7 Basic Types

Make no mistake about it: Computers process numbers—not symbols. We measure our understanding (and control) by the extent to which we can arithmetize an activity.

So far, we've used only two of C's ***basic*** (built-in) ***types:*** `int` and `float`. (We've also seen `_Bool`, which is a basic type in C99.) This chapter describes the rest of the basic types and discusses important issues about types in general. Section 7.1 reveals the full range of integer types, which include long integers, short integers, and unsigned integers. Section 7.2 introduces the `double` and `long double` types, which provide a larger range of values and greater precision than `float`. Section 7.3 covers the `char` type, which we'll need in order to work with character data. Section 7.4 tackles the thorny topic of converting a value of one type to an equivalent value of another. Section 7.5 shows how to use `typedef` to define new type names. Finally, Section 7.6 describes the `sizeof` operator, which measures the amount of storage required for a type.

7.1 Integer Types

C supports two fundamentally different kinds of numeric types: integer types and floating types. Values of an ***integer type*** are whole numbers, while values of a floating type can have a fractional part as well. The integer types, in turn, are divided into two categories: signed and unsigned.

Signed and Unsigned Integers

The leftmost bit of a ***signed*** integer (known as the ***sign bit***) is 0 if the number is positive or zero, 1 if it's negative. Thus, the largest 16-bit integer has the binary representation

0111111111111111

which has the value 32,767 ($2^{15} - 1$). The largest 32-bit integer is

01111111111111111111111111111111

which has the value 2,147,483,647 ($2^{31} - 1$). An integer with no sign bit (the leftmost bit is considered part of the number's magnitude) is said to be ***unsigned***. The largest 16-bit unsigned integer is 65,535 ($2^{16} - 1$), and the largest 32-bit unsigned integer is 4,294,967,295 ($2^{32} - 1$).

By default, integer variables are signed in C—the leftmost bit is reserved for the sign. To tell the compiler that a variable has no sign bit, we declare it to be `unsigned`. Unsigned numbers are primarily useful for systems programming and low-level, machine-dependent applications. We'll discuss typical applications for unsigned numbers in Chapter 20; until then, we'll generally avoid them.

C's integer types come in different sizes. The `int` type is usually 32 bits, but may be 16 bits on older CPUs. Since some programs require numbers that are too large to store in `int` form, C also provides ***long*** integers. At times, we may need to conserve memory by instructing the compiler to store a number in less space than normal; such a number is called a ***short*** integer.

To construct an integer type that exactly meets our needs, we can specify that a variable is `long` or `short`, `signed` or `unsigned`. We can even combine specifiers (e.g., `long unsigned int`). However, only the following six combinations actually produce different types:

```
short int
unsigned short int

int
unsigned int

long int
unsigned long int
```

Other combinations are synonyms for one of these six types. (For example, `long signed int` is the same as `long int`, since integers are always signed unless otherwise specified.) Incidentally, the order of the specifiers doesn't matter; `unsigned short int` is the same as `short unsigned int`.

C allows us to abbreviate the names of integer types by dropping the word `int`. For example, `unsigned short int` may be abbreviated to `unsigned short`, and `long int` may be abbreviated to just `long`. Omitting `int` is a widespread practice among C programmers, and some newer C-based languages (including Java) actually require the programmer to write `short` or `long` rather than `short int` or `long int`. For these reasons, I'll often omit the word `int` when it's not strictly necessary.

The range of values represented by each of the six integer types varies from one machine to another. However, there are a couple of rules that all compilers must obey. First, the C standard requires that `short int`, `int`, and `long int` each cover a certain minimum range of values (see Section 23.2 for details). Second, the standard requires that `int` not be shorter than `short int`, and `long int` not be shorter than `int`. However, it's possible that `short int` represents the same range of values as `int`; also, `int` may have the same range as `long int`.

Table 7.1 shows the usual range of values for the integer types on a 16-bit machine; note that `short int` and `int` have identical ranges.

Table 7.1 Integer Types on a 16-bit Machine

Type	*Smallest Value*	*Largest Value*
`short int`	–32,768	32,767
`unsigned short int`	0	65,535
`int`	–32,768	32,767
`unsigned int`	0	65,535
`long int`	–2,147,483,648	2,147,483,647
`unsigned long int`	0	4,294,967,295

Table 7.2 shows the usual ranges on a 32-bit machine; here `int` and `long int` have identical ranges.

Table 7.2 Integer Types on a 32-bit Machine

Type	*Smallest Value*	*Largest Value*
`short int`	–32,768	32,767
`unsigned short int`	0	65,535
`int`	–2,147,483,648	2,147,483,647
`unsigned int`	0	4,294,967,295
`long int`	–2,147,483,648	2,147,483,647
`unsigned long int`	0	4,294,967,295

In recent years, 64-bit CPUs have become more common. Table 7.3 shows typical ranges for the integer types on a 64-bit machine (especially under UNIX).

Table 7.3 Integer Types on a 64-bit Machine

Type	*Smallest Value*	*Largest Value*
`short int`	–32,768	32,767
`unsigned short int`	0	65,535
`int`	–2,147,483,648	2,147,483,647
`unsigned int`	0	4,294,967,295
`long int`	–9,223,372,036,854,775,808	9,223,372,036,854,775,807
`unsigned long int`	0	18,446,744,073,709,551,615

Once more, let me emphasize that the ranges shown in Tables 7.1, 7.2, and 7.3 aren't mandated by the C standard and may vary from one compiler to another. One way to determine the ranges of the integer types for a particular implementation is to check the `<limits.h>` header, which is part of the standard library. This header defines macros that represent the smallest and largest values of each integer type.

