  ;
```

are idioms meaning "search for the null character at the end of a string." The first version leaves `s` pointing to the null character. The second version is more concise, but leaves `s` pointing just past the null character.

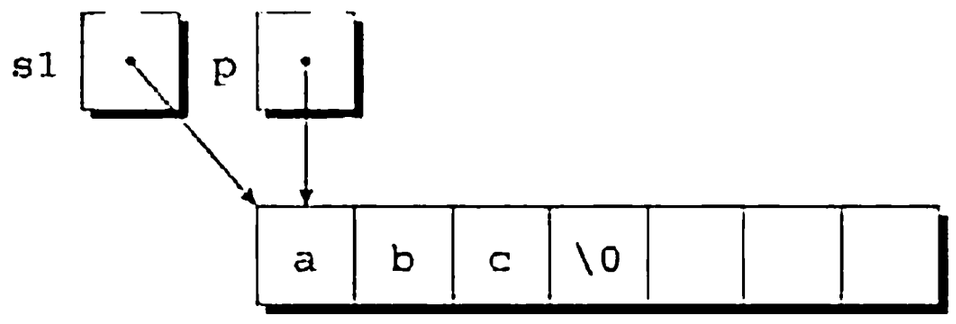
Copying a String

Copying a string is another common operation. To introduce C's "string copy" idiom, we'll develop two versions of the `strcat` function. Let's start with a straightforward but somewhat lengthy version:

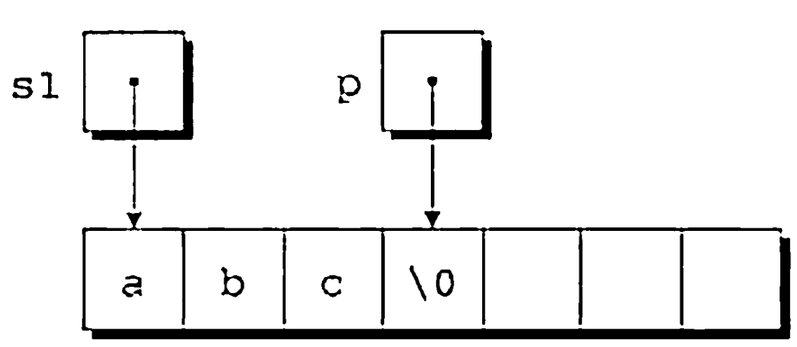
```
char *strcat(char *s1, const char *s2)
{
  char *p = s1;

  while (*p != '\0')
    p++;
  while (*s2 != '\0') {
    *p = *s2;
    p++;
    s2++;
  }
  *p = '\0';
  return s1;
}
```

This version of `strcat` uses a two-step algorithm: (1) Locate the null character at the end of the string `s1` and make `p` point to it. (2) Copy characters one by one from `s2` to where `p` is pointing.

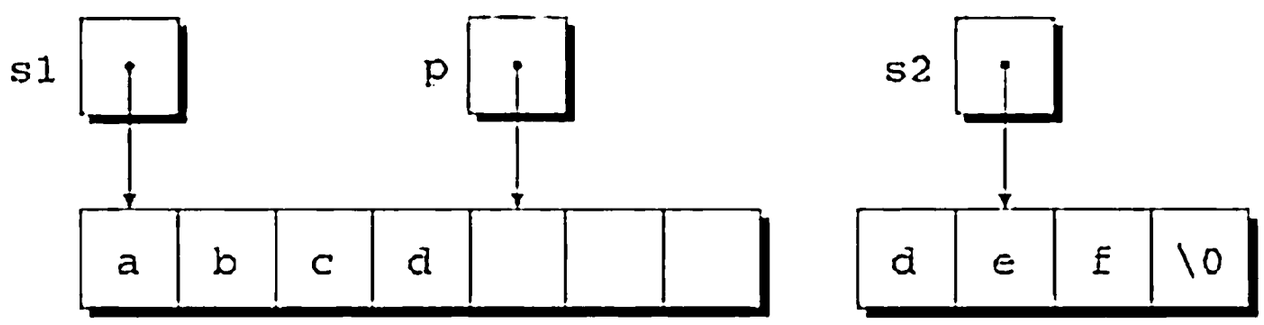
The first `while` statement in the function implements step (1). `p` is set to point to the first character in the `s1` string. Assuming that `s1` points to the string `"abc"`, we have the following picture:



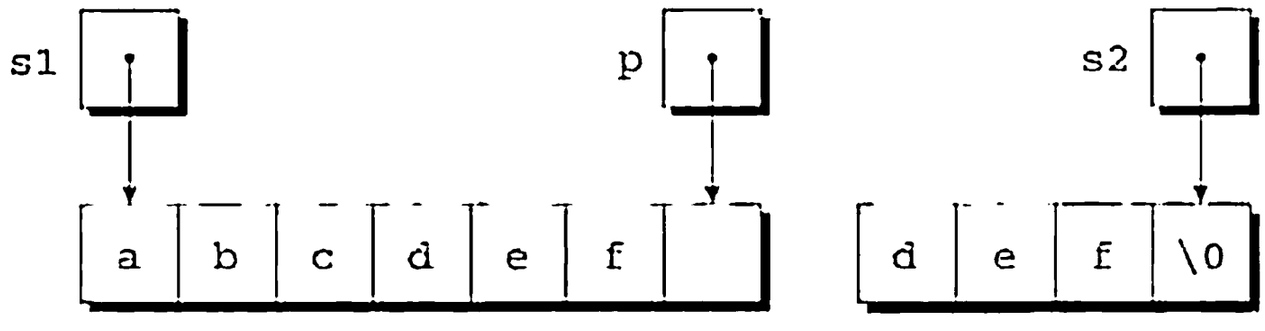
p is then incremented as long as it doesn't point to a null character. When the loop terminates, p must be pointing to the null character:



The second `while` statement implements step (2). The loop body copies one character from where `s2` points to where p points, then increments both p and `s2`. If `s2` originally points to the string `"def"`, here's what the strings will look like after the first loop iteration:



The loop terminates when `s2` points to the null character:



After putting a null character where p is pointing, `strcat` returns.

By a process similar to the one we used for `strlen`, we can condense the definition of `strcat`, arriving at the following version:

```
char *strcat(char *s1, const char *s2)
{
  char *p = s1;

  while (*p)
    p++;
  while (*p++ = *s2++)
    ;
  return s1;
}
```

The heart of our streamlined `strcat` function is the "string copy" idiom:

idiom

```
while (*p++ = *s2++)
  ;
```

If we ignore the two ++ operators, the expression inside the parentheses simplifies to an ordinary assignment:

```
*p = *s2
```

This expression copies a character from where `s2` points to where `p` points. After the assignment, both `p` and `s2` are incremented, thanks to the ++ operators. Repeatedly executing this expression has the effect of copying a series of characters from where `s2` points to where `p` points.

But what causes the loop to terminate? Since the primary operator inside the parentheses is assignment, the `while` statement tests the value of the assignment—the character that was copied. All characters except the null character test true, so the loop won't terminate until the null character has been copied. And since the loop terminates *after* the assignment, we don't need a separate statement to put a null character at the end of the new string.

13.7 Arrays of Strings

Let's now turn to a question that we'll often encounter: what's the best way to store an array of strings? The obvious solution is to create a two-dimensional array of characters, then store the strings in the array, one per row. Consider the following example:

```
char planets[][8] = {"Mercury", "Venus", "Earth",
                     "Mars", "Jupiter", "Saturn",
                     "Uranus", "Neptune", "Pluto"};
```

(In 2006, the International Astronomical Union demoted Pluto from "planet" to "dwarf planet," but I've left it in the `planets` array for old times' sake.) Note that we're allowed to omit the number of rows in the `planets` array—since that's obvious from the number of elements in the initializer—but C requires that we specify the number of columns.

The figure at the top of the next page shows what the `planets` array will look like. Not all our strings were long enough to fill an entire row of the array, so C padded them with null characters. There's a bit of wasted space in this array, since only three planets have names long enough to require eight characters (including the terminating null character). The `remind.c` program (Section 13.5) is a glaring example of this kind of waste. It stores reminders in rows of a two-dimensional character array, with 60 characters set aside for each reminder. In our example, the reminders ranged from 18 to 37 characters in length, so the amount of wasted space was considerable.

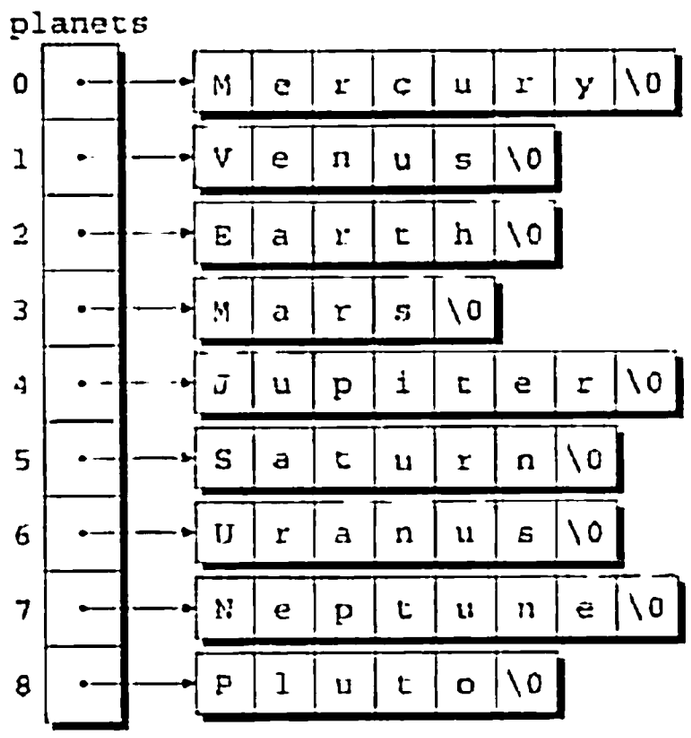
	0	1	2	3	4	5	6	7
0	M	e	r	c	u	r	y	\0
1	V	e	n	u	s	\0	\0	\0
2	E	a	r	t	h	\0	\0	\0
3	M	a	r	s	\0	\0	\0	\0
4	J	u	p	i	t	e	r	\0
5	S	a	t	u	r	n	\0	\0
6	U	r	a	n	u	s	\0	\0
7	N	e	p	t	u	n	e	\0
8	P	l	u	t	o	\0	\0	\0

The inefficiency that's apparent in these examples is common when working with strings, since most collections of strings will have a mixture of long strings and short strings. What we need is a ***ragged array:*** a two-dimensional array whose rows can have different lengths. C doesn't provide a "ragged array type," but it does give us the tools to simulate one. The secret is to create an array whose elements are *pointers* to strings.

Here's the `planets` array again. this time as an array of pointers to strings:

```
char *planets[] = {"Mercury", "Venus", "Earth",
                   "Mars", "Jupiter", "Saturn",
                   "Uranus", "Neptune", "Pluto"};
```

Not much of a change, eh? We simply removed one pair of brackets and put an asterisk in front of `planets`. The effect on how `planets` is stored is dramatic. though:



Each element of `planets` is a pointer to a null-terminated string. There are no longer any wasted characters in the strings, although we've had to allocate space for the pointers in the `planets` array.

To access one of the planet names, all we need do is subscript the `planets` array. Because of the relationship between pointers and arrays, accessing a character in a planet name is done in the same way as accessing an element of a two-

dimensional array. To search the `planets` array for strings beginning with the letter M, for example, we could use the following loop:

```
for (i = 0; i < 9; i++)
  if (planets[i][0] == 'M')
    printf("%s begins with M\n", planets[i]);
```

Command-Line Arguments

When we run a program, we'll often need to supply it with information—a file name, perhaps, or a switch that modifies the program's behavior. Consider the UNIX `ls` command. If we run `ls` by typing

```
ls
```

at the command line, it will display the names of the files in the current directory. But if we instead type

```
ls -l
```

then `ls` will display a "long" (detailed) listing of files, showing the size of each file, the file's owner, the date and time the file was last modified, and so forth. To modify the behavior of `ls` further, we can specify that it show details for just one file:

```
ls -l remind.c
```

`ls` will display detailed information about the file named `remind.c`.

Command-line information is available to all programs, not just operating system commands. To obtain access to these ***command-line arguments*** (called ***program parameters*** in the C standard), we must define `main` as a function with two parameters, which are customarily named `argc` and `argv`:

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

`argc` ("argument count") is the number of command-line arguments (including the name of the program itself). `argv` ("argument vector") is an array of pointers to the command-line arguments, which are stored in string form. `argv[0]` points to the name of the program, while `argv[1]` through `argv[argc-1]` point to the remaining command-line arguments.

`argv` has one additional element, `argv[argc]`, which is always a ***null pointer***—a special pointer that points to nothing. We'll discuss null pointers in a later chapter; for now, all we need to know is that the macro `NULL` represents a null pointer.

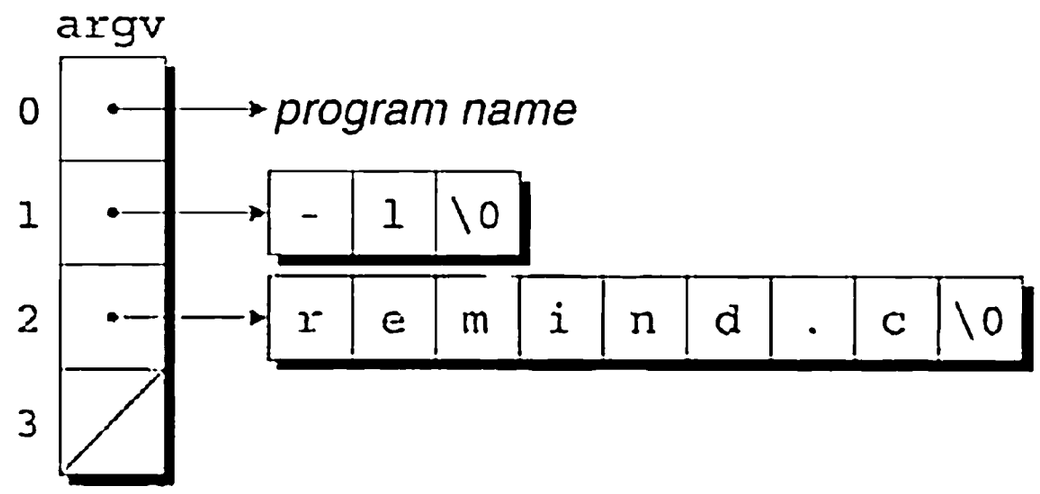
null pointers ➤ 17.1

If the user enters the command line

```
ls -l remind.c
```

then `argc` will be 3, `argv[0]` will point to a string containing the program

name, `argv[1]` will point to the string `"-l"`, `argv[2]` will point to the string `"remind.c"`, and `argv[3]` will be a null pointer:



This figure doesn't show the program name in detail, since it may include a path or other information that depends on the operating system. If the program name isn't available, `argv[0]` points to an empty string.

Since `argv` is an array of pointers, accessing command-line arguments is easy. Typically, a program that expects command-line arguments will set up a loop that examines each argument in turn. One way to write such a loop is to use an integer variable as an index into the `argv` array. For example, the following loop prints the command-line arguments, one per line:

```
int i;

for (i = 1; i < argc; i++)
  printf("%s\n", argv[i]);
```

Another technique is to set up a pointer to `argv[1]`, then increment the pointer repeatedly to step through the rest of the array. Since the last element of `argv` is always a null pointer, the loop can terminate when it finds a null pointer in the array:

```
char **p;

for (p = &argv[1]; *p != NULL; p++)
  printf("%s\n", *p);
```

Since `p` is a *pointer* to a *pointer* to a character, we've got to use it carefully. Setting `p` equal to `&argv[1]` makes sense; `argv[1]` is a pointer to a character, so `&argv[1]` will be a pointer to a pointer. The test `*p != NULL` is OK, since `*p` and `NULL` are both pointers. Incrementing `p` looks good; `p` points to an array element, so incrementing it will advance it to the next element. Printing `*p` is fine, since `*p` points to the first character in a string.

PROGRAM

Checking Planet Names

Our next program, `planet.c`, illustrates how to access command-line arguments. The program is designed to check a series of strings to see which ones are names of planets. When the program is run, the user will put the strings to be tested on the command line:

```
planet Jupiter venus Earth fred
```

The program will indicate whether or not each string is a planet name; if it is, the program will also display the planet's number (with planet 1 being the one closest to the Sun):

```
Jupiter is planet 5
venus is not a planet
Earth is planet 3
fred is not a planet
```

Notice that the program doesn't recognize a string as a planet name unless its first letter is upper-case and its remaining letters are lower-case.

planet.c

```
/* Checks planet names */

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

#define NUM_PLANETS 9

int main(int argc, char *argv[])
{
  char *planets[] = {"Mercury", "Venus", "Earth",
                     "Mars", "Jupiter", "Saturn",
                     "Uranus", "Neptune", "Pluto"};
  int i, j;

  for (i = 1; i < argc; i++) {
    for (j = 0; j < NUM_PLANETS; j++)
      if (strcmp(argv[i], planets[j]) == 0) {
        printf("%s is planet %d\n", argv[i], j + 1);
        break;
      }
    if (j == NUM_PLANETS)
      printf("%s is not a planet\n", argv[i]);
  }

  return 0;
}
```

The program visits each command-line argument in turn, comparing it with the strings in the `planets` array until it finds a match or reaches the end of the array. The most interesting part of the program is the call of `strcmp`, in which the arguments are `argv[i]` (a pointer to a command-line argument) and `planets[j]` (a pointer to a planet name).

Q & A

Q: **How long can a string literal be?**

A: According to the C89 standard, compilers must allow string literals to be at least

509 characters long. (Yes, you read that right—509. Don't ask.) C99 increases the
C99 minimum to 4095 characters.

Q: Why aren't string literals called "string constants"?

A: Because they're not necessarily constant. Since string literals are accessed through pointers, there's nothing to prevent a program from attempting to modify the characters in a string literal.

Q: How do we write a string literal that represents "über" if `"\xfcber"` doesn't work? [p. 278]

A: The secret is to write two adjacent string literals and let the compiler join them into one. In this example, writing `"\xfc" "ber"` will give us a string literal that represents the word "über."

Q: Modifying a string literal seems harmless enough. Why does it cause undefined behavior? [p. 280]

A: Some compilers try to reduce memory requirements by storing single copies of identical string literals. Consider the following example:

```
char *p = "abc", *q = "abc";
```

A compiler might choose to store `"abc"` just once, making both `p` and `q` point to it. If we were to change `"abc"` through the pointer `p`, the string that `q` points to would also be affected. Needless to say, this could lead to some annoying bugs. Another potential problem is that string literals might be stored in a "read-only" area of memory; a program that attempts to modify such a literal will simply crash.

Q: Should every array of characters include room for a null character?

A: Not necessarily, since not every array of characters is used as a string. Including room for the null character (and actually putting one into the array) is necessary only if you're planning to pass it to a function that requires a null-terminated string.

You do *not* need a null character if you'll only be performing operations on individual characters. For example, a program might have an array of characters that it will use to translate from one character set to another:

```
char translation_table[128];
```

The only operation that the program will perform on this array is subscripting. (The value of `translation_table[ch]` will be the translated version of the character `ch`.) We would not consider `translation_table` to be a string: it need not contain a null character, and no string operations will be performed on it.

Q: If `printf` and `scanf` expect their first argument to have type `char *`, does that mean that the argument can be a string *variable* instead of a string *literal*?

A: Yes, as the following example shows:

```
char fmt[] = "%d\n";
int i;
...
printf(fmt, i);
```

This ability opens the door to some intriguing possibilities—reading a format string as input, for example.

Q: If I want `printf` to write a string `str`, can't I just supply `str` as the format string, as in the following example?