`<limits.h>` header ➤23.2

C99 Integer Types in C99

C99 provides two additional standard integer types, `long long int` and `unsigned long long int`. These types were added because of the growing need for very large integers and the ability of newer processors to support 64-bit arithmetic. Both `long long` types are required to be at least 64 bits wide, so the range of `long long int` values is typically -2^{63} (–9,223,372,036,854,775,808) to $2^{63} - 1$ (9,223,372,036,854,775,807), and range of `unsigned long long int` values is usually 0 to $2^{64} - 1$ (18,446,744,073,709,551,615).

The `short int`, `int`, `long int`, and `long long int` types (along with the `signed char` type) are called ***standard signed integer types*** in C99. The `unsigned short int`, `unsigned int`, `unsigned long int`, and `unsigned long long int` types (along with the `unsigned char` type and the `_Bool` type) are called ***standard unsigned integer types.***

signed char type ➤7.3
unsigned char type ➤7.3
_Bool type ➤5.2

In addition to the standard integer types, the C99 standard allows implementation-defined ***extended integer types,*** both signed and unsigned. For example, a compiler might provide signed and unsigned 128-bit integer types.

Integer Constants

Let's turn our attention to ***constants***—numbers that appear in the text of a program, not numbers that are read, written, or computed. C allows integer constants to be written in decimal (base 10), octal (base 8), or hexadecimal (base 16).

Octal and Hexadecimal Numbers

An octal number is written using only the digits 0 through 7. Each position in an octal number represents a power of 8 (just as each position in a decimal number represents a power of 10). Thus, the octal number 237 represents the decimal number $2 \times 8^2 + 3 \times 8^1 + 7 \times 8^0 = 128 + 24 + 7 = 159$.

A hexadecimal (or hex) number is written using the digits 0 through 9 plus the letters A through F, which stand for 10 through 15, respectively. Each position in a hex number represents a power of 16; the hex number 1AF has the decimal value $1 \times 16^2 + 10 \times 16^1 + 15 \times 16^0 = 256 + 160 + 15 = 431$.

- ***Decimal*** constants contain digits between `0` and `9`, but must not begin with a zero:

```
15   255   32767
```

- ***Octal*** constants contain only digits between `0` and `7`, and *must* begin with a zero:

```
017   0377   077777
```

- ***Hexadecimal*** constants contain digits between `0` and `9` and letters between `a` and `f`, and always begin with `0x`:

```
0xf   0xff   0x7fff
```

The letters in a hexadecimal constant may be either upper or lower case:

```
0xff   0xfF   0xFf   0xFF   0Xff   0XfF   0XFf   0XFF
```

Keep in mind that octal and hexadecimal are nothing more than an alternative way of writing numbers; they have no effect on how the numbers are actually stored. (Integers are always stored in binary, regardless of what notation we've used to express them.) We can switch from one notation to another at any time, and even mix them: `10 + 015 + 0x20` has the value 55 (decimal). Octal and hex are most convenient for writing low-level programs; we won't use these notations much until Chapter 20.

The type of a *decimal* integer constant is normally `int`. However, if the value of the constant is too large to store as an `int`, the constant has type `long int` instead. In the unlikely case that the constant is too large to store as a `long int`, the compiler will try `unsigned long int` as a last resort. The rules for determining the type of an *octal* or *hexadecimal* constant are slightly different: the compiler will go through the types `int`, `unsigned int`, `long int`, and `unsigned long int` until it finds one capable of representing the constant.

To force the compiler to treat a constant as a long integer, just follow it with the letter `L` (or `l`):

```
15L   0377L   0x7fffL
```

To indicate that a constant is unsigned, put the letter `U` (or `u`) after it:

```
15U   0377U   0x7fffU
```

`L` and `U` may be used in combination to show that a constant is both long *and* unsigned: `0xffffffffUL`. (The order of the `L` and `U` doesn't matter, nor does their case.)

C99 Integer Constants in C99

In C99, integer constants that end with either `LL` or `ll` (the case of the two letters must match) have type `long long int`. Adding the letter `U` (or `u`) before or after the `LL` or `ll` denotes a constant of type `unsigned long long int`.

C99's general rules for determining the type of an integer constant are a bit different from those in C89. The type of a decimal constant with no suffix (`U`, `u`, `L`, `l`, `LL`, or `ll`) is the "smallest" of the types `int`, `long int`, or `long long int` that can represent the value of that constant. For an octal or hexadecimal constant, however, the list of possible types is `int`, `unsigned int`, `long int`, `unsigned long int`, `long long int`, and `unsigned long long int`, in that order. Any suffix at the end of a constant changes the list of possible types. For

example, a constant that ends with `U` (or `u`) must have one of the types `unsigned int`, `unsigned long int`, or `unsigned long long int`. A decimal constant that ends with `L` (or `l`) must have one of the types `long int` or `long long int`. There's also a provision for a constant to have an extended integer type if it's too large to represent using one of the standard integer types.

Integer Overflow

When arithmetic operations are performed on integers, it's possible that the result will be too large to represent. For example, when an arithmetic operation is performed on two `int` values, the result must be able to be represented as an `int`. If the result can't be represented as an `int` (because it requires too many bits), we say that ***overflow*** has occurred.

The behavior when integer overflow occurs depends on whether the operands were signed or unsigned. When overflow occurs during an operation on *signed* integers, the program's behavior is undefined. Recall from Section 4.4 that the consequences of undefined behavior may vary. Most likely the result of the operation will simply be wrong, but the program could crash or exhibit other undesirable behavior.

When overflow occurs during an operation on *unsigned* integers, though, the result *is* defined: we get the correct answer modulo 2^n, where n is the number of bits used to store the result. For example, if we add 1 to the unsigned 16-bit number 65,535, the result is guaranteed to be 0.

Reading and Writing Integers

Suppose that a program isn't working because one of its `int` variables is overflowing. Our first thought is to change the type of the variable from `int` to `long int`. But we're not done yet; we need to see how the change will affect the rest of the program. In particular, we must check whether the variable is used in a call of `printf` or `scanf`. If so, the format string in the call will need to be changed, since the `%d` conversion works only for the `int` type.

Reading and writing unsigned, short, and long integers requires several new conversion specifiers:

Q&A

- When reading or writing an *unsigned* integer, use the letter `u`, `o`, or `x` instead of `d` in the conversion specification. If the `u` specifier is present, the number is read (or written) in decimal notation; `o` indicates octal notation, and `x` indicates hexadecimal notation.

```
unsigned int u;

scanf("%u", &u);    /* reads  u in base 10 */
printf("%u", u);    /* writes u in base 10 */
scanf("%o", &u);    /* reads  u in base  8 */
printf("%o", u);    /* writes u in base  8 */
```

```
scanf("%x", &u);    /* reads  u in base 16 */
printf("%x", u);    /* writes u in base 16 */
```

- When reading or writing a *short* integer, put the letter `h` in front of `d`, `o`, `u`, or `x`:

```
short s;

scanf("%hd", &s);
printf("%hd", s);
```

- When reading or writing a *long* integer, put the letter `l` ("ell," not "one") in front of `d`, `o`, `u`, or `x`:

```
long l;

scanf("%ld", &l);
printf("%ld", l);
```

C99

- When reading or writing a *long long* integer (C99 only), put the letters `ll` in front of `d`, `o`, `u`, or `x`:

```
long long ll;

scanf("%lld", &ll);
printf("%lld", ll);
```

PROGRAM

Summing a Series of Numbers (Revisited)

In Section 6.1, we wrote a program that sums a series of integers entered by the user. One problem with this program is that the sum (or one of the input numbers) might exceed the largest value allowed for an `int` variable. Here's what might happen if the program is run on a machine whose integers are 16 bits long:

```
This program sums a series of integers.
Enter integers (0 to terminate): 10000 20000 30000 0
The sum is: -5536
```

The sum was 60,000, which wouldn't fit in an `int` variable, so overflow occurred. When overflow occurs with signed numbers, the outcome is undefined. In this case, we got an apparently meaningless number. To improve the program, let's switch to `long` variables.

sum2.c

```
/* Sums a series of numbers (using long variables) */

#include <stdio.h>

int main(void)
{
  long n, sum = 0;

  printf("This program sums a series of integers.\n");
```

```
    printf("Enter integers (0 to terminate): ");

    scanf("%ld", &n);
    while (n != 0) {
      sum += n;
      scanf("%ld", &n);
    }
    printf("The sum is: %ld\n", sum);

    return 0;
}
```

The change was fairly simple: we declared `n` and `sum` to be `long` variables instead of `int` variables, then we changed the conversion specifications in `scanf` and `printf` to `%ld` instead of `%d`.

7.2 Floating Types

The integer types aren't suitable for all applications. Sometimes we'll need variables that can store numbers with digits after the decimal point, or numbers that are exceedingly large or small. Numbers like these are stored in floating-point format (so called because the decimal point "floats"). C provides three ***floating types***, corresponding to different floating-point formats:

`float`	Single-precision floating-point
`double`	Double-precision floating-point
`long double`	Extended-precision floating-point

`float` is suitable when the amount of precision isn't critical (calculating temperatures to one decimal point, for example). `double` provides greater precision—enough for most programs. `long double`, which supplies the ultimate in precision, is rarely used.

The C standard doesn't state how much precision the `float`, `double`, and `long double` types provide, since different computers may store floating-point numbers in different ways. Most modern computers follow the specifications in IEEE Standard 754 (also known as IEC 60559), so we'll use it as an example.

The IEEE Floating-Point Standard

IEEE Standard 754, developed by the Institute of Electrical and Electronics Engineers, provides two primary formats for floating-point numbers: single precision (32 bits) and double precision (64 bits). Numbers are stored in a form of scientific notation, with each number having three parts: a ***sign***, an ***exponent***, and a ***fraction***. The number of bits reserved for the exponent determines how large (or small) numbers can be, while the number of bits in the fraction determines the precision. In single-precision format, the exponent is 8 bits long, while the fraction occupies 23

bits. As a result, a single-precision number has a maximum value of approximately 3.40×10^{38}, with a precision of about 6 decimal digits.

The IEEE standard also describes two other formats, single extended precision and double extended precision. The standard doesn't specify the number of bits in these formats, although it requires that the single extended type occupy at least 43 bits and the double extended type at least 79 bits. For more information about the IEEE standard and floating-point arithmetic in general, see "What every computer scientist should know about floating-point arithmetic" by David Goldberg (*ACM Computing Surveys*, vol. 23, no. 1 (March 1991): 5–48).