```
printf(str);
```

A: Yes, but it's risky. If `str` contains the `%` character, you won't get the desired result, since `printf` will assume it's the beginning of a conversion specification.

***Q: How can `read_line` detect whether `getchar` has failed to read a character? [p. 287]**

A: If it can't read a character, either because of an error or because of end-of-file, `getchar` returns the value EOF, which has type `int`. Here's a revised version of `read_line` that tests whether the return value of `getchar` is EOF. Changes are marked in **bold:**

EOF macro ➤22.4

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

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

Q: Why does `strcmp` return a number that's less than, equal to, or greater than zero? Also, does the exact return value have any significance? [p. 292]

A: `strcmp`'s return value probably stems from the way the function is traditionally written. Consider the version in Kernighan and Ritchie's *The C Programming Language:*

```
int strcmp(char *s, char *t)
{
  int i;

  for (i = 0; s[i] == t[i]; i++)
    if (s[i] == '\0')
      return 0;
  return s[i] - t[i];
}
```

The return value is the difference between the first "mismatched" characters in the `s` and `t` strings, which will be negative if `s` points to a "smaller" string than `t` and positive if `s` points to a "larger" string. There's no guarantee that `strcmp` is actually written this way, though, so it's best not to assume that the magnitude of its return value has any particular meaning.

Q: My compiler issues a warning when I try to compile the `while` statement in the `strcat` function:

```
while (*p++ = *s2++)
  ;
```

What am I doing wrong?

A: Nothing. Many compilers—but not all, by any means—issue a warning if you use = where == is normally expected. This warning is valid at least 95% of the time, and it will save you a lot of debugging if you heed it. Unfortunately, the warning isn't relevant in this particular example; we actually *do* mean to use =, not ==. To get rid of the warning, rewrite the `while` loop as follows:

```
while ((*p++ = *s2++) != 0)
  ;
```

Since the `while` statement normally tests whether `*p++ = *s2++` is not 0, we haven't changed the meaning of the statement. The warning goes away, however, because the statement now tests a condition, not an assignment. With the GCC compiler, putting a pair of parentheses around the assignment is another way to avoid a warning:

```
while ((*p++ = *s2++))
  ;
```

Q: Are the `strlen` and `strcat` functions actually written as shown in Section 13.6?

A: Possibly, although it's common practice for compiler vendors to write these functions—and many other string functions—in assembly language instead of C. The string functions need to be as fast as possible, since they're used often and have to deal with strings of arbitrary length. Writing these functions in assembly language makes it possible to achieve great efficiency by taking advantage of any special string-handling instructions that the CPU may provide.

Q: Why does the C standard use the term "program parameters" instead of "command-line arguments"? [p. 302]

A: Programs aren't always run from a command line. In a typical graphical user interface, for example, programs are launched with a mouse click. In such an environment, there's no traditional command line, although there may be other ways of passing information to a program; the term "program parameters" leaves the door open for these alternatives.

Q: Do I have to use the names `argc` and `argv` for `main`'s parameters? [p. 302]

A: No. Using the names `argc` and `argv` is merely a convention, not a language requirement.

Q: I've seen `argv` declared as `**argv` instead of `*argv[]`. Is this legal?

A: Certainly. When declaring a parameter, writing `*a` is always the same as writing `a[]`, regardless of the type of `a`'s elements.

Q: We've seen how to set up an array whose elements are pointers to string literals. Are there any other applications for arrays of pointers?

A: Yes. Although we've focused on arrays of pointers to character strings, that's not the only application of arrays of pointers. We could just as easily have an array whose elements point to any type of data, whether in array form or not. Arrays of pointers are particularly useful in conjunction with dynamic storage allocation.

dynamic storage allocation ➤ 17.1

Exercises

Section 13.3

1. The following function calls supposedly write a single new-line character, but some are incorrect. Identify which calls don't work and explain why.

```
(a) printf("%c", '\n');      (g) putchar('\n');
(b) printf("%c", "\n");      (h) putchar("\n");
(c) printf("%s", '\n');      (i) puts('\n');
(d) printf("%s", "\n");      (j) puts("\n");
(e) printf('\n');            (k) puts("");
(f) printf("\n");
```

W 2. Suppose that `p` has been declared as follows:

```
char *p = "abc";
```

Which of the following function calls are legal? Show the output produced by each legal call, and explain why the others are illegal.

```
(a) putchar(p);
(b) putchar(*p);
(c) puts(p);
(d) puts(*p);
```

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

```
scanf("%d%s%d", &i, s, &j);
```

If the user enters `12abc34 56def78`, what will be the values of `i`, `s`, and `j` after the call? (Assume that `i` and `j` are `int` variables and `s` is an array of characters.)

W 4. Modify the `read_line` function in each of the following ways:

(a) Have it skip white space before beginning to store input characters.

(b) Have it stop reading at the first white-space character. *Hint:* To determine whether or not a character is white space, call the `isspace` function.

isspace function ➤ 23.5

(c) Have it stop reading at the first new-line character, then store the new-line character in the string.
(d) Have it leave behind characters that it doesn't have room to store.

Section 13.4

5. (a) Write a function named `capitalize` that capitalizes all letters in its argument. The argument will be a null-terminated string containing arbitrary characters, not just letters. Use array subscripting to access the characters in the string. *Hint:* Use the `toupper` function to convert each character to upper-case.

`toupper` function ➤*23.5*

(b) Rewrite the `capitalize` function, this time using pointer arithmetic to access the characters in the string.

W 6. Write a function named `censor` that modifies a string by replacing every occurrence of `foo` by `xxx`. For example, the string `"food fool"` would become `"xxxd xxxl"`. Make the function as short as possible without sacrificing clarity.

Section 13.5

7. Suppose that `str` is an array of characters. Which one of the following statements is not equivalent to the other three?

(a) `*str = 0;`
(b) `str[0] = '\0';`
(c) `strcpy(str, "");`
(d) `strcat(str, "");`

W *8. What will be the value of the string `str` after the following statements have been executed?

```
strcpy(str, "tire-bouchon");
strcpy(&str[4], "d-or-wi");
strcat(str, "red?");
```

9. What will be the value of the string `s1` after the following statements have been executed?

```
strcpy(s1, "computer");
strcpy(s2, "science");
if (strcmp(s1, s2) < 0)
  strcat(s1, s2);
else
  strcat(s2, s1);
s1[strlen(s1)-6] = '\0';
```

W 10. The following function supposedly creates an identical copy of a string. What's wrong with the function?

```
char *duplicate(const char *p)
{
  char *q;

  strcpy(q, p);
  return q;
}
```

11. The Q&A section at the end of this chapter shows how the `strcmp` function might be written using array subscripting. Modify the function to use pointer arithmetic instead.

12. Write the following function:

```
void get_extension(const char *file_name, char *extension);
```

`file_name` points to a string containing a file name. The function should store the extension on the file name in the string pointed to by `extension`. For example, if the file name is `"memo.txt"`, the function will store `"txt"` in the string pointed to by `extension`. If the file name doesn't have an extension, the function should store an empty string (a single null character) in the string pointed to by `extension`. Keep the function as simple as possible by having it use the `strlen` and `strcpy` functions.

13. Write the following function:

```
void build_index_url(const char *domain, char *index_url);
```

`domain` points to a string containing an Internet domain, such as `"knking.com"`. The function should add `"http://www."` to the beginning of this string and `"/index.html"` to the end of the string, storing the result in the string pointed to by `index_url`. (In this example, the result will be `"http://www.knking.com/index.html"`.) You may assume that `index_url` points to a variable that is long enought to hold the resulting string. Keep the function as simple as possible by having it use the `strcat` and `strcpy` functions.

Section 13.6

*14. What does the following program print?

```
#include <stdio.h>

int main(void)
{
  char s[] = "Hsjodi", *p;

  for (p = s; *p; p++)
    --*p;
  puts(s);
  return 0;
}
```

W *15. Let `f` be the following function:

```
int f(char *s, char *t)
{
  char *p1, *p2;

  for (p1 = s; *p1; p1++) {
    for (p2 = t; *p2; p2++)
      if (*p1 == *p2) break;
    if (*p2 == '\0') break;
  }
  return p1 - s;
}
```

(a) What is the value of `f("abcd", "babc")`?
(b) What is the value of `f("abcd", "bcd")`?
(c) In general, what value does `f` return when passed two strings `s` and `t`?

W 16. Use the techniques of Section 13.6 to condense the `count_spaces` function of Section 13.4. In particular, replace the `for` statement by a `while` loop.

17. Write the following function:

```
bool test_extension(const char *file_name,
                    const char *extension);
```

`file_name` points to a string containing a file name. The function should return `true` if the file's extension matches the string pointed to by `extension`, ignoring the case of letters. For example, the call `test_extension("memo.txt", "TXT")` would return `true`. Incorporate the "search for the end of a string" idiom into your function. *Hint:* Use the `toupper` function to convert characters to upper-case before comparing them.

`toupper` function ➤*23.5*

18. Write the following function:

```
void remove_filename(char *url);
```

`url` points to a string containing a URL (Uniform Resource Locator) that ends with a file name (such as `"http://www.knking.com/index.html"`). The function should modify the string by removing the file name and the preceding slash. (In this example, the result will be `"http://www.knking.com"`.) Incorporate the "search for the end of a string" idiom into your function. *Hint:* Have the function replace the last slash in the string by a null character.

Programming Projects

W 1. Write a program that finds the "smallest" and "largest" in a series of words. After the user enters the words, the program will determine which words would come first and last if the words were listed in dictionary order. The program must stop accepting input when the user enters a four-letter word. Assume that no word is more than 20 letters long. An interactive session with the program might look like this:

```
Enter word: dog
Enter word: zebra
Enter word: rabbit
Enter word: catfish
Enter word: walrus
Enter word: cat
Enter word: fish

Smallest word: cat
Largest word: zebra
```

Hint: Use two strings named `smallest_word` and `largest_word` to keep track of the "smallest" and "largest" words entered so far. Each time the user enters a new word, use `strcmp` to compare it with `smallest_word`; if the new word is "smaller," use `strcpy` to save it in `smallest_word`. Do a similar comparison with `largest_word`. Use `strlen` to determine when the user has entered a four-letter word.

2. Improve the `remind.c` program of Section 13.5 in the following ways:

   (a) Have the program print an error message and ignore a reminder if the corresponding day is negative or larger than 31. *Hint:* Use the `continue` statement.

   (b) Allow the user to enter a day, a 24-hour time, and a reminder. The printed reminder list should be sorted first by day, then by time. (The original program allows the user to enter a time, but it's treated as part of the reminder.)

   (c) Have the program print a one-*year* reminder list. Require the user to enter days in the form *month/day*.

3. Modify the `deal.c` program of Section 8.2 so that it prints the full names of the cards it deals:

```
Enter number of cards in hand: 5
Your hand:
Seven of clubs
Two of spades
Five of diamonds
Ace of spades
Two of hearts
```

Hint: Replace `rank_code` and `suit_code` by arrays containing pointers to strings.

W 4. Write a program named `reverse.c` that echoes its command-line arguments in reverse order. Running the program by typing

```
reverse void and null
```

should produce the following output:

```
null and void
```

5. Write a program named `sum.c` that adds up its command-line arguments, which are assumed to be integers. Running the program by typing

```
sum 8 24 62
```

should produce the following output:

```
Total: 94
```

`atoi` function ➤*26.2*

Hint: Use the `atoi` function to convert each command-line argument from string form to integer form.

W 6. Improve the `planet.c` program of Section 13.7 by having it ignore case when comparing command-line arguments with strings in the `planets` array.

7. Modify Programming Project 11 from Chapter 5 so that it uses arrays containing pointers to strings instead of `switch` statements. For example, instead of using a `switch` statement to print the word for the first digit, use the digit as an index into an array that contains the strings `"twenty"`, `"thirty"`, and so forth.

8. Modify Programming Project 5 from Chapter 7 so that it includes the following function:

```
int compute_scrabble_value(const char *word);
```

The function returns the SCRABBLE value of the string pointed to by `word`.

9. Modify Programming Project 10 from Chapter 7 so that it includes the following function:

```
int compute_vowel_count(const char *sentence);
```

The function returns the number of vowels in the string pointed to by the `sentence` parameter.

10. Modify Programming Project 11 from Chapter 7 so that it includes the following function:

```
void reverse_name(char *name);
```

The function expects `name` to point to a string containing a first name followed by a last name. It modifies the string so that the last name comes first, followed by a comma, a space, the first initial, and a period. The original string may contain extra spaces before the first name, between the first and last names, and after the last name.

11. Modify Programming Project 13 from Chapter 7 so that it includes the following function:

```
double compute_average_word_length(const char *sentence);
```

The function returns the average length of the words in the string pointed to by `sentence`.

12. Modify Programming Project 14 from Chapter 8 so that it stores the words in a two-dimensional `char` array as it reads the sentence, with each row of the array storing a single word. Assume that the sentence contains no more than 30 words and no word is more than 20 characters long. Be sure to store a null character at the end of each word so that it can be treated as a string.

13. Modify Programming Project 15 from Chapter 8 so that it includes the following function:

    ```
    void encrypt(char *message, int shift);
    ```

    The function expects `message` to point to a string containing the message to be encrypted; `shift` represents the amount by which each letter in the message is to be shifted.

14. Modify Programming Project 16 from Chapter 8 so that it includes the following function:

    ```
    bool are_anagrams(const char *word1, const char *word2);
    ```

    The function returns `true` if the strings pointed to by `word1` and `word2` are anagrams.

15. Modify Programming Project 6 from Chapter 10 so that it includes the following function:

    ```
    int evaluate_RPN_expression(const char *expression);
    ```

    The function returns the value of the RPN expression pointed to by `expression`.

16. Modify Programming Project 1 from Chapter 12 so that it includes the following function:

    ```
    void reverse(char *message);
    ```

    The function reverses the string pointed to by `message`. *Hint:* Use two pointers, one initially pointing to the first character of the string and the other initially pointing to the last character. Have the function reverse these characters and then move the pointers toward each other, repeating the process until the pointers meet.

17. Modify Programming Project 2 from Chapter 12 so that it includes the following function:

    ```
    bool is_palindrome(const char *message);
    ```

    The function returns `true` if the string pointed to by `message` is a palindrome.

18. Write a program that accepts a date from the user in the form *mm/dd/yyyy* and then displays it in the form *month dd, yyyy*, where *month* is the name of the month:

    ```
    Enter a date (mm/dd/yyyy): 2/17/2011
    You entered the date February 17, 2011
    ```

    Store the month names in an array that contains pointers to strings.

14 The Preprocessor

There will always be things we wish to say in our programs that in all known languages can only be said poorly.

In previous chapters, I've used the `#define` and `#include` directives without going into detail about what they do. These directives—and others that we haven't yet covered—are handled by the *preprocessor*, a piece of software that edits C programs just prior to compilation. Its reliance on a preprocessor makes C (along with C++) unique among major programming languages.

The preprocessor is a powerful tool, but it also can be a source of hard-to-find bugs. Moreover, the preprocessor can easily be misused to create programs that are almost impossible to understand. Although some C programmers depend heavily on the preprocessor, I recommend that it—like so many other things in life—be used in moderation.

This chapter begins by describing how the preprocessor works (Section 14.1) and giving some general rules that affect all preprocessing directives (Section 14.2). Sections 14.3 and 14.4 cover two of the preprocessor's major capabilities: macro definition and conditional compilation. (I'll defer detailed coverage of file inclusion, the other major capability, until Chapter 15.) Section 14.5 discusses the preprocessor's lesser-used directives: `#error`, `#line`, and `#pragma`.

14.1 How the Preprocessor Works

The behavior of the preprocessor is controlled by *preprocessing directives:* commands that begin with a # character. We've encountered two of these directives, `#define` and `#include`, in previous chapters.

The `#define` directive defines a *macro*—a name that represents something else, such as a constant or frequently used expression. The preprocessor responds to a `#define` directive by storing the name of the macro together with its definition.

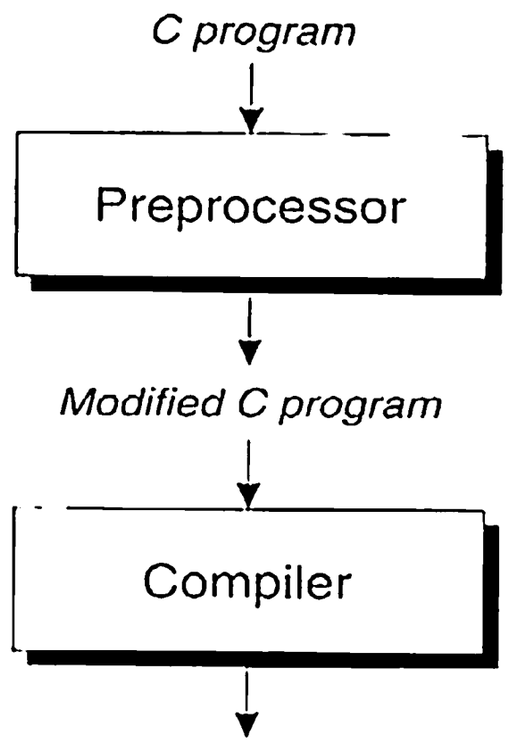
When the macro is used later in the program, the preprocessor "expands" the macro, replacing it by its defined value.

The `#include` directive tells the preprocessor to open a particular file and "include" its contents as part of the file being compiled. For example, the line

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

instructs the preprocessor to open the file named `stdio.h` and bring its contents into the program. (Among other things, `stdio.h` contains prototypes for C's standard input/output functions.)

The following diagram shows the preprocessor's role in the compilation process:



The input to the preprocessor is a C program, possibly containing directives. The preprocessor executes these directives, removing them in the process. The output of the preprocessor is another C program: an edited version of the original program, containing no directives. The preprocessor's output goes directly into the compiler, which checks the program for errors and translates it to object code (machine instructions).

To see what the preprocessor does, let's apply it to the `celsius.c` program of Section 2.6. Here's the original program:

```
/* Converts a Fahrenheit temperature to Celsius */

#include <stdio.h>

#define FREEZING_PT 32.0f
#define SCALE_FACTOR (5.0f / 9.0f)

int main(void)
{
  float fahrenheit, celsius;

  printf("Enter Fahrenheit temperature: ");
  scanf("%f", &fahrenheit);

  celsius = (fahrenheit - FREEZING_PT) * SCALE_FACTOR;
```

```
  printf("Celsius equivalent is: %.1f\n", celsius);

  return 0;
}
```

After preprocessing, the program will have the following appearance:

```
Blank line
Blank line
Lines brought in from stdio.h
Blank line
Blank line
Blank line
Blank line
Blank line
int main(void)
{
  float fahrenheit, celsius;

  printf("Enter Fahrenheit temperature: ");
  scanf("%f", &fahrenheit);

  celsius = (fahrenheit - 32.0f) * (5.0f / 9.0f);

  printf("Celsius equivalent is: %.1f\n", celsius);

  return 0;
}
```

The preprocessor responded to the `#include` directive by bringing in the contents of `stdio.h`. The preprocessor also removed the `#define` directives and replaced `FREEZING_PT` and `SCALE_FACTOR` wherever they appeared later in the file. Notice that the preprocessor doesn't remove lines containing directives; instead, it simply makes them empty.

As this example shows, the preprocessor does a bit more than just execute directives. In particular, it replaces each comment with a single space character. Some preprocessors go further and remove unnecessary white-space characters, including spaces and tabs at the beginning of indented lines.

In the early days of C, the preprocessor was a separate program that fed its output into the compiler. Nowadays, the preprocessor is often part of the compiler, and some of its output may not necessarily be C code. (For example, including a standard header such as `<stdio.h>` may have the effect of making its functions available to the program without necessarily copying the contents of the header into the program's source code.) Still, it's useful to think of the preprocessor as separate from the compiler. In fact, most C compilers provide a way to view the output of the preprocessor. Some compilers generate preprocessor output when a certain option is specified (GCC will do so when the `-E` option is used). Others come with a separate program that behaves like the integrated preprocessor. Check your compiler's documentation for more information.

A word of caution: The preprocessor has only a limited knowledge of C. As a result, it's quite capable of creating illegal programs as it executes directives. Often the original program looks fine, making errors harder to find. In complicated

programs, examining the output of the preprocessor may prove useful for locating this kind of error.

14.2 Preprocessing Directives

Most preprocessing directives fall into one of three categories:

- ***Macro definition.*** The `#define` directive defines a macro; the `#undef` directive removes a macro definition.
- ***File inclusion.*** The `#include` directive causes the contents of a specified file to be included in a program.
- ***Conditional compilation.*** The `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, and `#endif` directives allow blocks of text to be either included in or excluded from a program, depending on conditions that can be tested by the preprocessor.

The remaining directives—`#error`, `#line`, and `#pragma`—are more specialized and therefore used less often. We'll devote the rest of this chapter to an in-depth examination of preprocessing directives. The only directive we won't discuss in detail is `#include`, since it's covered in Section 15.2.

Before we go further, let's look at a few rules that apply to all directives:

- ***Directives always begin with the # symbol.*** The # symbol need not be at the beginning of a line, as long as only white space precedes it. After the # comes the name of the directive, followed by any other information the directive requires.
- ***Any number of spaces and horizontal tab characters may separate the tokens in a directive.*** For example, the following directive is legal:

```
#    define    N    100
```

- ***Directives always end at the first new-line character, unless explicitly continued.*** To continue a directive to the next line, we must end the current line with a \ character. For example, the following directive defines a macro that represents the capacity of a hard disk, measured in bytes:

```
#define DISK_CAPACITY (SIDES *                  \
                       TRACKS_PER_SIDE *        \
                       SECTORS_PER_TRACK *      \
                       BYTES_PER_SECTOR)
```

- ***Directives can appear anywhere in a program.*** Although we usually put `#define` and `#include` directives at the beginning of a file, other directives are more likely to show up later, even in the middle of function definitions.
- ***Comments may appear on the same line as a directive.*** In fact, it's good practice to put a comment at the end of a macro definition to explain the meaning of the macro:

```
#define FREEZING_PT 32.0f    /* freezing point of water */
```

14.3 Macro Definitions

The macros that we've been using since Chapter 2 are known as *simple* macros, because they have no parameters. The preprocessor also supports *parameterized* macros. We'll look first at simple macros, then at parameterized macros. After covering them separately, we'll examine properties shared by both.

Simple Macros

The definition of a ***simple macro*** (or ***object-like macro,*** as it's called in the C standard) has the form

#define directive (simple macro)

```
#define identifier replacement-list
```

replacement-list is any sequence of ***preprocessing tokens,*** which are similar to the tokens discussed in Section 2.8. Whenever we use the term "token" in this chapter, it means "preprocessing token."

A macro's replacement list may include identifiers, keywords, numeric constants, character constants, string literals, operators, and punctuation. When it encounters a macro definition, the preprocessor makes a note that *identifier* represents *replacement-list*; wherever *identifier* appears later in the file, the preprocessor substitutes *replacement-list*.

Don't put any extra symbols in a macro definition—they'll become part of the replacement list. Putting the = symbol in a macro definition is a common error:

```
#define N = 100    /*** WRONG ***/
...
int a[N];          /* becomes int a[= 100]; */
```

In this example, we've (incorrectly) defined `N` to be a pair of tokens (= and `100`). Ending a macro definition with a semicolon is another popular mistake:

```
#define N 100;     /*** WRONG ***/
...
int a[N];          /* becomes int a[100;]; */
```

Here `N` is defined to be the tokens `100` and `;`.

The compiler will detect most errors caused by extra symbols in a macro definition. Unfortunately, the compiler will flag each use of the macro as incorrect, rather than identifying the actual culprit—the macro's definition—which will have been removed by the preprocessor.

Simple macros are primarily used for defining what Kernighan and Ritchie call "manifest constants." Using macros, we can give names to numeric, character,

and string values:

```
#define STR_LEN 80
#define TRUE    1
#define FALSE   0
#define PI      3.14159
#define CR      '\r'
#define EOS     '\0'
#define MEM_ERR "Error: not enough memory"
```

Using `#define` to create names for constants has several significant advantages:

- ***It makes programs easier to read.*** The name of the macro—if well-chosen—helps the reader understand the meaning of the constant. The alternative is a program full of "magic numbers" that can easily mystify the reader.
- ***It makes programs easier to modify.*** We can change the value of a constant throughout a program by modifying a single macro definition. "Hard-coded" constants are more difficult to change, especially since they sometimes appear in a slightly altered form. (For example, a program with an array of length 100 may have a loop that goes from 0 to 99. If we merely try to locate occurrences of `100` in the program, we'll miss the `99`.)
- ***It helps avoid inconsistencies and typographical errors.*** If a numerical constant like `3.14159` appears many times in a program, chances are it will occasionally be written `3.1416` or `3.14195` by accident.

Although simple macros are most often used to define names for constants, they do have other applications:

- ***Making minor changes to the syntax of C.*** We can—in effect—alter the syntax of C by defining macros that serve as alternate names for C symbols. For example, programmers who prefer Pascal's `begin` and `end` to C's { and } can define the following macros:

  ```
  #define BEGIN {
  #define END   }
  ```

  We could go so far as to invent our own language. For example, we might create a LOOP "statement" that establishes an infinite loop:

  ```
  #define LOOP for (;;)
  ```

  Changing the syntax of C usually isn't a good idea, though, since it can make programs harder for others to understand.

- ***Renaming types.*** In Section 5.2, we created a Boolean type by renaming `int`:

  ```
  #define BOOL int
  ```

  type definitions ➤ 7.5

  Although some programmers use macros for this purpose, type definitions are a superior way to define type names.

- ***Controlling conditional compilation.*** Macros play an important role in controlling conditional compilation, as we'll see in Section 14.4. For example, the presence of the following line in a program might indicate that it's to be com-

piled in "debugging mode," with extra statements included to produce debugging output:

```
#define DEBUG
```

Incidentally, it's legal for a macro's replacement list to be empty, as this example shows.

When macros are used as constants, C programmers customarily capitalize all letters in their names. However, there's no consensus as to how to capitalize macros used for other purposes. Since macros (especially parameterized macros) can be a source of bugs, some programmers like to draw attention to them by using all upper-case letters in their names. Others prefer lower-case names, following the style of Kernighan and Ritchie's *The C Programming Language*.

Parameterized Macros

The definition of a ***parameterized macro*** (also known as a ***function-like macro***) has the form

#define directive (parameterized macro)

`#define` *identifier* (x_1 , x_2 , ... , x_n) *replacement-list*

where x_1, x_2, ..., x_n are identifiers (the macro's ***parameters***). The parameters may appear as many times as desired in the replacement list.

There must be *no space* between the macro name and the left parenthesis. If space is left, the preprocessor will assume that we're defining a simple macro; it will treat (x_1 , x_2 , ... , x_n) as part of the replacement list.

When the preprocessor encounters the definition of a parameterized macro, it stores the definition away for later use. Wherever a macro ***invocation*** of the form *identifier* (y_1 , y_2 , ... , y_n) appears later in the program (where y_1, y_2, ..., y_n are sequences of tokens), the preprocessor replaces it with *replacement-list*, substituting y_1 for x_1, y_2 for x_2, and so forth.

For example, suppose that we've defined the following macros:

```
#define MAX(x,y)    ((x)>(y)?(x):(y))
#define IS_EVEN(n)  ((n)%2==0)
```

(The number of parentheses in these macros may seem excessive, but there's a reason, as we'll see later in this section.) Now suppose that we invoke the two macros in the following way:

```
i = MAX(j+k, m-n);
if (IS_EVEN(i)) i++;
```

The preprocessor will replace these lines by

```
i = ((j+k)>(m-n)?(j+k):(m-n));
if (((i)%2==0)) i++;
```

As this example shows, parameterized macros often serve as simple functions. `MAX` behaves like a function that computes the larger of two values. `IS_EVEN` behaves like a function that returns 1 if its argument is an even number and 0 otherwise.

Here's a more complicated macro that behaves like a function:

```
#define TOUPPER(c) ('a'<=(c)&&(c)<='z'?(c)-'a'+'A':(c))
```

This macro tests whether the character `c` is between `'a'` and `'z'`. If so, it produces the upper-case version of `c` by subtracting `'a'` and adding `'A'`. If not, it leaves `c` unchanged. (The `<ctype.h>` header provides a similar function named `toupper` that's more portable.)

<ctype.h> header ➤ 23.5

A parameterized macro may have an empty parameter list. Here's an example:

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

The empty parameter list isn't really needed, but it makes `getchar` resemble a function. (Yes, this is the same `getchar` that belongs to `<stdio.h>`. We'll see in Section 22.4 that `getchar` is usually implemented as a macro as well as a function.)

Using a parameterized macro instead of a true function has a couple of advantages:

- ***The program may be slightly faster.*** A function call usually requires some overhead during program execution—context information must be saved, arguments copied, and so forth. A macro invocation, on the other hand, requires no run-time overhead. (Note, however, that C99's inline functions provide a way to avoid this overhead without the use of macros.)

C99

Inline functions ➤ 18.6

- ***Macros are "generic."*** Macro parameters, unlike function parameters, have no particular type. As a result, a macro can accept arguments of any type, provided that the resulting program—after preprocessing—is valid. For example, we could use the `MAX` macro to find the larger of two values of type `int`, `long`, `float`, `double`, and so forth.

But parameterized macros also have disadvantages:

- ***The compiled code will often be larger.*** Each macro invocation causes the insertion of the macro's replacement list, thereby increasing the size of the source program (and hence the compiled code). The more often the macro is used, the more pronounced this effect is. The problem is compounded when macro invocations are nested. Consider what happens when we use `MAX` to find the largest of three numbers:

```
n = MAX(i, MAX(j, k));
```

Here's the same statement after preprocessing:

```
n = ((i)>(((j)>(k)?(j):(k)))?(i):(((j)>(k)?(j):(k))));
```

- *Arguments aren't type-checked.* When a function is called, the compiler checks each argument to see if it has the appropriate type. If not, either the argument is converted to the proper type or the compiler produces an error message. Macro arguments aren't checked by the preprocessor, nor are they converted.
- *It's not possible to have a pointer to a macro.* As we'll see in Section 17.7, C allows pointers to functions, a concept that's quite useful in certain programming situations. Macros are removed during preprocessing, so there's no corresponding notion of "pointer to a macro"; as a result, macros can't be used in these situations.
- *A macro may evaluate its arguments more than once.* A function evaluates its arguments only once; a macro may evaluate its arguments two or more times. Evaluating an argument more than once can cause unexpected behavior if the argument has side effects. Consider what happens if one of MAX's arguments has a side effect:

```
n = MAX(i++, j);
```

Here's the same line after preprocessing:

```
n = ((i++)>(j)?(i++):(j));
```

If `i` is larger than `j`, then `i` will be (incorrectly) incremented twice and `n` will be assigned an unexpected value.

Errors caused by evaluating a macro argument more than once can be difficult to find, because a macro invocation looks the same as a function call. To make matters worse, a macro may work properly most of the time, failing only for certain arguments that have side effects. For self-protection, it's a good idea to avoid side effects in arguments.

Parameterized macros are good for more than just simulating functions. In particular, they're often used as patterns for segments of code that we find ourselves repeating. Suppose that we grow tired of writing

```
printf("%d\n", i);
```

every time we need to print an integer `i`. We might define the following macro, which makes it easier to display integers:

```
#define PRINT_INT(n) printf("%d\n", n)
```

Once `PRINT_INT` has been defined, the preprocessor will turn the line

```
PRINT_INT(i/j);
```

into

```
printf("%d\n", i/j);
```

The # Operator

Macro definitions may contain two special operators, # and ##. Neither operator is recognized by the compiler; instead, they're executed during preprocessing.

Q&A The # operator converts a macro argument into a string literal; it can appear only in the replacement list of a parameterized macro. (The operation performed by # is known as "stringization," a term that I'm sure you won't find in the dictionary.)

There are a number of uses for #; let's consider just one. Suppose that we decide to use the `PRINT_INT` macro during debugging as a convenient way to print the values of integer variables and expressions. The # operator makes it possible for `PRINT_INT` to label each value that it prints. Here's our new version of `PRINT_INT`:

```
#define PRINT_INT(n) printf(#n " = %d\n", n)
```

The # operator in front of `n` instructs the preprocessor to create a string literal from `PRINT_INT`'s argument. Thus, the invocation

```
PRINT_INT(i/j);
```

will become

```
printf("i/j" " = %d\n", i/j);
```

We saw in Section 13.1 that the compiler automatically joins adjacent string literals, so this statement is equivalent to

```
printf("i/j = %d\n", i/j);
```

When the program is executed, `printf` will display both the expression `i/j` and its value. If `i` is 11 and `j` is 2, for example, the output will be

```
i/j = 5
```

The ## Operator

The ## operator can "paste" two tokens (identifiers, for example) together to form a single token. (Not surprisingly, the ## operation is known as "token-pasting.") If one of the operands is a macro parameter, pasting occurs after the parameter has been replaced by the corresponding argument. Consider the following macro:

```
#define MK_ID(n) i##n
```

When `MK_ID` is invoked (as `MK_ID(1)`, say), the preprocessor first replaces the parameter `n` by the argument (`1` in this case). Next, the preprocessor joins `i` and `1` to make a single token (`i1`). The following declaration uses `MK_ID` to create three identifiers:

```
int MK_ID(1), MK_ID(2), MK_ID(3);
```

After preprocessing, this declaration becomes

```
int i1, i2, i3;
```

The ## operator isn't one of the most frequently used features of the preprocessor; in fact, it's hard to think of many situations that require it. To find a realistic application of ##, let's reconsider the MAX macro described earlier in this section. As we observed then, MAX doesn't behave properly if its arguments have side effects. The alternative to using the MAX macro is to write a max function. Unfortunately, one max function usually isn't enough; we may need a max function whose arguments are int values, one whose arguments are float values, and so on. All these versions of max would be identical except for the types of the arguments and the return type, so it seems a shame to define each one from scratch.

The solution is to write a macro that expands into the definition of a max function. The macro will have a single parameter, type, which represents the type of the arguments and the return value. There's just one snag: if we use the macro to create more than one max function, the program won't compile. (C doesn't allow two functions to have the same name if both are defined in the same file.) To solve this problem, we'll use the ## operator to create a different name for each version of max. Here's what the macro will look like:

```
#define GENERIC_MAX(type)          \
type type##_max(type x, type y)    \
{                                  \
  return x > y ? x : y;            \
}
```

Notice how type is joined with _max to form the name of the function.

Suppose that we happen to need a max function that works with float values. Here's how we'd use GENERIC_MAX to define the function:

```
GENERIC_MAX(float)
```

The preprocessor expands this line into the following code:

```
float float_max(float x, float y) { return x > y ? x : y; }
```

General Properties of Macros

Now that we've discussed both simple and parameterized macros, let's look at some rules that apply to both:

- *A macro's replacement list may contain invocations of other macros.* For example, we could define the macro TWO_PI in terms of the macro PI:

  ```
  #define PI      3.14159
  #define TWO_PI  (2*PI)
  ```

  When it encounters TWO_PI later in the program, the preprocessor replaces it by (2*PI). The preprocessor then *rescans* the replacement list to see if it

contains invocations of other macros (`PI`, in this case). The preprocessor will rescan the replacement list as many times as necessary to eliminate all macro names.

Q&A

- ***The preprocessor replaces only entire tokens, not portions of tokens.*** As a result, the preprocessor ignores macro names that are embedded in identifiers, character constants, and string literals. For example, suppose that a program contains the following lines:

```
#define SIZE 256

int BUFFER_SIZE;

if (BUFFER_SIZE > SIZE)
  puts("Error: SIZE exceeded");
```

After preprocessing, these lines will have the following appearance:

```
int BUFFER_SIZE;

if (BUFFER_SIZE > 256)
  puts("Error: SIZE exceeded");
```

The identifier `BUFFER_SIZE` and the string `"Error: SIZE exceeded"` weren't affected by preprocessing, even though both contain the word `SIZE`.

- ***A macro definition normally remains in effect until the end of the file in which it appears.*** Since macros are handled by the preprocessor, they don't obey normal scope rules. A macro defined inside the body of a function isn't local to that function; it remains defined until the end of the file.
- ***A macro may not be defined twice unless the new definition is identical to the old one.*** Differences in spacing are allowed, but the tokens in the macro's replacement list (and the parameters, if any) must be the same.
- ***Macros may be "undefined" by the `#undef` directive.*** The `#undef` directive has the form

`#undef` directive

```
#undef identifier
```

where *identifier* is a macro name. For example, the directive

```
#undef N
```

removes the current definition of the macro `N`. (If `N` hasn't been defined as a macro, the `#undef` directive has no effect.) One use of `#undef` is to remove the existing definition of a macro so that it can be given a new definition.

Parentheses in Macro Definitions

The replacement lists in our macro definitions have been full of parentheses. Is it really necessary to have so many? The answer is an emphatic yes; if we use fewer

parentheses, the macros will sometimes give unexpected—and undesirable—results.

There are two rules to follow when deciding where to put parentheses in a macro definition. First, if the macro's replacement list contains an operator, always enclose the replacement list in parentheses:

```
#define TWO_PI (2*3.14159)
```

Second, if the macro has parameters, put parentheses around each parameter every time it appears in the replacement list:

```
#define SCALE(x) ((x)*10)
```

Without the parentheses, we can't guarantee that the compiler will treat replacement lists and arguments as whole expressions. The compiler may apply the rules of operator precedence and associativity in ways that we didn't anticipate.

To illustrate the importance of putting parentheses around a macro's replacement list, consider the following macro definition, in which the parentheses are missing:

```
#define TWO_PI 2*3.14159
  /* needs parentheses around replacement list */
```

During preprocessing, the statement

```
conversion_factor = 360/TWO_PI;
```

becomes

```
conversion_factor = 360/2*3.14159;
```

The division will be performed before the multiplication, yielding a result different from the one intended.

Putting parentheses around the replacement list isn't enough if the macro has parameters—each occurrence of a parameter needs parentheses as well. For example, suppose that SCALE is defined as follows:

```
#define SCALE(x) (x*10)    /* needs parentheses around x */
```

During preprocessing, the statement

```
j = SCALE(i+1);
```

becomes

```
j = (i+1*10);
```

Since multiplication takes precedence over addition, this statement is equivalent to

```
j = i+10;
```

Of course, what we wanted was

```
j = (i+1)*10;
```

A shortage of parentheses in a macro definition can cause some of C's most frustrating errors. The program will usually compile and the macro will appear to work, failing only at the least convenient times.

Creating Longer Macros

The comma operator can be useful for creating more sophisticated macros by allowing us to make the replacement list a series of expressions. For example, the following macro will read a string and then print it:

```
#define ECHO(s) (gets(s), puts(s))
```

Calls of `gets` and `puts` are expressions, so it's perfectly legal to combine them using the comma operator. We can invoke ECHO as though it were a function:

```
ECHO(str);    /* becomes (gets(str), puts(str)); */
```

Instead of using the comma operator in the definition of ECHO, we could have enclosed the calls of `gets` and `puts` in braces to form a compound statement:

```
#define ECHO(s) { gets(s); puts(s); }
```

Unfortunately, this method doesn't work as well. Suppose that we use ECHO in an `if` statement:

```
if (echo_flag)
  ECHO(str);
else
  gets(str);
```

Replacing ECHO gives the following result:

```
if (echo_flag)
  { gets(str); puts(str); };
else
  gets(str);
```

The compiler treats the first two lines as a complete `if` statement:

```
if (echo_flag)
  { gets(str); puts(str); }
```

It treats the semicolon that follows as a null statement and produces an error message for the `else` clause, since it doesn't belong to any `if`. We could solve the problem by remembering not to put a semicolon after each invocation of ECHO, but then the program would look odd.

The comma operator solves this problem for ECHO, but not for all macros. Suppose that a macro needs to contain a series of *statements*, not just a series of *expressions*. The comma operator is of no help; it can glue together expressions.

but not statements. The solution is to wrap the statements in a do loop whose condition is false (and which therefore will be executed just once):

```
do { ... } while (0)
```

Notice that the do statement isn't complete—it needs a semicolon at the end. To see this trick (ahem, technique) in action, let's incorporate it into our ECHO macro:

```
#define ECHO(s)      \
        do {         \
          gets(s);   \
          puts(s);   \
        } while (0)
```

When ECHO is used, it must be followed by a semicolon, which completes the do statement:

```
ECHO(str);
  /* becomes do { gets(str); puts(str); } while (0); */
```

Predefined Macros

C has several predefined macros. Each macro represents an integer constant or string literal. As Table 14.1 shows, these macros provide information about the current compilation or about the compiler itself.

Table 14.1
Predefined Macros

Name	*Description*
`__LINE__`	Line number of file being compiled
`__FILE__`	Name of file being compiled
`__DATE__`	Date of compilation (in the form `"Mmm dd yyyy"`)
`__TIME__`	Time of compilation (in the form `"hh:mm:ss"`)
`__STDC__`	1 if the compiler conforms to the C standard (C89 or C99)

The `__DATE__` and `__TIME__` macros identify when a program was compiled. For example, suppose that a program begins with the following statements:

```
printf("Wacky Windows (c) 2010 Wacky Software, Inc.\n");
printf("Compiled on %s at %s\n", __DATE__, __TIME__);
```

Each time it begins to execute, the program will print two lines of the form

```
Wacky Windows (c) 2010 Wacky Software, Inc.
Compiled on Dec 23 2010 at 22:18:48
```

This information can be helpful for distinguishing among different versions of the same program.

We can use the `__LINE__` and `__FILE__` macros to help locate errors. Consider the problem of detecting the location of a division by zero. When a C program terminates prematurely because it divided by zero, there's usually no indication of which division caused the problem. The following macro can help us pinpoint the source of the error:

```
#define CHECK_ZERO(divisor) \
  if (divisor == 0) \
    printf("*** Attempt to divide by zero on line %d " \
           "of file %s ***\n", __LINE__, __FILE__)
```

The CHECK_ZERO macro would be invoked prior to a division:

```
CHECK_ZERO(j);
k = i / j;
```

If `j` happens to be zero, a message of the following form will be printed:

```
*** Attempt to divide by zero on line 9 of file foo.c ***
```

Error-detecting macros like this one are quite useful. In fact, the C library has a general-purpose error-detecting macro named `assert`.

assert macro ➤24.1

The `__STDC__` macro exists and has the value 1 if the compiler conforms to the C standard (either C89 or C99). By having the preprocessor test this macro, a program can adapt to a compiler that predates the C89 standard (see Section 14.4 for an example).

C99 Additional Predefined Macros in C99

C99 provides a few additional predefined macros (Table 14.2).

Table 14.2 Additional Predefined Macros in C99

Name	*Description*
`__STDC__HOSTED__`	1 if this is a hosted implementation; 0 if it is freestanding
`__STDC__VERSION__`	Version of C standard supported
`__STDC_IEC_559__` †	1 if IEC 60559 floating-point arithmetic is supported
`__STDC_IEC_559_COMPLEX__` †	1 if IEC 60559 complex arithmetic is supported
`__STDC_ISO_10646__` †	*yyyymm*L if `wchar_t` values match the ISO 10646 standard of the specified year and month

†Conditionally defined

To understand the meaning of `__STDC__HOSTED__`, we need some new vocabulary. An ***implementation*** of C consists of the compiler plus other software necessary to execute C programs. C99 divides implementations into two categories: hosted and freestanding. 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 beyond a few of the most basic. (In particular, a freestanding implementation doesn't have to support the `<stdio.h>` header.) The `__STDC__HOSTED__` macro represents the constant 1 if the compiler is a hosted implementation; otherwise, the macro has the value 0.

complex types ➤27.3

Q&A

The `__STDC__VERSION__` macro provides a way to check which version of the C standard is recognized by the compiler. This macro first appeared in Amendment 1 to the C89 standard, where its value was specified to be the long

integer constant `199409L` (representing the year and month of the amendment). If a compiler conforms to the C99 standard, the value is `199901L`. For each subsequent version of the standard (and each amendment to the standard), this macro will have a different value.

A C99 compiler may (or may not) define three additional macros. Each macro is defined only if the compiler meets a certain requirement:

- `__STDC_IEC_559__` is defined (and has the value 1) if the compiler performs floating-point arithmetic according to the IEC 60559 standard (another name for the IEEE 754 standard).

IEEE floating-point standard ➤7.2

- `__STDC_IEC_559_COMPLEX__` is defined (and has the value 1) if the compiler performs complex arithmetic according to the IEC 60559 standard.
- `__STDC_ISO_10646__` is defined as an integer constant of the form *yyyymm*`L` (for example, `199712L`) if values of type `wchar_t` are represented by the codes in the ISO/IEC 10646 standard (with revisions as of the specified year and month).

wchar_t type ➤25.2

ISO/IEC 10646 standard ➤25.2

C99 Empty Macro Arguments

C99 allows any or all of the arguments in a macro call to be empty. Such a call will contain the same number of commas as a normal call, however. (That way, it's easy to see which arguments have been omitted.)

In most cases, the effect of an empty argument is clear. Wherever the corresponding parameter name appears in the replacement list, it's replaced by nothing—it simply disappears from the replacement list. Here's an example:

```
#define ADD(x,y)  (x+y)
```

After preprocessing, the statement

```
i = ADD(j,k);
```

becomes

```
i =  (j+k);
```

whereas the statement

```
i = ADD(,k);
```

becomes

```
i =  (+k);
```

When an empty argument is an operand of the # or ## operators, special rules apply. If an empty argument is "stringized" by the # operator, the result is `""` (the empty string):

```
#define MK_STR(x) #x
...
char empty_string[] = MK_STR();
```

After preprocessing, the declaration will have the following appearance:

```
char empty_string[] = "";
```

If one of the arguments of the ## operator is empty, it's replaced by an invisible "placemarker" token. Concatenating an ordinary token with a placemarker token yields the original token (the placemarker disappears). If two placemarker tokens are concatenated, the result is a single placemarker. Once macro expansion has been completed, placemarker tokens disappear from the program. Consider the following example:

```
#define JOIN(x,y,z) x##y##z
...
int JOIN(a,b,c), JOIN(a,b,), JOIN(a,,c), JOIN(,,c);
```

After preprocessing, the declaration will have the following appearance:

```
int abc, ab, ac, c;
```

The missing arguments were replaced by placemarker tokens, which then disappeared when concatenated with any nonempty arguments. All three arguments to the JOIN macro could even be missing, which would yield an empty result.

C99 Macros with a Variable Number of Arguments

variable-length argument lists
➤26.1

In C89, a macro must have a fixed number of arguments, if it has any at all. C99 loosens things up a bit, allowing macros that take an unlimited number of arguments. This feature has long been available for functions, so it's not surprising that macros were finally put on an equal footing.

The primary reason for having a macro with a variable number of arguments is that it can pass these arguments to a function that accepts a variable number of arguments, such as `printf` or `scanf`. Here's an example:

```
#define TEST(condition, ...) ((condition)? \
  printf("Passed test: %s\n", #condition): \
  printf(__VA_ARGS__))
```

The . . . token, known as *ellipsis*, goes at the end of a macro's parameter list, preceded by ordinary parameters, if there are any. `__VA_ARGS__` is a special identifier that can appear only in the replacement list of a macro with a variable number of arguments; it represents all the arguments that correspond to the ellipsis. (There must be at least one argument that corresponds to the ellipsis, although that argument may be empty.) The `TEST` macro requires at least two arguments. The first argument matches the `condition` parameter; the remaining arguments match the ellipsis.

Here's an example that shows how the `TEST` macro might be used:

```
TEST(voltage <= max_voltage,
     "Voltage %d exceeds %d\n", voltage, max_voltage);
```

The preprocessor will produce the following output (reformatted for readability):

```
((voltage <= max_voltage)?
  printf("Passed test: %s\n", "voltage <= max_voltage"):
  printf("Voltage %d exceeds %d\n", voltage, max_voltage));
```

When the program is executed, the program will display the message

```
Passed test: voltage <= max_voltage
```

if `voltage` is no more than `max_voltage`. Otherwise, it will display the values of `voltage` and `max_voltage`:

```
Voltage 125 exceeds 120
```

C99 The `__func__` Identifier

Another new feature of C99 is the `__func__` identifier. `__func__` has nothing to do with the preprocessor, so it actually doesn't belong in this chapter. However, like many preprocessor features, it's useful for debugging, so I've chosen to discuss it here.

Every function has access to the `__func__` identifier, which behaves like a string variable that stores the name of the currently executing function. The effect is the same as if each function contains the following declaration at the beginning of its body:

```
static const char __func__[] = "function-name";
```

where *function-name* is the name of the function. The existence of this identifier makes it possible to write debugging macros such as the following:

```
#define FUNCTION_CALLED() printf("%s called\n", __func__);
#define FUNCTION_RETURNS() printf("%s returns\n", __func__);
```

Calls of these macros can then be placed inside functions to trace their calls:

```
void f(void)
{
  FUNCTION_CALLED();   /* displays "f called" */
  ...
  FUNCTION_RETURNS();  /* displays "f returns" */
}
```

Another use of `__func__`: it can be passed to a function to let it know the name of the function that called it.

14.4 Conditional Compilation

The C preprocessor recognizes a number of directives that support ***conditional compilation***—the inclusion or exclusion of a section of program text depending on the outcome of a test performed by the preprocessor.

The #if and #endif Directives

Suppose we're in the process of debugging a program. We'd like the program to print the values of certain variables, so we put calls of `printf` in critical parts of the program. Once we've located the bugs, it's often a good idea to let the `printf` calls remain, just in case we need them later. Conditional compilation allows us to leave the calls in place, but have the compiler ignore them.

Here's how we'll proceed. We'll first define a macro and give it a nonzero value:

```
#define DEBUG 1
```

The name of the macro doesn't matter. Next, we'll surround each group of `printf` calls by an `#if`-`#endif` pair:

```
#if DEBUG
printf("Value of i: %d\n", i);
printf("Value of j: %d\n", j);
#endif
```

During preprocessing, the `#if` directive will test the value of DEBUG. Since its value isn't zero, the preprocessor will leave the two calls of `printf` in the program (the `#if` and `#endif` lines will disappear, though). If we change the value of DEBUG to zero and recompile the program, the preprocessor will remove all four lines from the program. The compiler won't see the calls of `printf`, so they won't occupy any space in the object code and won't cost any time when the program is run. We can leave the `#if`-`#endif` blocks in the final program, allowing diagnostic information to be produced later (by recompiling with DEBUG set to 1) if any problems turn up.

In general, the `#if` directive has the form

`#if` directive

`#if` constant-expression

The `#endif` directive is even simpler:

`#endif` directive

`#endif`

Q&A When the preprocessor encounters the `#if` directive, it evaluates the constant expression. If the value of the expression is zero, the lines between `#if` and `#endif` will be removed from the program during preprocessing. Otherwise, the lines between `#if` and `#endif` will remain in the program to be processed by the compiler—the `#if` and `#endif` will have had no effect on the program.

It's worth noting that the `#if` directive treats undefined identifiers as macros that have the value 0. Thus, if we neglect to define DEBUG, the test

```
#if DEBUG
```

will fail (but not generate an error message), while the test

```
#if !DEBUG
```

will succeed.

The `defined` Operator

We encountered the # and ## operators in Section 14.3. There's just one other operator, `defined`, that's specific to the preprocessor. When applied to an identifier, `defined` produces the value 1 if the identifier is a currently defined macro; it produces 0 otherwise. The `defined` operator is normally used in conjunction with the `#if` directive; it allows us to write

```
#if defined(DEBUG)
...
#endif
```

The lines between the `#if` and `#endif` directives will be included in the program only if DEBUG is defined as a macro. The parentheses around DEBUG aren't required; we could simply write

```
#if defined DEBUG
```

Since `defined` tests only whether DEBUG is defined or not, it's not necessary to give DEBUG a value:

```
#define DEBUG
```

The `#ifdef` and `#ifndef` Directives

The `#ifdef` directive tests whether an identifier is currently defined as a macro:

`#ifdef` directive

`#ifdef` *identifier*

Using `#ifdef` is similar to using `#if`:

```
#ifdef identifier
Lines to be included if identifier is defined as a macro
#endif
```

Q&A Strictly speaking, there's no need for `#ifdef`, since we can combine the `#if` directive with the `defined` operator to get the same effect. In other words, the directive

`#ifdef` *identifier*

is equivalent to

`#if defined(`*identifier*`)`

The `#ifndef` directive is similar to `#ifdef`, but tests whether an identifier is *not* defined as a macro:

`#ifndef` directive

`#ifndef` *identifier*

Writing

`#ifndef` *identifier*

is the same as writing

`#if !defined(`*identifier*`)`

The `#elif` and `#else` Directives

`#if`, `#ifdef`, and `#ifndef` blocks can be nested just like ordinary `if` statements. When nesting occurs, it's a good idea to use an increasing amount of indentation as the level of nesting grows. Some programmers put a comment on each closing `#endif` to indicate what condition the matching `#if` tests:

```
#if DEBUG
...
#endif /* DEBUG */
```

This technique makes it easier for the reader to find the beginning of the `#if` block.

For additional convenience, the preprocessor supports the `#elif` and `#else` directives:

`#elif` directive

`#elif` *constant-expression*

`#else` directive

`#else`

`#elif` and `#else` can be used in conjunction with `#if`, `#ifdef`, or `#ifndef` to test a series of conditions:

```
#if expr1
Lines to be included if expr1 is nonzero
#elif expr2
Lines to be included if expr1 is zero but expr2 is nonzero
#else
Lines to be included otherwise
#endif
```

Although the `#if` directive is shown above, an `#ifdef` or `#ifndef` directive can be used instead. Any number of `#elif` directives—but at most one `#else`—may appear between `#if` and `#endif`.

Uses of Conditional Compilation

Conditional compilation is certainly handy for debugging, but its uses don't stop there. Here are a few other common applications:

- ***Writing programs that are portable to several machines or operating systems.*** The following example includes one of three groups of lines depending on whether `WIN32`, `MAC_OS`, or `LINUX` is defined as a macro:

  ```
  #if defined(WIN32)
  ...
  #elif defined(MAC_OS)
  ...
  #elif defined(LINUX)
  ...
  #endif
  ```

  A program might contain many of these `#if` blocks. At the beginning of the program, one (and only one) of the macros will be defined, thereby selecting a particular operating system. For example, defining the `LINUX` macro might indicate that the program is to run under the Linux operating system.

- ***Writing programs that can be compiled with different compilers.*** Different compilers often recognize somewhat different versions of C. Some accept a standard version of C, some don't. Some provide machine-specific language extensions; some don't, or provide a different set of extensions. Conditional compilation can allow a program to adjust to different compilers. Consider the problem of writing a program that might have to be compiled using an older, nonstandard compiler. The `__STDC__` macro allows the preprocessor to detect whether a compiler conforms to the standard (either C89 or C99); if it doesn't, we may need to change certain aspects of the program. In particular, we may have to use old-style function declarations (discussed in the Q&A at the end of Chapter 9) instead of function prototypes. At each point where functions are declared, we can put the following lines:

  ```
  #if __STDC__
  Function prototypes
  #else
  Old-style function declarations
  #endif
  ```

- ***Providing a default definition for a macro.*** Conditional compilation allows us to check whether a macro is currently defined and, if not, give it a default definition. For example, the following lines will define the macro `BUFFER_SIZE` if it wasn't previously defined:

  ```
  #ifndef BUFFER_SIZE
  #define BUFFER_SIZE 256
  #endif
  ```

- *Temporarily disabling code that contains comments.* We can't use a /*...*/ comment to "comment out" code that already contains /*...*/ comments. Instead, we can use an `#if` directive:

```
#if 0
Lines containing comments
#endif
```

Q&A Disabling code in this way is often called "conditioning out."

Section 15.2 discusses another common use of conditional compilation: protecting header files against multiple inclusion.

14.5 Miscellaneous Directives

To end the chapter, we'll take a brief look at the `#error`, `#line`, and `#pragma` directives. These directives are more specialized than the ones we've already examined, and they're used much less frequently.

The `#error` Directive

The `#error` directive has the form

`#error` directive

`#error message`

where *message* is any sequence of tokens. If the preprocessor encounters an `#error` directive, it prints an error message which must include *message*. The exact form of the error message can vary from one compiler to another; it might be something like

`Error directive:` *message*

or perhaps just

`#error` *message*

Encountering an `#error` directive indicates a serious flaw in the program; some compilers immediately terminate compilation without attempting to find other errors.

`#error` directives are frequently used in conjunction with conditional compilation to check for situations that shouldn't arise during a normal compilation. For example, suppose that we want to ensure that a program can't be compiled on a machine whose `int` type isn't capable of storing numbers up to 100,000. The largest possible `int` value is represented by the `INT_MAX` macro, so all we need do is invoke an `#error` directive if `INT_MAX` isn't at least 100,000:

INT_MAX macro ➤23.2

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

Attempting to compile the program on a machine whose integers are stored in 16 bits will produce a message such as

```
Error directive: int type is too small
```

The `#error` directive is often found in the `#else` part of an `#if-#elif-#else` series:

```
#if defined(WIN32)
...
#elif defined(MAC_OS)
...
#elif defined(LINUX)
...
#else
#error No operating system specified
#endif
```

The `#line` Directive

The `#line` directive is used to alter the way program lines are numbered. (Lines are usually numbered 1, 2, 3, as you'd expect.) We can also use this directive to make the compiler think that it's reading the program from a file with a different name.

The `#line` directive has two forms. In one form, we specify a line number:

`#line` directive (form 1)

```
#line n
```

n must be a sequence of digits representing an integer between 1 and 32767 (2147483647 in C99). This directive causes subsequent lines in the program to be numbered *n*, *n* + 1, *n* + 2, and so forth.

C99

In the second form of the `#line` directive, both a line number and a file name are specified:

`#line` directive (form 2)

```
#line n "file"
```

The lines that follow this directive are assumed to come from *file*, with line numbers starting at *n*. The values of *n* and/or the *file* string can be specified using macros.

One effect of the `#line` directive is to change the value of the `__LINE__` macro (and possibly the `__FILE__` macro). More importantly, most compilers will use the information from the `#line` directive when generating error messages.

For example, suppose that the following directive appears at the beginning of the file `foo.c`:

```
#line 10 "bar.c"
```

Let's say that the compiler detects an error on line 5 of `foo.c`. The error message will refer to line 13 of file `bar.c`, not line 5 of file `foo.c`. (Why line 13? The directive occupies line 1 of `foo.c`, so the renumbering of `foo.c` begins at line 2, which is treated as line 10 of `bar.c`.)

At first glance, the `#line` directive is mystifying. Why would we want error messages to refer to a different line and possibly a different file? Wouldn't this make programs harder to debug?

In fact, the `#line` directive isn't used very often by programmers. Instead, it's used primarily by programs that generate C code as output. The most famous example of such a program is `yacc` (Yet Another Compiler-Compiler), a UNIX utility that automatically generates part of a compiler. (The GNU version of `yacc` is named `bison`.) Before using `yacc`, the programmer prepares a file that contains information for `yacc` as well as fragments of C code. From this file, `yacc` generates a C program, `y.tab.c`, that incorporates the code supplied by the programmer. The programmer then compiles `y.tab.c` in the usual way. By inserting `#line` directives in `y.tab.c`, `yacc` tricks the compiler into believing that the code comes from the original file—the one written by the programmer. As a result, any error messages produced during the compilation of `y.tab.c` will refer to lines in the original file, not lines in `y.tab.c`. This makes debugging easier, because error messages refer to the file written by the programmer, not the (more complicated) file generated by `yacc`.

The `#pragma` Directive

The `#pragma` directive provides a way to request special behavior from the compiler. This directive is most useful for programs that are unusually large or that need to take advantage of the capabilities of a particular compiler.

The `#pragma` directive has the form

`#pragma` directive

`#pragma` *tokens*

where *tokens* are arbitrary tokens. `#pragma` directives can be very simple (a single token) or they can be much more elaborate:

```
#pragma data(heap_size => 1000, stack_size => 2000)
```

Not surprisingly, the set of commands that can appear in `#pragma` directives is different for each compiler; you'll have to consult the documentation for your compiler to see which commands it allows and what those commands do. Incidentally, the preprocessor must ignore any `#pragma` directive that contains an unrecognized command; it's not permitted to give an error message.

In C89, there are no standard pragmas—they're all implementation-defined.

C99 C99 has three standard pragmas, all of which use STDC as the first token following #pragma. These pragmas are FP_CONTRACT (covered in Section 23.4), CX_LIMITED_RANGE (Section 27.4), and FENV_ACCESS (Section 27.6).

C99 The _Pragma Operator

C99 introduces the _Pragma operator, which is used in conjunction with the #pragma directive. A _Pragma expression has the form

_Pragma expression

_Pragma (*string-literal*)

When it encounters such an expression, the preprocessor "destringizes" the string literal (yes, that's the term used in the C99 standard!) by removing the double quotes around the string and replacing the escape sequences \" and \\ by the characters " and \, respectively. The result is a series of tokens, which are then treated as though they appear in a #pragma directive. For example, writing

```
_Pragma("data(heap_size => 1000, stack_size => 2000)")
```

is the same as writing

```
#pragma data(heap_size => 1000, stack_size => 2000)
```

The _Pragma operator lets us work around a limitation of the preprocessor: the fact that a preprocessing directive can't generate another directive. _Pragma, however, is an operator, not a directive, and can therefore appear in a macro definition. This makes it possible for a macro expansion to leave behind a #pragma directive.

Let's look at an example from the GCC manual. The following macro uses the _Pragma operator:

```
#define DO_PRAGMA(x) _Pragma(#x)
```

The macro would be invoked as follows:

```
DO_PRAGMA(GCC dependency "parse.y")
```

After expansion, the result will be

```
#pragma GCC dependency "parse.y"
```

which is one of the pragmas supported by GCC. (It issues a warning if the date of the specified file—`parse.y` in this example—is more recent than the date of the current file—the one being compiled.) Note that the argument to the call of DO_PRAGMA is a series of tokens. The # operator in the definition of DO_PRAGMA causes the tokens to be stringized into `"GCC dependency \"parse.y\""`: this string is then passed to the _Pragma operator, which destringizes it, producing a #pragma directive containing the original tokens.

Q & A

Q: I've seen programs that contain a # on a line by itself. Is this legal?

A: Yes. This is the *null directive;* it has no effect. Some programmers use null directives for spacing within conditional compilation blocks:

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

Blank lines would also work, of course, but the # helps the reader see the extent of the block.

Q: I'm not sure which constants in a program need to be defined as macros. Are there any guidelines to follow? [p. 319]

A: One rule of thumb says that every numeric constant, other than 0 or 1, should be a macro. Character and string constants are problematic, since replacing a character or string constant by a macro doesn't always improve readability. I recommend using a macro instead of a character constant or string literal provided that (1) the constant is used more than once and (2) the possibility exists that the constant might someday be modified. Because of rule (2), I don't use macros such as

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

although some programmers do.

Q: What does the # operator do if the argument that it's supposed to "stringize" contains a " or \ character? [p. 324]

A: It converts " to \" and \ to \\. Consider the following macro:

```
#define STRINGIZE(x) #x
```

The preprocessor will replace `STRINGIZE("foo")` by `"\"foo\""`.

***Q: I can't get the following macro to work properly:**

```
#define CONCAT(x,y) x##y
```

`CONCAT(a,b)` gives `ab`, as expected, but `CONCAT(a,CONCAT(b,c))` gives an odd result. What's going on?

A: Thanks to rules that Kernighan and Ritchie call "bizarre," macros whose replacement lists depend on ## usually can't be called in a nested fashion. The problem is that `CONCAT(a,CONCAT(b,c))` isn't expanded in a "normal" fashion, with `CONCAT(b,c)` yielding `bc`, then `CONCAT(a,bc)` giving `abc`. Macro parameters that are preceded or followed by ## in a replacement list aren't expanded at

the time of substitution. As a result, `CONCAT(a,CONCAT(b,c))` expands to `aCONCAT(b,c)`, which can't be expanded further, since there's no macro named `aCONCAT`.

There's a way to solve the problem, but it's not pretty. The trick is to define a second macro that simply calls the first one:

```
#define CONCAT2(x,y) CONCAT(x,y)
```

Writing `CONCAT2(a,CONCAT2(b,c))` now yields the desired result. As the preprocessor expands the outer call of `CONCAT2`, it will expand `CONCAT2(b,c)` as well; the difference is that `CONCAT2`'s replacement list doesn't contain ##. If none of this makes any sense, don't worry; it's not a problem that arises often.

The # operator has a similar difficulty, by the way. If `#x` appears in a replacement list, where `x` is a macro parameter, the corresponding argument is not expanded. Thus, if `N` is a macro representing `10`, and `STR(x)` has the replacement list `#x`, expanding `STR(N)` yields `"N"`, not `"10"`. The solution is similar to the one we used with `CONCAT`: defining a second macro whose job is to call `STR`.

*Q: **Suppose that the preprocessor encounters the original macro name during rescanning, as in the following example:**

```
#define N (2*M)
#define M (N+1)
i = N;   /* infinite loop? */
```

The preprocessor will replace `N` by `(2*M)`, then replace `M` by `(N+1)`. Will the preprocessor replace `N` again, thus going into an infinite loop? [p. 326]

A: Some old preprocessors will indeed go into an infinite loop, but newer ones shouldn't. According to the C standard, if the original macro name reappears during the expansion of a macro, the name is not replaced again. Here's how the assignment to `i` will look after preprocessing:

```
i = (2*(N+1));
```

Some enterprising programmers take advantage of this behavior by writing macros whose names match reserved words or functions in the standard library. Consider the `sqrt` library function. `sqrt` computes the square root of its argument, returning an implementation-defined value if the argument is negative. Perhaps we would prefer that `sqrt` return 0 if its argument is negative. Since `sqrt` is part of the standard library, we can't easily change it. We can, however, define a `sqrt` *macro* that evaluates to 0 when given a negative argument:

sqrt function ➤23.3

```
#undef sqrt
#define sqrt(x) ((x)>=0?sqrt(x):0)
```

A later call of `sqrt` will be intercepted by the preprocessor, which expands it into the conditional expression shown here. The call of `sqrt` inside the conditional expression won't be replaced during rescanning, so it will remain for the compiler

to handle. (Note the use of `#undef` to undefine `sqrt` before defining the `sqrt` macro. As we'll see in Section 21.1, the standard library is allowed to have both a macro and a function with the same name. Undefining `sqrt` before defining our own `sqrt` macro is a defensive measure, in case the library has already defined `sqrt` as a macro.)

Q: I get an error when I try to use predefined macros such as `__LINE__` and `__FILE__`. Is there a special header that I need to include?

A: No. These macros are recognized automatically by the preprocessor. Make sure that you have *two* underscores at the beginning and end of each macro name, not one.

Q: What's the purpose of distinguishing between a "hosted implementation" and a "freestanding implementation"? If a freestanding implementation doesn't even support the `<stdio.h>` header, what use is it? [p. 330]

A: A hosted implementation is needed for most programs (including the ones in this book), which rely on the underlying operating system for input/output and other essential services. A freestanding implementation of C would be used for programs that require no operating system (or only a minimal operating system). For example, a freestanding implementation would be needed for writing the kernel of an operating system (which requires no traditional input/output and therefore doesn't need `<stdio.h>` anyway). Freestanding implementations are also useful for writing software for embedded systems.

Q: I thought the preprocessor was just an editor. How can it evaluate constant expressions? [p. 334]

A: The preprocessor is more sophisticated than you might expect; it knows enough about C to be able to evaluate constant expressions, although it doesn't do so in quite the same way as the compiler. (For one thing, the preprocessor treats any undefined name as having the value 0. The other differences are too esoteric to go into here.) In practice, the operands in a preprocessor constant expression are usually constants, macros that represent constants, and applications of the `defined` operator.

Q: Why does C provide the `#ifdef` and `#ifndef` directives, since we can get the same effect using the `#if` directive and the `defined` operator? [p. 335]

A: The `#ifdef` and `#ifndef` directives have been a part of C since the 1970s. The `defined` operator, on the other hand, was added to C in the 1980s during standardization. So the real question is: Why was `defined` added to the language? The answer is that `defined` adds flexibility. Instead of just being able to test the existence of a single macro using `#ifdef` or `#ifndef`, we can now test any number of macros using `#if` together with `defined`. For example, the following directive checks whether `FOO` and `BAR` are defined but `BAZ` is not defined:

```
#if defined(FOO) && defined(BAR) && !defined(BAZ)
```

Q: I wanted to compile a program that I hadn't finished writing, so I "conditioned out" the unfinished part:

```
#if 0
...
#endif
```

When I compiled the program, I got an error message referring to one of the lines between `#if` and `#endif`. Doesn't the preprocessor just ignore these lines? [p. 338]

A: No, the lines aren't completely ignored. Comments are processed before preprocessing directives are executed, and the source code is divided into preprocessing tokens. Thus, an unterminated comment between `#if` and `#endif` may cause an error message. Also, an unpaired single quote or double quote character may cause undefined behavior.

Exercises

Section 14.3

1. Write parameterized macros that compute the following values.
   (a) The cube of `x`.
   (b) The remainder when `n` is divided by 4.
   (c) 1 if the product of `x` and `y` is less than 100, 0 otherwise.

   Do your macros always work? If not, describe what arguments would make them fail.

W 2. Write a macro `NELEMS(a)` that computes the number of elements in a one-dimensional array `a`. *Hint:* See the discussion of the `sizeof` operator in Section 8.1.

3. Let `DOUBLE` be the following macro:

   ```
   #define DOUBLE(x) 2*x
   ```

   (a) What is the value of `DOUBLE(1+2)`?
   (b) What is the value of `4/DOUBLE(2)`?
   (c) Fix the definition of `DOUBLE`.

W 4. For each of the following macros, give an example that illustrates a problem with the macro and show how to fix it.

   (a) `#define AVG(x,y)  (x+y)/2`

   (b) `#define AREA(x,y)  (x)*(y)`

W *5. Let `TOUPPER` be the following macro:

   ```
   #define TOUPPER(c) ('a'<=(c)&&(c)<='z'?(c)-'a'+'A':(c))
   ```

   Let `s` be a string and let `i` be an `int` variable. Show the output produced by each of the following program fragments.

   (a)
   ```
   strcpy(s, "abcd");
   i = 0;
   putchar(TOUPPER(s[++i]));
   ```

(b)
```
strcpy(s, "0123");
i = 0;
putchar(TOUPPER(s[++i]));
```

6. (a) Write a macro `DISP(f,x)` that expands into a call of `printf` that displays the value of the function `f` when called with argument `x`. For example.

```
DISP(sqrt, 3.0);
```

should expand into

```
printf("sqrt(%g) = %g\n", 3.0, sqrt(3.0));
```

(b) Write a macro `DISP2(f,x,y)` that's similar to `DISP` but works for functions with two arguments.

W *7. Let `GENERIC_MAX` be the following macro:

```
#define GENERIC_MAX(type)          \
type type##_max(type x, type y)    \
{                                  \
  return x > y ? x : y;            \
}
```

(a) Show the preprocessor's expansion of `GENERIC_MAX(long)`.
(b) Explain why `GENERIC_MAX` doesn't work for basic types such as `unsigned long`.
(c) Describe a technique that would allow us to use `GENERIC_MAX` with basic types such as `unsigned long`. *Hint:* Don't change the definition of `GENERIC_MAX`.

*8. Suppose we want a macro that expands into a string containing the current line number and file name. In other words, we'd like to write

```
const char *str = LINE_FILE;
```

and have it expand into

```
const char *str = "Line 10 of file foo.c";
```

where `foo.c` is the file containing the program and 10 is the line on which the invocation of `LINE_FILE` appears. *Warning:* This exercise is for experts only. Be sure to read the Q&A section carefully before attempting!

9. Write the following parameterized macros.

(a) `CHECK(x,y,n)` – Has the value 1 if both `x` and `y` fall between 0 and `n` – 1, inclusive.
(b) `MEDIAN(x,y,z)` – Finds the median of `x`, `y`, and `z`.
(c) `POLYNOMIAL(x)` – Computes the polynomial $3x^5 + 2x^4 - 5x^3 - x^2 + 7x - 6$.

10. Functions can often—but not always—be written as parameterized macros. Discuss what characteristics of a function would make it unsuitable as a macro.

11. (C99) C programmers often use the `fprintf` function to write error messages:

`fprintf` function ➤*22.3*

```
fprintf(stderr, "Range error: index = %d\n", index);
```

`stderr` stream ➤*22.1*

`stderr` is C's "standard error" stream; the remaining arguments are the same as those for `printf`, starting with the format string. Write a macro named `ERROR` that generates the call of `fprintf` shown above when given a format string and the items to be displayed:

```
ERROR("Range error: index = %d\n", index);
```

Section 14.4

W 12. Suppose that the macro `M` has been defined as follows:

```
#define M 10
```

Which of the following tests will fail?

(a) `#if M`
(b) `#ifdef M`
(c) `#ifndef M`
(d) `#if defined(M)`
(e) `#if !defined(M)`

13. (a) Show what the following program will look like after preprocessing. You may ignore any lines added to the program as a result of including the `<stdio.h>` header.

```
#include <stdio.h>

#define N 100

void f(void);

int main(void)
{
  f();
#ifdef N
#undef N
#endif
  return 0;
}

void f(void)
{
#if defined(N)
  printf("N is %d\n", N);
#else
  printf("N is undefined\n");
#endif
}
```

(b) What will be the output of this program?

W *14. Show what the following program will look like after preprocessing. Some lines of the program may cause compilation errors; find all such errors.

```
#define N = 10
#define INC(x) x+1
#define SUB (x,y) x-y
#define SQR(x) ((x)*(x))
#define CUBE(x) (SQR(x)*(x))
#define M1(x,y) x##y
#define M2(x,y) #x #y

int main(void)
{
  int a[N], i, j, k, m;

#ifdef N
  i = j;
#else
  j = i;
#endif

  i = 10 * INC(j);
```

```
    i = SUB(j, k);
    i = SQR(SQR(j));
    i = CUBE(j);
    i = M1(j, k);
    puts(M2(i, j));

#undef SQR
    i = SQR(j);
#define SQR
    i = SQR(j);

    return 0;
}
```

15. Suppose that a program needs to display messages in either English, French, or Spanish. Using conditional compilation, write a program fragment that displays one of the following three messages, depending on whether or not the specified macro is defined:

```
Insert Disk 1          (if ENGLISH is defined)
Inserez Le Disque 1    (if FRENCH is defined)
Inserte El Disco 1     (if SPANISH is defined)
```

Section 14.5

*16. (C99) Assume that the following macro definitions are in effect:

```
#define IDENT(x) PRAGMA(ident #x)
#define PRAGMA(x) _Pragma(#x)
```

What will the following line look like after macro expansion?

```
IDENT(foo)
```

15 Writing Large Programs

Around computers it is difficult to find the correct unit of time to measure progress. Some cathedrals took a century to complete. Can you imagine the grandeur and scope of a program that would take as long?

Although some C programs are small enough to be put in a single file, most aren't. Programs that consist of more than one file are the rule rather than the exception. In this chapter, we'll see that a typical program consists of several source files and usually some header files as well. Source files contain definitions of functions and external variables; header files contain information to be shared among source files. Section 15.1 discusses source files, while Section 15.2 covers header files. Section 15.3 describes how to divide a program into source files and header files. Section 15.4 then shows how to "build" (compile and link) a program that consists of more than one file, and how to "rebuild" a program after part of it has been changed.

15.1 Source Files

Up to this point, we've assumed that a C program consists of a single file. In fact, a program may be divided among any number of *source files*. By convention, source files have the extension `.c`. Each source file contains part of the program, primarily definitions of functions and variables. One source file must contain a function named `main`, which serves as the starting point for the program.

For example, suppose that we want to write a simple calculator program that evaluates integer expressions entered in Reverse Polish notation (RPN), in which operators follow operands. If the user enters an expression such as

```
30 5 - 7 *
```

we want the program to print its value (175, in this case). Evaluating an RPN expression is easy if we have the program read the operands and operators, one by one, using a stack to keep track of intermediate results. If the program reads a

stacks ➤ 10.2

number, we'll have it push the number onto the stack. If it reads an operator, we'll have it pop two numbers from the stack, perform the operation, and then push the result back onto the stack. When the program reaches the end of the user's input, the value of the expression will be on the stack. For example, the program will evaluate the expression `30 5 - 7 *` in the following way:

1. Push 30 onto the stack.
2. Push 5 onto the stack.
3. Pop the top two numbers from the stack, subtract 5 from 30, giving 25, and then push the result back onto the stack.
4. Push 7 onto the stack.
5. Pop the top two numbers from the stack, multiply them, and then push the result back onto the stack.

After these steps, the stack will contain the value of the expression (175).

Turning this strategy into a program isn't hard. The program's `main` function will contain a loop that performs the following actions:

```
Read a "token" (a number or an operator).
If the token is a number, push it onto the stack.
If the token is an operator, pop its operands from the stack, perform the operation, and then push the result back onto the stack.
```

When dividing a program like this one into files, it makes sense to put related functions and variables into the same file. The function that reads tokens could go into one source file (`token.c`, say), together with any functions that have to do with tokens. Stack-related functions such as `push`, `pop`, `make_empty`, `is_empty`, and `is_full` could go into a different file, `stack.c`. The variables that represent the stack would also go into `stack.c`. The `main` function would go into yet another file, `calc.c`.

Splitting a program into multiple source files has significant advantages:

- Grouping related functions and variables into a single file helps clarify the structure of the program.
- Each source file can be compiled separately—a great time-saver if the program is large and must be changed frequently (which is common during program development).
- Functions are more easily reused in other programs when grouped in separate source files. In our example, splitting off `stack.c` and `token.c` from the `main` function makes it simpler to reuse the stack functions and token functions in the future.

15.2 Header Files

When we divide a program into several source files, problems arise: How can a function in one file call a function that's defined in another file? How can a func-

tion access an external variable in another file? How can two files share the same macro definition or type definition? The answer lies with the `#include` directive, which makes it possible to share information—function prototypes, macro definitions, type definitions, and more—among any number of source files.

The `#include` directive tells the preprocessor to open a specified file and insert its contents into the current file. Thus, if we want several source files to have access to the same information, we'll put that information in a file and then use `#include` to bring the file's contents into each of the source files. Files that are included in this fashion are called ***header files*** (or sometimes ***include files***); I'll discuss them in more detail later in this section. By convention, header files have the extension `.h`.

Note: The C standard uses the term "source file" to refer to all files written by the programmer, including both `.c` and `.h` files. I'll use "source file" to refer to `.c` files only.

The `#include` Directive

The `#include` directive has two primary forms. The first form is used for header files that belong to C's own library:

`#include` directive (form 1)

```
#include <filename>
```

The second form is used for all other header files, including any that we write:

`#include` directive (form 2)

```
#include "filename"
```

Q&A

The difference between the two is a subtle one having to do with how the compiler locates the header file. Here are the rules that most compilers follow:

- `#include <`*filename*`>`: Search the directory (or directories) in which system header files reside. (On UNIX systems, for example, system header files are usually kept in the directory `/usr/include`.)
- `#include "`*filename*`"`: Search the current directory, then search the directory (or directories) in which system header files reside.

The places to be searched for header files can usually be altered, often by a command-line option such as `-I`*path*.

Don't use brackets when including header files that you have written:

```
#include <myheader.h>    /*** WRONG ***/
```

The preprocessor will probably look for `myheader.h` where the system header files are kept (and, of course, won't find it).

The file name in an `#include` directive may include information that helps locate the file, such as a directory path or drive specifier:

```
#include "c:\cprogs\utils.h"   /* Windows path */

#include "/cprogs/utils.h"     /* UNIX path */
```

Although the quotation marks in the `#include` directive make file names look like string literals, the preprocessor doesn't treat them that way. (That's fortunate, since `\c` and `\u`—which appear in the Windows example—would be treated as escape sequences in a string literal.)

portability tip

It's usually best not to include path or drive information in `#include` *directives. Such information makes it difficult to compile a program when it's transported to another machine or, worse, another operating system.*

For example, the following Windows `#include` directives specify drive and/or path information that may not always be valid:

```
#include "d:utils.h"
#include "\cprogs\include\utils.h"
#include "d:\cprogs\include\utils.h"
```

The following directives are better: they don't mention specific drives, and paths are relative rather than absolute:

```
#include "utils.h"
#include "..\include\utils.h"
```

The `#include` directive has a third form that's used less often than the other two:

`#include` directive (form 3)

`#include` *tokens*

preprocessing tokens ➤ *14.3*

where *tokens* is any sequence of preprocessing tokens. The preprocessor will scan the tokens and replace any macros that it finds. After macro replacement, the resulting directive must match one of the other forms of `#include`. The advantage of the third kind of `#include` is that the file name can be defined by a macro rather than being "hard-coded" into the directive itself, as the following example shows:

```
#if defined(IA32)
  #define CPU_FILE "ia32.h"
#elif defined(IA64)
  #define CPU_FILE "ia64.h"
#elif defined(AMD64)
  #define CPU_FILE "amd64.h"
#endif

#include CPU_FILE
```

Sharing Macro Definitions and Type Definitions

Most large programs contain macro definitions and type definitions that need to be shared by several source files (or, in the most extreme case, by *all* source files). These definitions should go into header files.

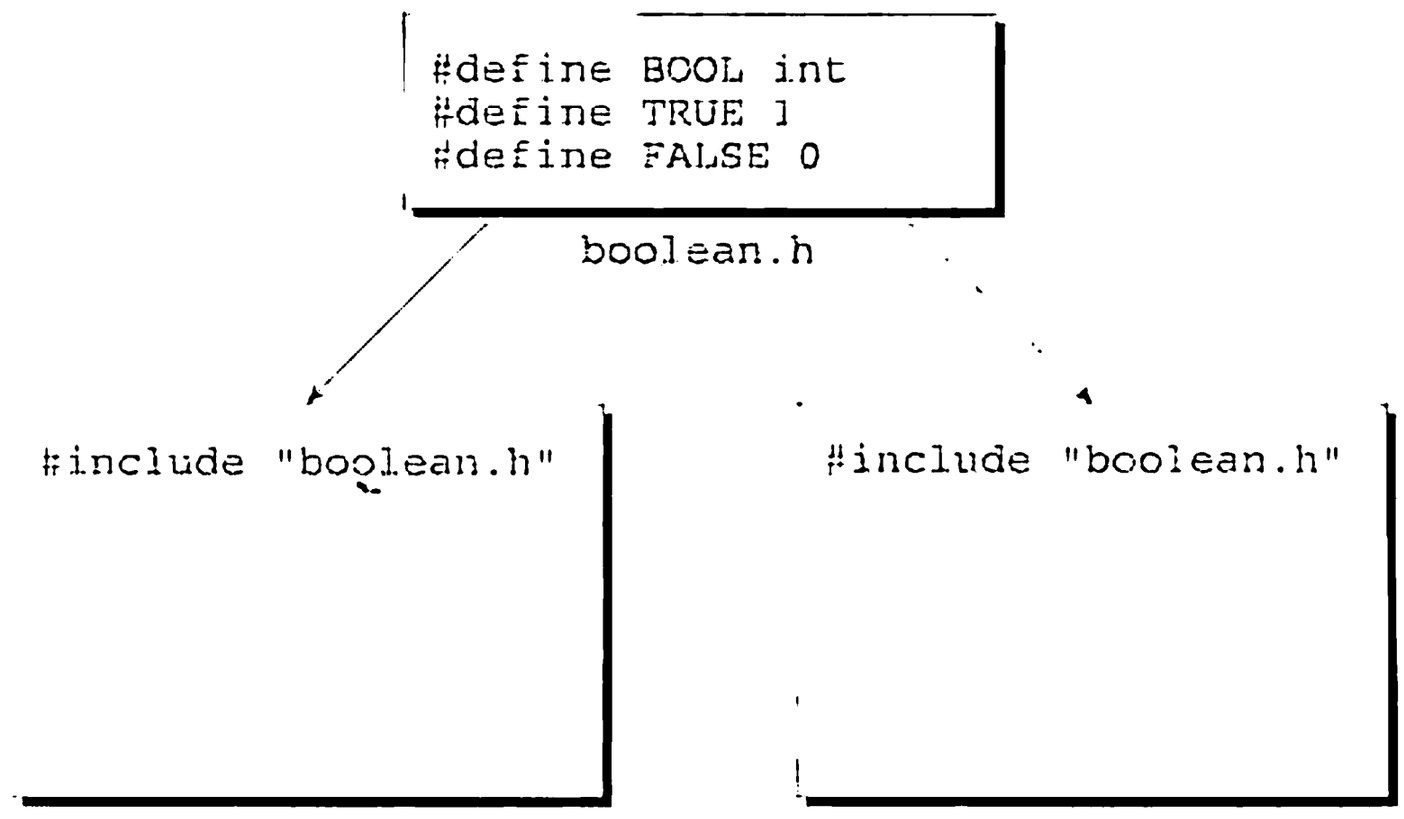
For example, suppose that we're writing a program that uses macros named `BOOL`, `TRUE`, and `FALSE`. (There's no need for these in C99, of course, because the `<stdbool.h>` header defines similar macros.) Instead of repeating the definitions of these macros in each source file that needs them, it makes more sense to put the definitions in a header file with a name like `boolean.h`:

```
#define BOOL int
#define TRUE 1
#define FALSE 0
```

Any source file that requires these macros will simply contain the line

```
#include "boolean.h"
```

In the following figure, two files include `boolean.h`:



Type definitions are also common in header files. For example, instead of defining a `BOOL` macro, we might use `typedef` to create a `Bool` type. If we do, the `boolean.h` file will have the following appearance:

```
#define TRUE 1
#define FALSE 0
typedef int Bool;
```

Putting definitions of macros and types in header files has some clear advantages. First, we save time by not having to copy the definitions into the source files where they're needed. Second, the program becomes easier to modify. Changing the definition of a macro or type requires only that we edit a single header file: we don't have to modify the many source files in which the macro or type is used. Third, we don't have to worry about inconsistencies caused by source files containing different definitions of the same macro or type.

Sharing Function Prototypes

Suppose that a source file contains a call of a function `f` that's defined in another file, `foo.c`. Calling `f` without declaring it first is risky. Without a prototype to rely on, the compiler is forced to assume that `f`'s return type is `int` and that the number of parameters matches the number of arguments in the call of `f`. The arguments themselves are converted automatically to a kind of "standard form" by the default argument promotions. The compiler's assumptions may well be wrong, but it has no way to check them, since it compiles only one file at a time. If the assumptions are incorrect, the program probably won't work, and there won't be any clues as to why it doesn't. (For this reason, C99 prohibits calling a function for which the compiler has not yet seen a declaration or definition.)

default argument promotions ➤ 93

When calling a function `f` that's defined in another file, always make sure that the compiler has seen a prototype for `f` prior to the call.

Our first impulse is to declare `f` in the file where it's called. That solves the problem but can create a maintenance nightmare. Suppose that the function is called in fifty different source files. How can we ensure that `f`'s prototypes are the same in all the files? How can we guarantee that they match the definition of `f` in `foo.c`? If `f` should change later, how can we find all the files where it's used?

Q&A

The solution is obvious: put `f`'s prototype in a header file, then include the header file in all the places where `f` is called. Since `f` is defined in `foo.c`, let's name the header file `foo.h`. In addition to including `foo.h` in the source files where `f` is called, we'll need to include it in `foo.c`, enabling the compiler to check that `f`'s prototype in `foo.h` matches its definition in `foo.c`.

Always include the header file declaring a function `f` in the source file that contains `f`'s definition. Failure to do so can cause hard-to-find bugs, since calls of `f` elsewhere in the program may not match `f`'s definition.

If `foo.c` contains other functions, most of them should be declared in the same header file as `f`. After all, the other functions in `foo.c` are presumably related to `f`; any file that contains a call of `f` probably needs some of the other functions in `foo.c`. Functions that are intended for use only within `foo.c` shouldn't be declared in a header file, however; to do so would be misleading.

To illustrate the use of function prototypes in header files, let's return to the RPN calculator of Section 15.1. The `stack.c` file will contain definitions of the `make_empty`, `is_empty`, `is_full`, `push`, and `pop` functions. The following prototypes for these functions should go in the `stack.h` header file:

```
void make_empty(void);
int is_empty(void);
```

```
int is_full(void);
void push(int i);
int pop(void);
```

(To avoid complicating the example, `is_empty` and `is_full` will return `int` values instead of Boolean values.) We'll include `stack.h` in `calc.c` to allow the compiler to check any calls of stack functions that appear in the latter file. We'll also include `stack.h` in `stack.c` so the compiler can verify that the prototypes in `stack.h` match the definitions in `stack.c`. The following figure shows `stack.h`, `stack.c`, and `calc.c`:

`stack.h`

```
void make_empty(void);
int is_empty(void);
int is_full(void);
void push(int i);
int pop(void);
```

`calc.c`

```
#include "stack.h"

int main(void)
{
  make_empty();
  …
}
```

`stack.c`

```
#include "stack.h"

int contents[100];
int top = 0;

void make_empty(void)
{ … }

int is_empty(void)
{ … }

int is_full(void)
{ … }

void push(int i)
{ … }

int pop(void)
{ … }
```

Sharing Variable Declarations

external variables ➤ *10.2*

External variables can be shared among files in much the same way functions are. To share a function, we put its *definition* in one source file, then put *declarations* in other files that need to call the function. Sharing an external variable is done in much the same way.

Up to this point, we haven't needed to distinguish between a variable's declaration and its definition. To declare a variable `i`, we've written

```
int i;              /* declares i and defines it as well */
```

which not only declares `i` to be a variable of type `int`, but defines `i` as well, by causing the compiler to set aside space for `i`. To declare `i` without defining it, we must put the keyword `extern` at the beginning of its declaration:

extern keyword ➤ 18.2

```
extern int i;   /* declares i without defining it */
```

`extern` informs the compiler that `i` is defined elsewhere in the program (most likely in a different source file), so there's no need to allocate space for it.

`extern` works with variables of all types. When we use it in the declaration of an array, we can omit the length of the array:

Q&A

```
extern int a[];
```

Since the compiler doesn't allocate space for `a` at this time, there's no need for it to know `a`'s length.

To share a variable `i` among several source files, we first put a definition of `i` in one file:

```
int i;
```

If `i` needs to be initialized, the initializer would go here. When this file is compiled, the compiler will allocate storage for `i`. The other files will contain declarations of `i`:

```
extern int i;
```

By declaring `i` in each file, it becomes possible to access and/or modify `i` within those files. Because of the word `extern`, however, the compiler doesn't allocate additional storage for `i` each time one of the files is compiled.

When a variable is shared among files, we'll face a challenge similar to one that we had with shared functions: ensuring that all declarations of a variable agree with the definition of the variable.

When declarations of the same variable appear in different files, the compiler can't check that the declarations match the variable's definition. For example, one file may contain the definition

```
int i;
```

while another file contains the declaration

```
extern long i;
```

An error of this kind can cause the program to behave unpredictably.

To avoid inconsistency, declarations of shared variables are usually put in header files. A source file that needs access to a particular variable can then include the appropriate header file. In addition, each header file that contains a

variable declaration is included in the source file that contains the variable's definition, enabling the compiler to check that the two match.

Although sharing variables among files is a long-standing practice in the C world, it has significant disadvantages. In Section 19.2, we'll see what the problems are and learn how to design programs that don't need shared variables.

Nested Includes

A header file may itself contain `#include` directives. Although this practice may seem a bit odd, it can be quite useful in practice. Consider the `stack.h` file, which contains the following prototypes:

```
int is_empty(void);
int is_full(void);
```

Since these functions return only 0 or 1, it's a good idea to declare their return type to be `Bool` instead of `int`, where `Bool` is the type that we defined earlier in this section:

```
Bool is_empty(void);
Bool is_full(void);
```

Of course, we'll need to include the `boolean.h` file in `stack.h` so that the definition of `Bool` is available when `stack.h` is compiled. (In C99, we'd include `<stdbool.h>` instead of `boolean.h` and declare the return types of the two functions to be `bool` rather than `Bool`.)

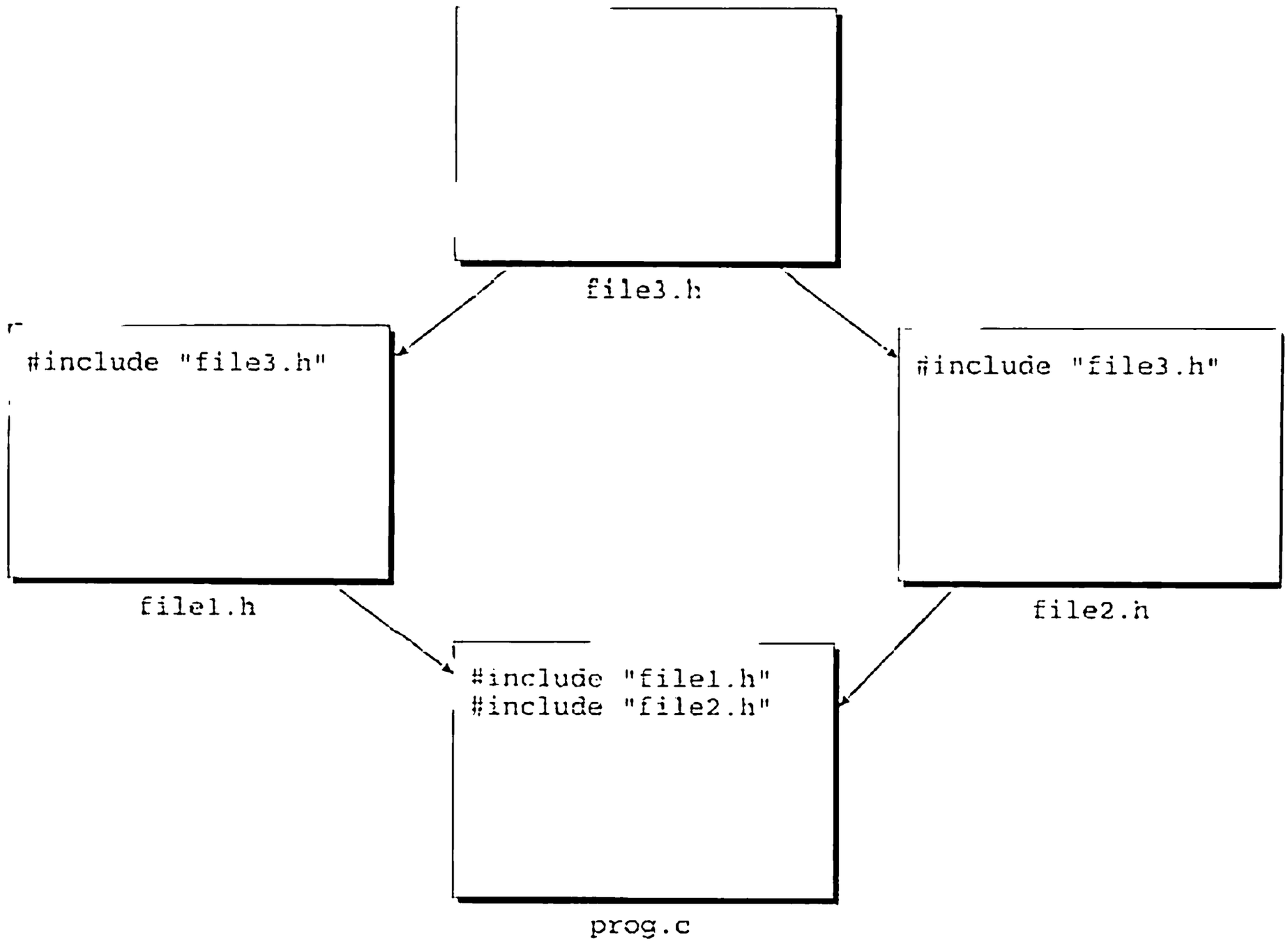
Traditionally, C programmers shun nested includes. (Early versions of C didn't allow them at all.) However, the bias against nested includes has largely faded away, in part because nested includes are common practice in C++.

Protecting Header Files

If a source file includes the same header file twice, compilation errors may result. This problem is common when header files include other header files. For example, suppose that `file1.h` includes `file3.h`, `file2.h` includes `file3.h`, and `prog.c` includes both `file1.h` and `file2.h` (see the figure at the top of the next page). When `prog.c` is compiled, `file3.h` will be compiled twice.

Including the same header file twice doesn't always cause a compilation error. If the file contains only macro definitions, function prototypes, and/or variable declarations, there won't be any difficulty. If the file contains a type definition, however, we'll get a compilation error.

Just to be safe, it's probably a good idea to protect all header files against multiple inclusion; that way, we can add type definitions to a file later without the risk that we might forget to protect the file. In addition, we might save some time during program development by avoiding unnecessary recompilation of the same header file.



To protect a header file, we'll enclose the contents of the file in an `#ifndef`-`#endif` pair. For example, the `boolean.h` file could be protected in the following way:

```
#ifndef BOOLEAN_H
#define BOOLEAN_H

#define TRUE 1
#define FALSE 0
typedef int Bool;

#endif
```

When this file is included the first time, the `BOOLEAN_H` macro won't be defined, so the preprocessor will allow the lines between `#ifndef` and `#endif` to stay. But if the file should be included a second time, the preprocessor will remove the lines between `#ifndef` and `#endif`.

The name of the macro (`BOOLEAN_H`) doesn't really matter. However, making it resemble the name of the header file is a good way to avoid conflicts with other macros. Since we can't name the macro `BOOLEAN.H` (identifiers can't contain periods), a name such as `BOOLEAN_H` is a good alternative.

`#error` Directives in Header Files

`#error` directives ➤ *14.5*

`#error` directives are often put in header files to check for conditions under which the header file shouldn't be included. For example, suppose that a header

file uses a feature that didn't exist prior to the original C89 standard. To prevent the header file from being used with older, nonstandard compilers, it could contain an #ifndef directive that tests for the existence of the __STDC__ macro:

__STDC__ macro ➤ 14.3

```
#ifndef __STDC__
#error This header requires a Standard C compiler
#endif
```

15.3 Dividing a Program into Files

Let's now use what we know about header files and source files to develop a simple technique for dividing a program into files. We'll concentrate on functions, but the same principles apply to external variables as well. We'll assume that the program has already been designed; that is, we've decided what functions the program will need and how to arrange the functions into logically related groups. (We'll discuss program design in Chapter 19.)

Here's how we'll proceed. Each set of functions will go into a separate source file (let's use the name `foo.c` for one such file). In addition, we'll create a header file with the same name as the source file, but with the extension `.h` (`foo.h`, in our case). Into `foo.h`, we'll put prototypes for the functions defined in `foo.c`. (Functions that are designed for use only within `foo.c` need not—and should not—be declared in `foo.h`. The `read_char` function in our next program is an example.) We'll include `foo.h` in each source file that needs to call a function defined in `foo.c`. Moreover, we'll include `foo.h` in `foo.c` so that the compiler can check that the function prototypes in `foo.h` are consistent with the definitions in `foo.c`.

The `main` function will go in a file whose name matches the name of the program—if we want the program to be known as `bar`, then `main` should be in the file `bar.c`. It's possible that there are other functions in the same file as `main`, so long as they're not called from other files in the program.

PROGRAM Text Formatting

To illustrate the technique that we've just discussed, let's apply it to a small text-formatting program named `justify`. As sample input to `justify`, we'll use a file named `quote` that contains the following (poorly formatted) quotation from "The development of the C programming language" by Dennis M. Ritchie (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):

```
   C     is quirky,  flawed,     and  an
enormous   success.      Although accidents of   history
 surely  helped,   it evidently    satisfied   a   need

    for  a   system  implementation    language    efficient
```

```
enough to displace assembly language,
  yet sufficiently abstract and fluent to describe
 algorithms and interactions in a wide variety
of environments.
                    -- Dennis M. Ritchie
```

To run the program from a UNIX or Windows prompt. we'd enter the command

```
justify <quote
```

The < symbol informs the operating system that `justify` will read from the file `quote` instead of accepting input from the keyboard. This feature, supported by UNIX, Windows, and other operating systems, is called ***input redirection***. When given the `quote` file as input, the `justify` program will produce the following output:

Input redirection ➤22.1