Table 7.4 shows the characteristics of the floating types when implemented according to the IEEE standard. (The table shows the smallest positive *normalized* values. Subnormal numbers can be smaller.) The `long double` type isn't shown in the table, since its length varies from one machine to another, with 80 bits and 128 bits being the most common sizes.

subnormal numbers ➤23.4

Table 7.4 Floating Type Characteristics (IEEE Standard)

Type	*Smallest Positive Value*	*Largest Value*	*Precision*
`float`	1.17549×10^{-38}	3.40282×10^{38}	6 digits
`double`	2.22507×10^{-308}	1.79769×10^{308}	15 digits

On computers that don't follow the IEEE standard, Table 7.4 won't be valid. In fact, on some machines, `float` may have the same set of values as `double`, or `double` may have the same values as `long double`. Macros that define the characteristics of the floating types can be found in the `<float.h>` header.

<float.h> header ➤23.1

C99

In C99, the floating types are divided into two categories. The `float`, `double`, and `long double` types fall into one category, called the ***real floating types***. Floating types also include the ***complex types*** (`float _Complex`, `double _Complex`, and `long double _Complex`), which are new in C99.

complex types ➤27.3

Floating Constants

Floating constants can be written in a variety of ways. The following constants, for example, are all valid ways of writing the number 57.0:

```
57.0  57.  57.0e0  57E0  5.7e1  5.7e+1  .57e2  570.e-1
```

A floating constant must contain a decimal point and/or an exponent; the exponent indicates the power of 10 by which the number is to be scaled. If an exponent is present, it must be preceded by the letter `E` (or `e`). An optional `+` or `-` sign may appear after the `E` (or `e`).

By default, floating constants are stored as double-precision numbers. In other words, when a C compiler finds the constant 57.0 in a program, it arranges for the number to be stored in memory in the same format as a `double` variable. This rule generally causes no problems, since `double` values are converted automatically to `float` when necessary.

Q&A

On occasion, it may be necessary to force the compiler to store a floating constant in `float` or `long double` format. To indicate that only single precision is desired, put the letter `F` (or `f`) at the end of the constant (for example, `57.0F`). To indicate that a constant should be stored in `long double` format, put the letter `L` (or `l`) at the end (`57.0L`).

C99 C99 has a provision for writing floating constants in hexadecimal. Such a constant begins with `0x` or `0X` (like a hexadecimal integer constant). This feature is rarely used.

Q&A

Reading and Writing Floating-Point Numbers

As we've discussed, the conversion specifications `%e`, `%f`, and `%g` are used for reading and writing single-precision floating-point numbers. Values of types `double` and `long double` require slightly different conversions:

- When *reading* a value of type `double`, put the letter `l` in front of `e`, `f`, or `g`:

```
double d;

scanf("%lf", &d);
```

Q&A *Note:* Use `l` only in a `scanf` format string, not a `printf` string. In a `printf` format string, the `e`, `f`, and `g` conversions can be used to write either

C99 `float` or `double` values. (C99 legalizes the use of `%le`, `%lf`, and `%lg` in calls of `printf`, although the `l` has no effect.)

- When reading or writing a value of type `long double`, put the letter `L` in front of `e`, `f`, or `g`:

```
long double ld;

scanf("%Lf", &ld);
printf("%Lf", ld);
```

7.3 Character Types

Q&A The only remaining basic type is `char`, the character type. The values of type `char` can vary from one computer to another, because different machines may have different underlying character sets.

Character Sets

ASCII character set ➤ *Appendix E*

Today's most popular character set is ***ASCII*** (American Standard Code for Information Interchange), a 7-bit code capable of representing 128 characters. In ASCII, the digits 0 to 9 are represented by the codes 0110000–0111001, and the uppercase letters A to Z are represented by 1000001–1011010. ASCII is often extended

to a 256-character code known as ***Latin-1*** that provides the characters necessary for Western European and many African languages.

A variable of type `char` can be assigned any single character:

```
char ch;

ch = 'a';   /* lower-case a */
ch = 'A';   /* upper-case A */
ch = '0';   /* zero         */
ch = ' ';   /* space        */
```

Notice that character constants are enclosed in single quotes, not double quotes.

Operations on Characters

Working with characters in C is simple, because of one fact: *C treats characters as small integers*. After all, characters are encoded in binary, and it doesn't take much imagination to view these binary codes as integers. In ASCII, for example, character codes range from 0000000 to 1111111, which we can think of as the integers from 0 to 127. The character `'a'` has the value 97, `'A'` has the value 65, `'0'` has the value 48, and `' '` has the value 32. The connection between characters and integers in C is so strong that character constants actually have `int` type rather than `char` type (an interesting fact, but not one that will often matter to us).

When a character appears in a computation, C simply uses its integer value. Consider the following examples, which assume the ASCII character set:

```
char ch;
int i;

i = 'a';        /* i is now 97   */
ch = 65;        /* ch is now 'A' */
ch = ch + 1;    /* ch is now 'B' */
ch++;           /* ch is now 'C' */
```

Characters can be compared, just as numbers can. The following `if` statement checks whether `ch` contains a lower-case letter; if so, it converts `ch` to upper case.

```
if ('a' <= ch && ch <= 'z')
  ch = ch - 'a' + 'A';
```

Comparisons such as `'a' <= ch` are done using the integer values of the characters involved. These values depend on the character set in use, so programs that use `<`, `<=`, `>`, and `>=` to compare characters may not be portable.

The fact that characters have the same properties as numbers has some advantages. For example, we can easily write a `for` statement whose control variable steps through all the upper-case letters:

```
for (ch = 'A'; ch <= 'Z'; ch++) …
```

On the other hand, treating characters as numbers can lead to various programming errors that won't be caught by the compiler, and lets us write meaningless expressions such as `'a' * 'b' / 'c'`. It can also hamper portability, since our programs may be based on assumptions about the underlying character set. (Our `for` loop, for example, assumes that the letters from A to Z have consecutive codes.)

Signed and Unsigned Characters

Since C allows characters to be used as integers, it shouldn't be surprising that the `char` type—like the integer types—exists in both signed and unsigned versions. Signed characters normally have values between –128 and 127, while unsigned characters have values between 0 and 255.

The C standard doesn't specify whether ordinary `char` is a signed or an unsigned type; some compilers treat it as a signed type, while others treat it as an unsigned type. (Some even allow the programmer to select, via a compiler option, whether `char` should be signed or unsigned.)

Most of the time, we don't really care whether `char` is signed or unsigned. Once in a while, though, we do, especially if we're using a character variable to store a small integer. For this reason, C allows the use of the words `signed` and `unsigned` to modify `char`:

Q&A

```
signed char sch;
unsigned char uch;
```

portability tip

Don't assume that `char` *is either signed or unsigned by default. If it matters, use* `signed char` *or* `unsigned char` *instead of* `char`.

In light of the close relationship between characters and integers, C89 uses the term ***integral types*** to refer to both the integer types and the character types. Enumerated types are also integral types.

enumerated types ➤16.5

C99

C99 doesn't use the term "integral types." Instead, it expands the meaning of "integer types" to include the character types and the enumerated types. C99's `_Bool` type is considered to be an unsigned integer type.

_Bool type ➤5.2

Arithmetic Types

The integer types and floating types are collectively known as ***arithmetic types.*** Here's a summary of the arithmetic types in C89, divided into categories and subcategories:

- Integral types
  - `char`
  - Signed integer types (`signed char`, `short int`, `int`, `long int`)
  - Unsigned integer types (`unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int`)

- Enumerated types
- Floating types (`float`, `double`, `long double`)

C99 C99 has a more complicated hierarchy for its arithmetic types:

- Integer types
  - `char`
  - Signed integer types, both standard (`signed char`, `short int`, `int`, `long int`, `long long int`) and extended
  - Unsigned integer types, both standard (`unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int`, `unsigned long long int`, `_Bool`) and extended
  - Enumerated types
- Floating types
  - Real floating types (`float`, `double`, `long double`)
  - Complex types (`float _Complex`, `double _Complex`, `long double _Complex`)

Escape Sequences

A character constant is usually one character enclosed in single quotes, as we've seen in previous examples. However, certain special characters—including the new-line character—can't be written in this way, because they're invisible (nonprinting) or because they can't be entered from the keyboard. So that programs can deal with every character in the underlying character set, C provides a special notation, the ***escape sequence***.

There are two kinds of escape sequences: ***character escapes*** and ***numeric escapes***. We saw a partial list of character escapes in Section 3.1; Table 7.5 gives the complete set.

Table 7.5
Character Escapes

Name	*Escape Sequence*
Alert (bell)	`\a`
Backspace	`\b`
Form feed	`\f`
New line	`\n`
Carriage return	`\r`
Horizontal tab	`\t`
Vertical tab	`\v`
Backslash	`\\`
Question mark	`\?`
Single quote	`\'`
Double quote	`\"`

Q&A The `\a`, `\b`, `\f`, `\r`, `\t`, and `\v` escapes represent common ASCII control characters. The `\n` escape represents the ASCII line-feed character. The `\\` escape allows a character constant or string to contain the `\` character. The `\'` escape

allows a character constant to contain the ' character, while the \" escape allows a string to contain the " character. The \? escape is rarely used.

Q&A

Character escapes are handy, but they have a problem: the list of character escapes doesn't include all the nonprinting ASCII characters, just the most common. Character escapes are also useless for representing characters beyond the basic 128 ASCII characters. Numeric escapes, which can represent *any* character, are the solution to this problem.

To write a numeric escape for a particular character, first look up the character's octal or hexadecimal value in a table like the one in Appendix E. For example, the ASCII escape character (decimal value: 27) has the value 33 in octal and 1B in hex. Either of these codes can be used to write an escape sequence:

- An ***octal escape sequence*** consists of the \ character followed by an octal number with at most three digits. (This number must be representable as an unsigned character, so its maximum value is normally 377 octal.) For example, the escape character could be written `\33` or `\033`. Octal numbers in escape sequences—unlike octal constants—don't have to begin with `0`.
- A ***hexadecimal escape sequence*** consists of `\x` followed by a hexadecimal number. Although C places no limit on the number of digits in the hexadecimal number, it must be representable as an unsigned character (hence it can't exceed FF if characters are eight bits long). Using this notation, the escape character would be written `\x1b` or `\x1B`. The `x` must be in lower case, but the hex digits (such as `b`) can be upper or lower case.

When used as a character constant, an escape sequence must be enclosed in single quotes. For example, a constant representing the escape character would be written `'\33'` (or `'\x1b'`). Escape sequences tend to get a bit cryptic, so it's often a good idea to give them names using `#define`:

```
#define ESC '\33'   /* ASCII escape character */
```

Escape sequences can be embedded in strings as well, as we saw in Section 3.1.