```
C is quirky, flawed, and an enormous success. Although
accidents of history surely helped, it evidently satisfied a
need for a system implementation language efficient enough
to displace assembly language, yet sufficiently abstract and
fluent to describe algorithms and interactions in a wide
variety of environments. -- Dennis M. Ritchie
```

The output of `justify` will normally appear on the screen, but we can save it in a file by using ***output redirection***:

output redirection ➤22.1

```
justify <quote >newquote
```

The output of `justify` will go into the file `newquote`.

In general, `justify`'s output should be identical to its input, except that extra spaces and blank lines are deleted, and lines are filled and justified. "Filling" a line means adding words until one more word would cause the line to overflow. "Justifying" a line means adding extra spaces between words so that each line has exactly the same length (60 characters). Justification must be done so that the space between words in a line is equal (or as nearly equal as possible). The last line of the output won't be justified.

We'll assume that no word is longer than 20 characters. (A punctuation mark is considered part of the word to which it is adjacent.) That's a bit restrictive, of course, but once the program is written and debugged we can easily increase this limit to the point that it would virtually never be exceeded. If the program encounters a longer word, it must ignore all characters after the first 20, replacing them with a single asterisk. For example, the word

```
antidisestablishmentarianism
```

would be printed as

```
antidisestablishment*
```

Now that we understand what the program should do, it's time to think about a design. We'll start by observing that the program can't write the words one by one as they're read. Instead, it will have to store them in a "line buffer" until there are enough to fill a line. After further reflection, we decide that the heart of the program will be a loop that goes something like this:

```
for (;;) {
  read word;
  if (can't read word) {
    write contents of line buffer without justification;
    terminate program;
  }

  if (word doesn't fit in line buffer) {
    write contents of line buffer with justification;
    clear line buffer;
  }
  add word to line buffer;
}
```

Since we'll need functions that deal with words and functions that deal with the line buffer, let's split the program into three source files, putting all functions related to words in one file (`word.c`) and all functions related to the line buffer in another file (`line.c`). A third file (`justify.c`) will contain the `main` function. In addition to these files, we'll need two header files, `word.h` and `line.h`. The `word.h` file will contain prototypes for the functions in `word.c`; `line.h` will play a similar role for `line.c`.

By examining the main loop, we see that the only word-related function that we'll need is a `read_word` function. (If `read_word` can't read a word because it's reached the end of the input file, we'll have it signal the main loop by pretending to read an "empty" word.) Consequently, the `word.h` file is a small one:

word.h

```
#ifndef WORD_H
#define WORD_H

/*************************************************
 * read_word: Reads the next word from the input and *
 *            stores it in word. Makes word empty if no *
 *            word could be read because of end-of-file. *
 *            Truncates the word if its length exceeds *
 *            len.                                   *
 *************************************************/
void read_word(char *word, int len);

#endif
```

Notice how the `WORD_H` macro protects `word.h` from being included more than once. Although `word.h` doesn't really need it, it's good practice to protect all header files in this way.

The `line.h` file won't be as short as `word.h`. Our outline of the main loop reveals the need for functions that perform the following operations:

Write contents of line buffer without justification
Determine how many characters are left in line buffer
Write contents of line buffer with justification
Clear line buffer
Add word to line buffer

We'll call these functions `flush_line`, `space_remaining`, `write_line`, `clear_line`, and `add_word`. Here's what the `line.h` header file will look like:

line.h

```
#ifndef LINE_H
#define LINE_H

/************************************************************
 * clear_line: Clears the current line.                     *
 ************************************************************/
void clear_line(void);

/************************************************************
 * add_word: Adds word to the end of the current line.      *
 *           If this is not the first word on the line,     *
 *           puts one space before word.                    *
 ************************************************************/
void add_word(const char *word);

/************************************************************
 * space_remaining: Returns the number of characters left   *
 *                  in the current line.                    *
 ************************************************************/
int space_remaining(void);

/************************************************************
 * write_line: Writes the current line with                 *
 *             justification.                               *
 ************************************************************/
void write_line(void);

/************************************************************
 * flush_line: Writes the current line without              *
 *             justification. If the line is empty, does    *
 *             nothing.                                     *
 ************************************************************/
void flush_line(void);

#endif
```

Before we write the `word.c` and `line.c` files, we can use the functions declared in `word.h` and `line.h` to write `justify.c`, the main program. Writing this file is mostly a matter of translating our original loop design into C.

justify.c

```
/* Formats a file of text */

#include <string.h>
#include "line.h"
#include "word.h"

#define MAX_WORD_LEN 20

int main(void)
{
  char word[MAX_WORD_LEN+2];
  int word_len;

  clear_line();
  for (;;) {
    read_word(word, MAX_WORD_LEN+1);
    word_len = strlen(word);
    if (word_len == 0) {
      flush_line();
      return 0;
    }
    if (word_len > MAX_WORD_LEN)
      word[MAX_WORD_LEN] = '*';
    if (word_len + 1 > space_remaining()) {
      write_line();
      clear_line();
    }
    add_word(word);
  }
}
```

Including both `line.h` and `word.h` gives the compiler access to the function prototypes in both files as it compiles `justify.c`.

`main` uses a trick to handle words that exceed 20 characters. When it calls `read_word`, `main` tells it to truncate any word that exceeds 21 characters. After `read_word` returns, `main` checks whether `word` contains a string that's longer than 20 characters. If so, the word that was read must have been at least 21 characters long (before truncation), so `main` replaces the word's 21st character by an asterisk.

Now it's time to write `word.c`. Although the `word.h` header file has a prototype for only one function, `read_word`, we can put additional functions in `word.c` if we need to. As it turns out, `read_word` is easier to write if we add a small "helper" function, `read_char`. We'll assign `read_char` the task of reading a single character and, if it's a new-line character or tab, converting it to a space. Having `read_word` call `read_char` instead of `getchar` solves the problem of treating new-line characters and tabs as spaces.

Here's the `word.c` file:

word.c

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

```
int read_char(void)
{
  int ch = getchar();

  if (ch == '\n' || ch == '\t')
    return ' ';
  return ch;
}
```

```
void read_word(char *word, int len)
{
  int ch, pos = 0;

  while ((ch = read_char()) == ' ')
    ;
  while (ch != ' ' && ch != EOF) {
    if (pos < len)
      word[pos++] = ch;
    ch = read_char();
  }
  word[pos] = '\0';
}
```

Before we discuss `read_word`, a couple of comments are in order concerning the use of `getchar` in the `read_char` function. First, `getchar` returns an `int` value instead of a `char` value; that's why the variable `ch` in `read_char` is declared to have type `int` and why the return type of `read_char` is `int`. Also, `getchar` returns the value `EOF` when it's unable to continue reading (usually because it has reached the end of the input file).

EOF macro ➤22.4

`read_word` consists of two loops. The first loop skips over spaces, stopping at the first nonblank character. (`EOF` isn't a blank, so the loop stops if it reaches the end of the input file.) The second loop reads characters until encountering a space or `EOF`. The body of the loop stores the characters in `word` until reaching the `len` limit. After that, the loop continues reading characters but doesn't store them. The final statement in `read_word` ends the word with a null character, thereby making it a string. If `read_word` encounters `EOF` before finding a nonblank character, `pos` will be 0 at the end, making `word` an empty string.

The only file left is `line.c`, which supplies definitions of the functions declared in the `line.h` file. `line.c` will also need variables to keep track of the state of the line buffer. One variable, `line`, will store the characters in the current line. Strictly speaking, `line` is the only variable we need. For speed and convenience, however, we'll use two other variables: `line_len` (the number of characters in the current line) and `num_words` (the number of words in the current line).

Here's the `line.c` file:

line.c

```
#include <stdio.h>
#include <string.h>
#include "line.h"
```

```
#define MAX_LINE_LEN 60

char line[MAX_LINE_LEN+1];
int line_len = 0;
int num_words = 0;

void clear_line(void)
{
  line[0] = '\0';
  line_len = 0;
  num_words = 0;
}

void add_word(const char *word)
{
  if (num_words > 0) {
    line[line_len] = ' ';
    line[line_len+1] = '\0';
    line_len++;
  }
  strcat(line, word);
  line_len += strlen(word);
  num_words++;
}

int space_remaining(void)
{
  return MAX_LINE_LEN - line_len;
}

void write_line(void)
{
  int extra_spaces, spaces_to_insert, i, j;

  extra_spaces = MAX_LINE_LEN - line_len;
  for (i = 0; i < line_len; i++) {
    if (line[i] != ' ')
      putchar(line[i]);
    else {
      spaces_to_insert = extra_spaces / (num_words - 1);
      for (j = 1; j <= spaces_to_insert + 1; j++)
        putchar(' ');
      extra_spaces -= spaces_to_insert;
      num_words--;
    }
  }
  putchar('\n');
}

void flush_line(void)
{
  if (line_len > 0)
    puts(line);
}
```

Most of the functions in `line.c` are easy to write. The only tricky one is `write_line`, which writes a line with justification. `write_line` writes the characters in `line` one by one, pausing at the space between each pair of words to write additional spaces if needed. The number of additional spaces is stored in `spaces_to_insert`, which has the value `extra_spaces / (num_words - 1)`, where `extra_spaces` is initially the difference between the maximum line length and the actual line length. Since `extra_spaces` and `num_words` change after each word is printed, `spaces_to_insert` will change as well. If `extra_spaces` is 10 initially and `num_words` is 5, then the first word will be followed by 2 extra spaces, the second by 2, the third by 3, and the fourth by 3.

15.4 Building a Multiple-File Program

In Section 2.1, we examined the process of compiling and linking a program that fits into a single file. Let's expand that discussion to cover multiple-file programs. Building a large program requires the same basic steps as building a small one:

- *Compiling.* Each source file in the program must be compiled separately. (Header files don't need to be compiled; the contents of a header file are automatically compiled whenever a source file that includes it is compiled.) For each source file, the compiler generates a file containing object code. These files—known as *object files*—have the extension `.o` in UNIX and `.obj` in Windows.
- *Linking.* The linker combines the object files created in the previous step—along with code for library functions—to produce an executable file. Among other duties, the linker is responsible for resolving external references left behind by the compiler. (An external reference occurs when a function in one file calls a function defined in another file or accesses a variable defined in another file.)

Most compilers allow us to build a program in a single step. With the GCC compiler, for example, we'd use the following command to build the `justify` program of Section 15.3:

```
gcc -o justify justify.c line.c word.c
```

The three source files are first compiled into object code. The object files are then automatically passed to the linker, which combines them into a single file. The `-o` option specifies that we want the executable file to be named `justify`.

Makefiles

Putting the names of all the source files on the command line quickly gets tedious. Worse still, we could waste a lot of time when rebuilding a program if we recompile all source files, not just the ones that were affected by our most recent changes.

To make it easier to build large programs, UNIX originated the concept of the *makefile*, a file containing the information necessary to build a program. A makefile not only lists the files that are part of the program, but also describes *dependencies* among the files. Suppose that the file `foo.c` includes the file `bar.h`. We say that `foo.c` "depends" on `bar.h`, because a change to `bar.h` will require us to recompile `foo.c`.

Here's a UNIX makefile for the `justify` program. The makefile uses GCC for compilation and linking:

```
justify: justify.o word.o line.o
        gcc -o justify justify.o word.o line.o

justify.o: justify.c word.h line.h
        gcc -c justify.c

word.o: word.c word.h
        gcc -c word.c

line.o: line.c line.h
        gcc -c line.c
```

There are four groups of lines; each group is known as a *rule*. The first line in each rule gives a *target* file, followed by the files on which it depends. The second line is a *command* to be executed if the target should need to be rebuilt because of a change to one of its dependent files. Let's look at the first two rules; the last two are similar.

In the first rule, `justify` (the executable file) is the target:

```
justify: justify.o word.o line.o
        gcc -o justify justify.o word.o line.o
```

The first line states that `justify` depends on the files `justify.o`, `word.o`, and `line.o`; if any one of these three files has changed since the program was last built, then `justify` needs to be rebuilt. The command on the following line shows how the rebuilding is to be done (by using the `gcc` command to link the three object files).

In the second rule, `justify.o` is the target:

```
justify.o: justify.c word.h line.h
        gcc -c justify.c
```

The first line indicates that `justify.o` needs to be rebuilt if there's been a change to `justify.c`, `word.h`, or `line.h`. (The reason for mentioning `word.h` and `line.h` is that `justify.c` includes both these files, so it's potentially affected by a change to either one.) The next line shows how to update `justify.o` (by recompiling `justify.c`). The `-c` option tells the compiler to compile `justify.c` into an object file but not attempt to link it.

Q&A Once we've created a makefile for a program, we can use the `make` utility to build (or rebuild) the program. By checking the time and date associated with each

file in the program, make can determine which files are out of date. It then invokes the commands necessary to rebuild the program.

If you want to give make a try, here are a few details you'll need to know:

- Each command in a makefile must be preceded by a tab character, not a series of spaces. (In our example, the commands appear to be indented eight spaces, but it's actually a single tab character.)
- A makefile is normally stored in a file named `Makefile` (or `makefile`). When the make utility is used, it automatically checks the current directory for a file with one of these names.
- To invoke make, use the command

  `make` *target*

  where *target* is one of the targets listed in the makefile. To build the `justify` executable using our makefile, we would use the command

  ```
  make justify
  ```

- If no target is specified when make is invoked, it will build the target of the first rule. For example, the command

  ```
  make
  ```

  will build the `justify` executable, since `justify` is the first target in our makefile. Except for this special property of the first rule, the order of rules in a makefile is arbitrary.

make is complicated enough that entire books have been written about it, so we won't attempt to delve further into its intricacies. Let's just say that real makefiles aren't usually as easy to understand as our example. There are numerous techniques that reduce the amount of redundancy in makefiles and make them easier to modify; at the same time, though, these techniques greatly reduce their readability.

Not everyone uses makefiles, by the way. Other program maintenance tools are also popular, including the "project files" supported by some integrated development environments.

Errors During Linking

Some errors that can't be detected during compilation will be found during linking. In particular, if the definition of a function or variable is missing from a program, the linker will be unable to resolve external references to it, causing a message such as "*undefined symbol*" or "*undefined reference*."

Errors detected by the linker are usually easy to fix. Here are some of the most common causes:

- ***Misspellings.*** If the name of a variable or function is misspelled, the linker will report it as missing. For example, if the function `read_char` is defined

in the program but called as `read_cahr`, the linker will report that `read_cahr` is missing.

- *Missing files.* If the linker can't find the functions that are in file `foo.c`, it may not know about the file. Check the makefile or project file to make sure that `foo.c` is listed there.
- *Missing libraries.* The linker may not be able to find all library functions used in the program. A classic example occurs in UNIX programs that use the `<math.h>` header. Simply including the header in a program may not be enough: many versions of UNIX require that the `-lm` option be specified when the program is linked, causing the linker to search a system file that contains compiled versions of the `<math.h>` functions. Failing to use this option may cause "undefined reference" messages during linking.

Rebuilding a Program

During the development of a program, it's rare that we'll need to compile all its files. Most of the time, we'll test the program, make a change, then build the program again. To save time, the rebuilding process should recompile only those files that might be affected by the latest change.

Let's assume that we've designed our program in the way outlined in Section 15.3, with a header file for each source file. To see how many files will need to be recompiled after a change, we need to consider two possibilities.

The first possibility is that the change affects a single source file. In that case, only that file must be recompiled. (After that, the entire program will need to be relinked, of course.) Consider the `justify` program. Suppose that we decide to condense the `read_char` function in `word.c` (changes are marked in **bold**):

```
int read_char(void)
{
  int ch = getchar();

  return (ch == '\n' || ch == '\t') ? ' ' : ch;
}
```

This modification doesn't affect `word.h`, so we need only recompile `word.c` and relink the program.

The second possibility is that the change affects a header file. In that case, we should recompile all files that include the header file, since they could potentially be affected by the change. (Some of them might not be, but it pays to be conservative.)

As an example, consider the `read_word` function in the `justify` program. Notice that `main` calls `strlen` immediately after calling `read_word`, in order to determine the length of the word that was just read. Since `read_word` already knows the length of the word (`read_word`'s `pos` variable keeps track of the length), it seems silly to use `strlen`. Modifying `read_word` to return the word's length is easy. First, we change the prototype of `read_word` in `word.h`:

```
/*************************************************************
 * read_word: Reads the next word from the input and         *
 *            stores it in word. Makes word empty if no      *
 *            word could be read because of end-of-file.     *
 *            Truncates the word if its length exceeds       *
 *            len. Returns the number of characters          *
 *            stored.                                        *
 *************************************************************/
int read_word(char *word, int len);
```

Of course. we're careful to change the comment that accompanies `read_word`. Next. we change the definition of `read_word` in `word.c`:

```
int read_word(char *word, int len)
{
  int ch, pos = 0;

  while ((ch = read_char()) == ' ')
    ;
  while (ch != ' ' && ch != EOF) {
    if (pos < len)
      word[pos++] = ch;
    ch = read_char();
  }
  word[pos] = '\0';
  return pos;
}
```

Finally. we modify `justify.c` by removing the include of `<string.h>` and changing `main` as follows:

```
int main(void)
{
  char word[MAX_WORD_LEN+2];
  int word_len;

  clear_line();
  for (;;) {
    word_len = read_word(word, MAX_WORD_LEN+1);
    if (word_len == 0) {
      flush_line();
      return 0;
    }
    if (word_len > MAX_WORD_LEN)
      word[MAX_WORD_LEN] = '*';
    if (word_len + 1 > space_remaining()) {
      write_line();
      clear_line();
    }
    add_word(word);
  }
}
```

Once we've made these changes, we'll rebuild the `justify` program by recompiling `word.c` and `justify.c` and then relinking. There's no need to recompile `line.c`, which doesn't include `word.h` and therefore won't be affected by changes to it. With the GCC compiler, we could use the following command to rebuild the program:

```
gcc -o justify justify.c word.c line.o
```

Note the mention of `line.o` instead of `line.c`.

One of the advantages of using makefiles is that rebuilding is handled automatically. By examining the date of each file, the `make` utility can determine which files have changed since the program was last built. It then recompiles these files, together with all files that depend on them, either directly or indirectly. For example, if we make the indicated changes to `word.h`, `word.c`, and `justify.c` and then rebuild the `justify` program, `make` will perform the following actions:

1. Build `justify.o` by compiling `justify.c` (because `justify.c` and `word.h` were changed).
2. Build `word.o` by compiling `word.c` (because `word.c` and `word.h` were changed).
3. Build `justify` by linking `justify.o`, `word.o`, and `line.o` (because `justify.o` and `word.o` were changed).

Defining Macros Outside a Program

C compilers usually provide some method of specifying the value of a macro at the time a program is compiled. This ability makes it easy to change the value of a macro without editing any of the program's files. It's especially valuable when programs are built automatically using makefiles.

Most compilers (including GCC) support the `-D` option, which allows the value of a macro to be specified on the command line:

```
gcc -DDEBUG=1 foo.c
```

In this example, the `DEBUG` macro is defined to have the value `1` in the program `foo.c`, just as if the line

```
#define DEBUG 1
```

appeared at the beginning of `foo.c`. If the `-D` option names a macro without specifying its value, the value is taken to be 1.

Many compilers also support the `-U` option, which "undefines" a macro as if by using `#undef`. We can use `-U` to undefine a predefined macro or one that was defined earlier in the command line using `-D`.

predefined macros ➤ 14.3

Q & A

Q: **You don't have any examples that use the `#include` directive to include a source file. What would happen if we were to do this?**

A: That's not a good practice, although it's not illegal. Here's an example of the kind of trouble you can get into. Suppose that `foo.c` defines a function `f` that we'll need in `bar.c` and `baz.c`, so we put the directive