Escape sequences aren't the only special notations for representing characters. Trigraph sequences provide a way to represent the characters #, [, \,], ^, {, |, }, and ~, which may not be available on keyboards in some countries. C99 adds universal character names, which resemble escape sequences. Unlike escape sequences, however, universal character names are allowed in identifiers.

trigraph sequences ➤25.3

C99

universal character names ➤25.4

Character-Handling Functions

Earlier in this section, we saw how to write an `if` statement that converts a lower-case letter to upper-case:

```
if ('a' <= ch && ch <= 'z')
  ch = ch - 'a' + 'A';
```

This isn't the best method, though. A faster—and more portable—way to convert case is to call C's `toupper` library function:

```
ch = toupper(ch);   /* converts ch to upper case */
```

When it's called, `toupper` checks whether its argument (`ch` in this case) is a lower-case letter. If so, it returns the corresponding upper-case letter. Otherwise, `toupper` returns the value of the argument. In our example, we've used the assignment operator to store the return value of `toupper` back into the `ch` variable, although we could just as easily have done something else with it—stored it in another variable, say, or tested it in an `if` statement:

```
if (toupper(ch) == 'A') …
```

Programs that call `toupper` need to have the following `#include` directive at the top:

```
#include <ctype.h>
```

`toupper` isn't the only useful character-handling function in the C library. Section 23.5 describes them all and gives examples of their use.

Reading and Writing Characters using `scanf` and `printf`

The `%c` conversion specification allows `scanf` and `printf` to read and write single characters:

```
char ch;

scanf("%c", &ch);   /* reads a single character */
printf("%c", ch);   /* writes a single character */
```

`scanf` doesn't skip white-space characters before reading a character. If the next unread character is a space, then the variable `ch` in the previous example will contain a space after `scanf` returns. To force `scanf` to skip white space before reading a character, put a space in its format string just before `%c`:

```
scanf(" %c", &ch);   /* skips white space, then reads ch */
```

Recall from Section 3.2 that a blank in a `scanf` format string means "skip zero or more white-space characters."

Since `scanf` doesn't normally skip white space, it's easy to detect the end of an input line: check to see if the character just read is the new-line character. For example, the following loop will read and ignore all remaining characters in the current input line:

```
do {
  scanf("%c", &ch);
} while (ch != '\n');
```

When `scanf` is called the next time, it will read the first character on the next input line.

Reading and Writing Characters using `getchar` and `putchar`

Q&A

C provides other ways to read and write single characters. In particular, we can use the `getchar` and `putchar` functions instead of calling `scanf` and `printf`. `putchar` writes a single character:

```
putchar(ch);
```

Each time `getchar` is called, it reads one character, which it returns. In order to save this character, we must use assignment to store it in a variable:

```
ch = getchar();  /* reads a character and stores it in ch */
```

`getchar` actually returns an `int` value rather than a `char` value (the reason will be discussed in later chapters). As a result, it's not unusual for a variable to have type `int` rather than `char` if it will be used to store a character read by `getchar`. Like `scanf`, `getchar` doesn't skip white-space characters as it reads.

Using `getchar` and `putchar` (rather than `scanf` and `printf`) saves time when the program is executed. `getchar` and `putchar` are fast for two reasons. First, they're much simpler than `scanf` and `printf`, which are designed to read and write many kinds of data in a variety of formats. Second, `getchar` and `putchar` are usually implemented as macros for additional speed.

macros ➤ 14.3

`getchar` has another advantage over `scanf`: because it returns the character that it reads, `getchar` lends itself to various C idioms, including loops that search for a character or skip over all occurrences of a character. Consider the `scanf` loop that we used to skip the rest of an input line:

```
do {
  scanf("%c", &ch);
} while (ch != '\n');
```

Rewriting this loop using `getchar` gives us the following:

```
do {
  ch = getchar();
} while (ch != '\n');
```

Moving the call of `getchar` into the controlling expression allows us to condense the loop:

```
while ((ch = getchar()) != '\n')
  ;
```

This loop reads a character, stores it into the variable `ch`, then tests if `ch` is not equal to the new-line character. If the test succeeds, the loop body (which is empty) is executed, then the loop test is performed once more, causing a new character to be read. Actually, we don't even need the `ch` variable; we can just compare the return value of `getchar` with the new-line character:

idiom
```
while (getchar() != '\n')    /* skips rest of line */
  ;
```

The resulting loop is a well-known C idiom that's cryptic but worth learning.

`getchar` is useful in loops that skip characters as well as loops that search for characters. Consider the following statement, which uses `getchar` to skip an indefinite number of blank characters:

idiom
```
while ((ch = getchar()) == ' ')    /* skips blanks */
  ;
```

When the loop terminates, `ch` will contain the first nonblank character that `getchar` encountered.

Be careful if you mix `getchar` and `scanf` in the same program. `scanf` has a tendency to leave behind characters that it has "peeked" at but not read, including the new-line character. Consider what happens if we try to read a number first, then a character:

```
printf("Enter an integer: ");
scanf("%d", &i);
printf("Enter a command: ");
command = getchar();
```

The call of `scanf` will leave behind any characters that weren't consumed during the reading of `i`, including (but not limited to) the new-line character. `getchar` will fetch the first leftover character, which wasn't what we had in mind.

PROGRAM

Determining the Length of a Message

To illustrate how characters are read, let's write a program that calculates the length of a message. After the user enters the message, the program displays the length:

```
Enter a message: Brevity is the soul of wit.
Your message was 27 character(s) long.
```

The length includes spaces and punctuation, but not the new-line character at the end of the message.

We'll need a loop whose body reads a character and increments a counter. The loop will terminate as soon as a new-line character turns up. We could use either `scanf` or `getchar` to read characters; most C programmers would choose `getchar`. Using a straightforward `while` loop, we might end up with the following program.

length.c

```
/* Determines the length of a message */

#include <stdio.h>

int main(void)
{
  char ch;
  int len = 0;

  printf("Enter a message: ");
  ch = getchar();
  while (ch != '\n') {
    len++;
    ch = getchar();
  }
  printf("Your message was %d character(s) long.\n", len);

  return 0;
}
```

Recalling our discussion of idioms involving `while` loops and `getchar`, we realize that the program can be shortened:

length2.c

```
/* Determines the length of a message */

#include <stdio.h>

int main(void)
{
  int len = 0;

  printf("Enter a message: ");
  while (getchar() != '\n')
    len++;
  printf("Your message was %d character(s) long.\n", len);

  return 0;
}
```

7.4 Type Conversion

Computers tend to be more restrictive than C when it comes to arithmetic. For a computer to perform an arithmetic operation, the operands must usually be of the same size (the same number of bits) and be stored in the same way. A computer may be able to add two 16-bit integers directly, but not a 16-bit integer and a 32-bit integer or a 32-bit integer and a 32-bit floating-point number.

C, on the other hand, allows the basic types to be mixed in expressions. We can combine integers, floating-point numbers, and even characters in a single expression. The C compiler may then have to generate instructions that convert

some operands to different types so that the hardware will be able to evaluate the expression. If we add a 16-bit `short` and a 32-bit `int`, for example, the compiler will arrange for the `short` value to be converted to 32 bits. If we add an `int` and a `float`, the compiler will arrange for the `int` to be converted to `float` format. This conversion is a little more complicated, since `int` and `float` values are stored in different ways.

Because the compiler handles these conversions automatically, without the programmer's involvement, they're known as ***implicit conversions***. C also allows the programmer to perform ***explicit conversions***, using the cast operator. I'll discuss implicit conversions first, postponing explicit conversions until later in the section. Unfortunately, the rules for performing implicit conversions are somewhat complex, primarily because C has so many different arithmetic types.

Implicit conversions are performed in the following situations:

- When the operands in an arithmetic or logical expression don't have the same type. (C performs what are known as the ***usual arithmetic conversions***.)
- When the type of the expression on the right side of an assignment doesn't match the type of the variable on the left side.
- When the type of an argument in a function call doesn't match the type of the corresponding parameter.
- When the type of the expression in a `return` statement doesn't match the function's return type.

We'll discuss the first two cases now and save the others for Chapter 9.

The Usual Arithmetic Conversions

The usual arithmetic conversions are applied to the operands of most binary operators, including the arithmetic, relational, and equality operators. For example, let's say that `f` has type `float` and `i` has type `int`. The usual arithmetic conversions will be applied to the operands in the expression `f + i`, because their types aren't the same. Clearly it's safer to convert `i` to type `float` (matching `f`'s type) rather than convert `f` to type `int` (matching `i`'s type). An integer can always be converted to `float`; the worst that can happen is a minor loss of precision. Converting a floating-point number to `int`, on the other hand, would cost us the fractional part of the number. Worse still, we'd get a completely meaningless result if the original number were larger than the largest possible integer or smaller than the smallest integer.

The strategy behind the usual arithmetic conversions is to convert operands to the "narrowest" type that will safely accommodate both values. (Roughly speaking, one type is narrower than another if it requires fewer bytes to store.) The types of the operands can often be made to match by converting the operand of the narrower type to the type of the other operand (this act is known as ***promotion***). Among the most common promotions are the ***integral promotions***, which convert a character or short integer to type `int` (or to `unsigned int` in some cases).

Q&A

We can divide the rules for performing the usual arithmetic conversions into two cases:

- ***The type of either operand is a floating type.*** Use the following diagram to promote the operand whose type is narrower:

```
long double
     ↑
  double
     ↑
  float
```

  That is, if one operand has type `long double`, then convert the other operand to type `long double`. Otherwise, if one operand has type `double`, convert the other operand to type `double`. Otherwise, if one operand has type `float`, convert the other operand to type `float`. Note that these rules cover mixtures of integer and floating types: if one operand has type `long int`, for example, and the other has type `double`, the `long int` operand is converted to `double`.

- ***Neither operand type is a floating type.*** First perform integral promotion on both operands (guaranteeing that neither operand will be a character or short integer). Then use the following diagram to promote the operand whose type is narrower:

```
unsigned long int
        ↑
    long int
        ↑
  unsigned int
        ↑
     int
```

  There's one special case, but it occurs only when `long int` and `unsigned int` have the same length (32 bits, say). Under these circumstances, if one operand has type `long int` and the other has type `unsigned int`, both are converted to `unsigned long int`.

When a signed operand is combined with an unsigned operand, the signed operand is converted to an unsigned value. The conversion involves adding or subtracting a multiple of $n + 1$, where n is the largest representable value of the unsigned type. This rule can cause obscure programming errors.

Suppose that the `int` variable `i` has the value –10 and the `unsigned int` variable `u` has the value 10. If we compare `i` and `u` using the < operator, we might expect to get the result 1 (true). Before the comparison, however, `i` is converted to `unsigned int`. Since a negative number can't be represented as an unsigned integer, the converted value won't be –10. Instead, the value 4,294,967,296 is added (assuming that 4,294,967,295 is the largest `unsigned int` value), giving

a converted value of 4,294,967,286. The comparison `i < u` will therefore produce 0. Some compilers produce a warning message such as "*comparison between signed and unsigned*" when a program attempts to compare a signed number with an unsigned number.