```
#include "foo.c"
```

in both `bar.c` and `baz.c`. Each of these files will compile nicely. The problem comes later, when the linker discovers two copies of the object code for `f`. Of course, we would have gotten away with including `foo.c` if only `bar.c` had included it, not `baz.c` as well. To avoid problems, it's best to use `#include` only with header files, not source files.

Q: **What are the exact search rules for the `#include` directive? [p. 351]**

A: That depends on your compiler. The C standard is deliberately vague in its description of `#include`. If the file name is enclosed in *brackets*, the preprocessor looks in a "sequence of implementation-defined places," as the standard obliquely puts it. If the file name is enclosed in *quotation marks*, the file "is searched for in an implementation-defined manner" and, if not found, then searched as if its name had been enclosed in brackets. The reason for this waffling is simple: not all operating systems have hierarchical (tree-like) file systems.

To make matters even more interesting, the standard doesn't require that names enclosed in brackets be file names at all, leaving open the possibility that `#include` directives using `<>` are handled entirely within the compiler.

Q: **I don't understand why each source file needs its own header file. Why not have one big header file containing macro definitions, type definitions, and function prototypes? By including this file, each source file would have access to all the shared information it needs. [p. 354]**

A: The "one big header file" approach certainly works; a number of programmers use it. And it does have an advantage: with only one header file, there are fewer files to manage. For large programs, however, the disadvantages of this approach tend to outweigh its advantages.

Using a single header file provides no useful information to someone reading the program later. With multiple header files, the reader can quickly see what other parts of the program are used by a particular source file.

But that's not all. Since each source file depends on the big header file, changing it will cause all source files to be recompiled—a significant drawback in a large program. To make matters worse, the header file will probably change frequently because of the large amount of information it contains.

Q: **The chapter says that a shared array should be declared as follows:**

```
extern int a[];
```

Since arrays and pointers are closely related, would it be legal to write

```
extern int *a;
```

instead? [p. 356]

A: No. When used in expressions, arrays "decay" into pointers. (We've noticed this behavior when an array name is used as an argument in a function call.) In variable declarations, however, arrays and pointers are distinct types.

Q: **Does it hurt if a source file includes headers that it doesn't really need?**

A: Not unless the header has a declaration or definition that conflicts with one in the source file. Otherwise, the worst that can happen is a minor increase in the time it takes to compile the source file.

Q: **I needed to call a function in the file `foo.c`, so I included the matching header file, `foo.h`. My program compiled, but it won't link. Why?**

A: Compilation and linking are completely separate in C. Header files exist to provide information to the compiler, not the linker. If you want to call a function in `foo.c`, then you have to make sure that `foo.c` is compiled and that the linker is aware that it must search the object file for `foo.c` to find the function. Usually this means naming `foo.c` in the program's makefile or project file.

Q: **If my program calls a function in `<stdio.h>`, does that mean that all functions in `<stdio.h>` will be linked with the program?**

A: No. Including `<stdio.h>` (or any other header) has no effect on linking. In any event, most linkers will link only functions that your program actually needs.

Q: **Where can I get the `make` utility? [p. 367]**

A: `make` is a standard UNIX utility. The GNU version, known as GNU Make, is included in most Linux distributions. It's also available directly from the Free Software Foundation (*www.gnu.org/software/make/*).

Exercises

Section 15.1

1. Section 15.1 listed several advantages of dividing a program into multiple source files.
   (a) Describe several other advantages.
   (b) Describe some disadvantages.

Section 15.2

W 2. Which of the following should *not* be put in a header file? Why not?
   (a) Function prototypes
   (b) Function definitions

(c) Macro definitions
(d) Type definitions

3. We saw that writing `#include <`*file*`>` instead of `#include "`*file*`"` may not work if *file* is one that we've written. Would there be any problem with writing `#include "`*file*`"` instead of `#include <`*file*`>` if *file* is a system header?

4. Assume that `debug.h` is a header file with the following contents:

```
#ifdef DEBUG
#define PRINT_DEBUG(n) printf("Value of " #n ": %d\n", n)
#else
#define PRINT_DEBUG(n)
#endif
```

Let `testdebug.c` be the following source file:

```
#include <stdio.h>

#define DEBUG
#include "debug.h"

int main(void)
{
  int i = 1, j = 2, k = 3;

#ifdef DEBUG
  printf("Output if DEBUG is defined:\n");
#else
  printf("Output if DEBUG is not defined:\n");
#endif

  PRINT_DEBUG(i);
  PRINT_DEBUG(j);
  PRINT_DEBUG(k);
  PRINT_DEBUG(i + j);
  PRINT_DEBUG(2 * i + j - k);

  return 0;
}
```

(a) What is the output when the program is executed?
(b) What is the output if the `#define` directive is removed from `testdebug.c`?
(c) Explain why the output is different in parts (a) and (b).
(d) Is it necessary for the `DEBUG` macro to be defined *before* `debug.h` is included in order for `PRINT_DEBUG` to have the desired effect? Justify your answer.

Section 15.4

5. Suppose that a program consists of three source files—`main.c`, `f1.c`, and `f2.c`—plus two header files, `f1.h` and `f2.h`. All three source files include `f1.h`, but only `f1.c` and `f2.c` include `f2.h`. Write a makefile for this program, assuming that the compiler is `gcc` and that the executable file is to be named `demo`.

W 6. The following questions refer to the program described in Exercise 5.

(a) Which files need to be compiled when the program is built for the first time?
(b) If `f1.c` is changed after the program has been built, which files need to be recompiled?
(c) If `f1.h` is changed after the program has been built, which files need to be recompiled?
(d) If `f2.h` is changed after the program has been built, which files need to be recompiled?

Programming Projects

1. The `justify` program of Section 15.3 justifies lines by inserting extra spaces between words. The way the `write_line` function currently works, the words closer to the end of a line tend to have slightly wider gaps between them than the words at the beginning. (For example, the words closer to the end might have three spaces between them, while the words closer to the beginning might be separated by only two spaces.) Improve the program by having `write_line` alternate between putting the larger gaps at the end of the line and putting them at the beginning of the line.

2. Modify the `justify` program of Section 15.3 by having the `read_word` function (instead of `main`) store the `*` character at the end of a word that's been truncated.

3. Modify the `qsort.c` program of Section 9.6 so that the `quicksort` and `split` functions are in a separate file named `quicksort.c`. Create a header file named `quicksort.h` that contains prototypes for the two functions and have both `qsort.c` and `quicksort.c` include this file.

4. Modify the `remind.c` program of Section 13.5 so that the `read_line` function is in a separate file named `readline.c`. Create a header file named `readline.h` that contains a prototype for the function and have both `remind.c` and `readline.c` include this file.

5. Modify Programming Project 6 from Chapter 10 so that it has separate `stack.h` and `stack.c` files, as described in Section 15.2.

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*.

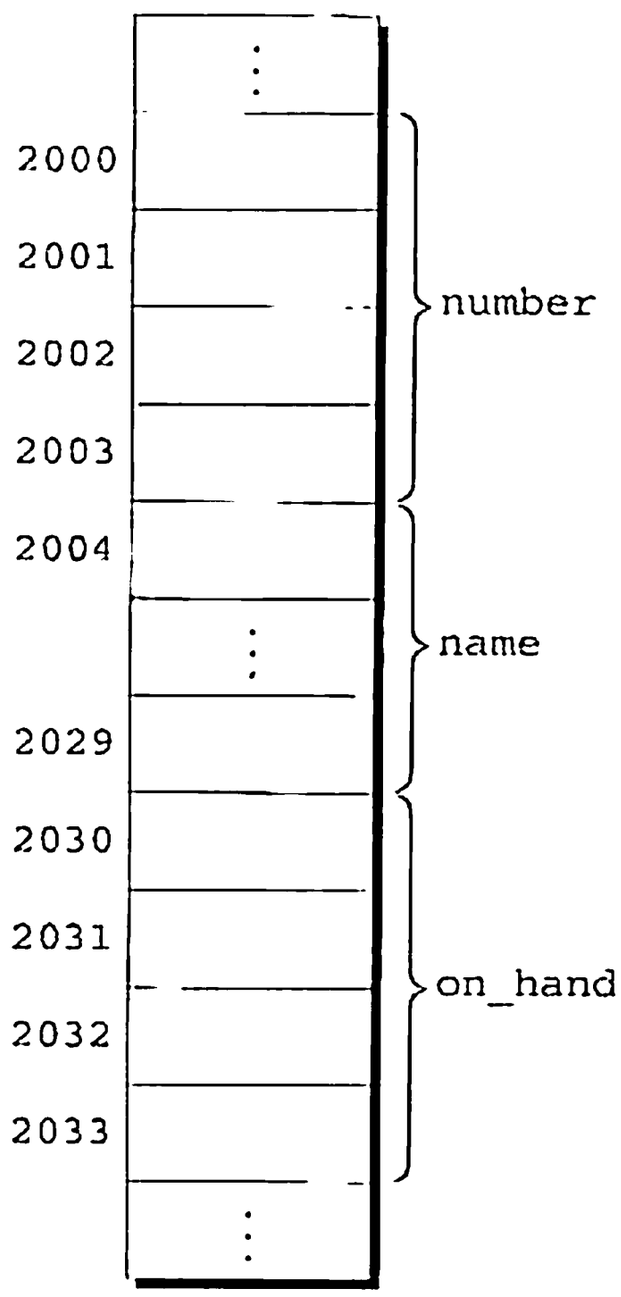
Declaring Structure Variables

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:

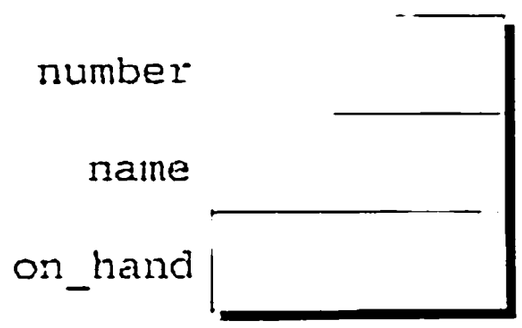
```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;
```

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.

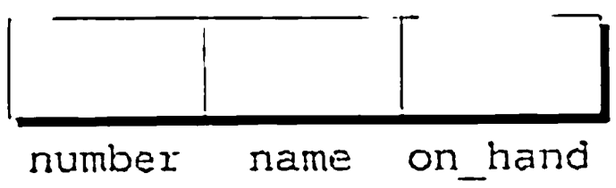
The members of a structure are stored in memory in the order in which they're declared. In order to show what the `part1` variable looks like in memory, let's assume that (1) `part1` is located at address 2000, (2) integers occupy four bytes, (3) `NAME_LEN` has the value 25, and (4) there are no gaps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

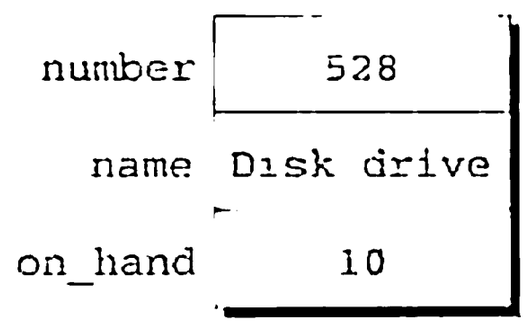
The `number` and `name` members in the `part1` and `part2` structures don't conflict with the `number` and `name` members in `employee1` and `employee2`.

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:



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);
```

lvalues ➤*4.2*

The members of a structure are lvalues, 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 */
```

table of operators ➤*Appendix A*

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:

```
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

type definitions ➤7.5

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.

Q&A

linked lists ➤ 17.5

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);
```

```
  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.

FILE type ➤22.1

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;
  ...
}
```

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.

automatic storage duration ➤ 10.1

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())
```

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`.

`isspace` function ➤23.5

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.

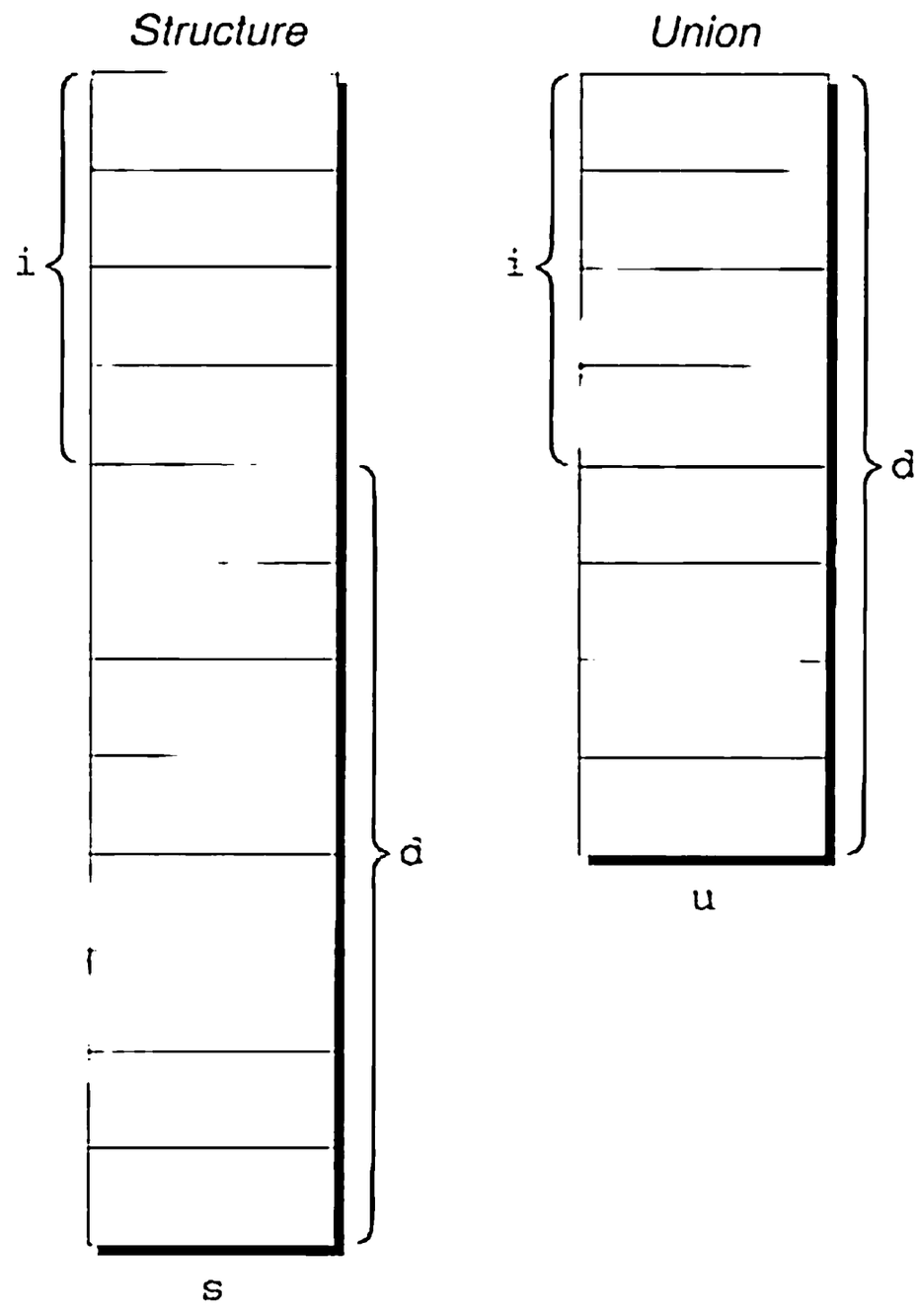
To illustrate the basic properties of unions, let's declare a union variable, u, with two members:

```
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.

protecting header files ➤ 15.2

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
C99 **modify the literal. Is that the case?**

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: C99 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.

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.

C99

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

A: C99 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

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:

(a) `s.shape_kind = RECTANGLE;`
(b) `s.center.x = 10;`
(c) `s.height = 25;`
(d) `s.u.rectangle.width = 8;`
(e) `s.u.circle = 5;`
(f) `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:

(a) Compute the area of s.

(b) 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.)

(c) 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?

(a) An enumeration constant may represent any integer specified by the programmer.

(b) Enumeration constants have exactly the same properties as constants created using #define.

(c) Enumeration constants have the values 0, 1, 2, ... by default.

(d) All constants in an enumeration must have different values.

(e) 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)?

```
(a) b = FALSE;
(b) b = i;
(c) b++;
(d) i = b;
(e) 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

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.

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);
```

`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.

`size_t` type ➤7.6

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);
```

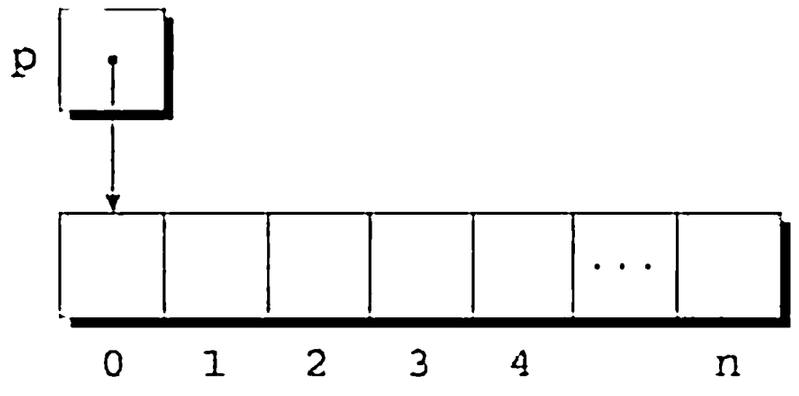
where `p` is a `char *` variable. (The argument is `n + 1` rather than `n` to allow room for the null character.) The generic pointer that `malloc` returns will be converted to `char *` when the assignment is performed; no cast is necessary. (In general, we can assign a `void *` value to a variable of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

Q&A

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

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

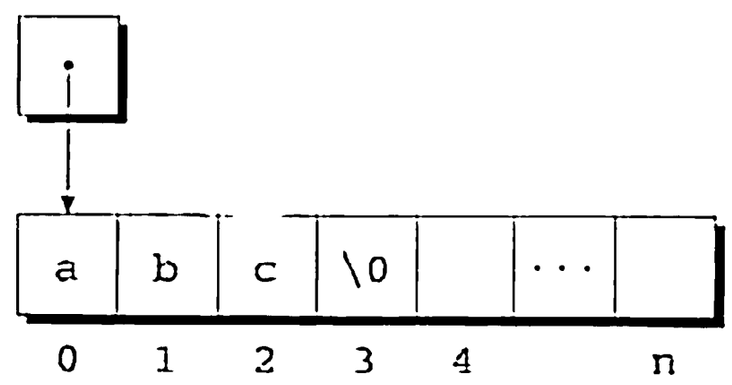
Memory allocated using `malloc` isn't cleared or initialized in any way, so `p` will point to an uninitialized array of `n + 1` characters:



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.

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.

`free` function ➤ 17.4

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

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.

`sizeof` operator ➤7.6

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:

Q&A

```
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.

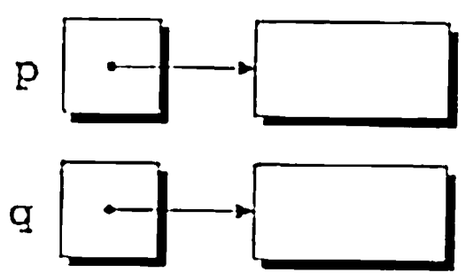
17.4 Deallocating Storage

`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.

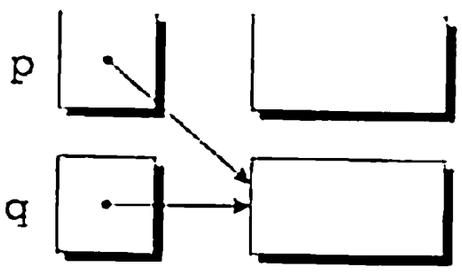
To make matters worse, a program may allocate blocks of memory and then lose track of them, thereby wasting space. Consider the following example:

```
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.

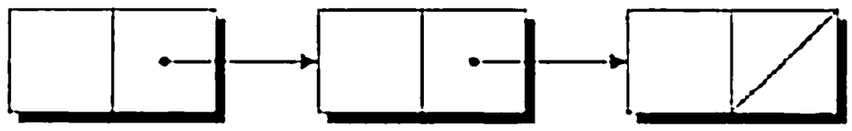
Attempting to access or modify a deallocated memory block causes undefined behavior. Trying to modify a deallocated memory block is likely to have disastrous consequences that may include a program crash.

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`.

Q&A

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;
```