Because of traps like this one, it's best to use unsigned integers as little as possible and, especially, never mix them with signed integers.

The following example shows the usual arithmetic conversions in action:

```
char c;
short int s;
int i;
unsigned int u;
long int l;
unsigned long int ul;
float f;
double d;
long double ld;

i = i + c;     /* c is converted to int                    */
i = i + s;     /* s is converted to int                    */
u = u + i;     /* i is converted to unsigned int           */
l = l + u;     /* u is converted to long int               */
ul = ul + l;   /* l is converted to unsigned long int      */
f = f + ul;    /* ul is converted to float                 */
d = d + f;     /* f is converted to double                 */
ld = ld + d;   /* d is converted to long double            */
```

Conversion During Assignment

The usual arithmetic conversions don't apply to assignment. Instead, C follows the simple rule that the expression on the right side of the assignment is converted to the type of the variable on the left side. If the variable's type is at least as "wide" as the expression's, this will work without a snag. For example:

```
char c;
int i;
float f;
double d;

i = c;   /* c is converted to int   */
f = i;   /* i is converted to float */
d = f;   /* f is converted to double */
```

Other cases are problematic. Assigning a floating-point number to an integer variable drops the fractional part of the number:

```
int i;

i = 842.97;     /* i is now 842 */
i = -842.97;    /* i is now -842 */
```

Q&A

Moreover, assigning a value to a variable of a narrower type will give a meaningless result (or worse) if the value is outside the range of the variable's type:

```
c = 10000;     /*** WRONG ***/
i = 1.0e20;    /*** WRONG ***/
f = 1.0e100;   /*** WRONG ***/
```

A "narrowing" assignment may elicit a warning from the compiler or from tools such as `lint`.

It's a good idea to append the `f` suffix to a floating-point constant if it will be assigned to a `float` variable, as we've been doing since Chapter 2:

```
f = 3.14159f;
```

Without the suffix, the constant `3.14159` would have type `double`, possibly causing a warning message.

C99 Implicit Conversions in C99

_Bool type ➤5.2

The rules for implicit conversions in C99 are somewhat different from the rules in C89, primarily because C99 has additional types (`_Bool`, `long long` types, extended integer types, and complex types).

For the purpose of defining conversion rules, C99 gives each integer type an "integer conversion rank." Here are the ranks from highest to lowest:

1. `long long int`, `unsigned long long int`
2. `long int`, `unsigned long int`
3. `int`, `unsigned int`
4. `short int`, `unsigned short int`
5. `char`, `signed char`, `unsigned char`
6. `_Bool`

For simplicity, I'm ignoring extended integer types and enumerated types.

In place of C89's integral promotions, C99 has "integer promotions," which involve converting any type whose rank is less than `int` and `unsigned int` to `int` (provided that all values of the type can be represented using `int`) or else to `unsigned int`.

As in C89, the C99 rules for performing the usual arithmetic conversions can be divided into two cases:

- ***The type of either operand is a floating type.*** As long as neither operand has a complex type, the rules are the same as before. (The conversion rules for complex types will be discussed in Section 27.3.)
- ***Neither operand type is a floating type.*** First perform integer promotion on both operands. If the types of the two operands are now the same, the process ends. Otherwise, use the following rules, stopping at the first one that applies:
  - If both operands have signed types or both have unsigned types, convert the

operand whose type has lesser integer conversion rank to the type of the operand with greater rank.

- If the unsigned operand has rank greater or equal to the rank of the type of the signed operand, convert the signed operand to the type of the unsigned operand.
- If the type of the signed operand can represent all of the values of the type of the unsigned operand, convert the unsigned operand to the type of the signed operand.
- Otherwise, convert both operands to the unsigned type corresponding to the type of the signed operand.

Incidentally, all arithmetic types can be converted to `_Bool` type. The result of the conversion is 0 if the original value is 0; otherwise, the result is 1.

Casting

Although C's implicit conversions are convenient, we sometimes need a greater degree of control over type conversion. For this reason, C provides ***casts***. A cast expression has the form

cast expression

(*type-name*) *expression*

type-name specifies the type to which the expression should be converted.

The following example shows how to use a cast expression to compute the fractional part of a `float` value:

```
float f, frac_part;

frac_part = f - (int) f;
```

The cast expression `(int) f` represents the result of converting the value of `f` to type `int`. C's usual arithmetic conversions then require that `(int) f` be converted back to type `float` before the subtraction can be performed. The difference between `f` and `(int) f` is the fractional part of `f`, which was dropped during the cast.

Cast expressions enable us to document type conversions that would take place anyway:

```
i = (int) f;    /* f is converted to int */
```

They also enable us to overrule the compiler and force it to do conversions that we want. Consider the following example:

```
float quotient;
int dividend, divisor;

quotient = dividend / divisor;
```

As it's now written, the result of the division—an integer—will be converted to `float` form before being stored in `quotient`. We probably want `dividend` and `divisor` converted to `float` *before* the division, though, so that we get a more exact answer. A cast expression will do the trick:

```
quotient = (float) dividend / divisor;
```

`divisor` doesn't need a cast, since casting `dividend` to `float` forces the compiler to convert `divisor` to `float` also.

Incidentally, C regards (*type-name*) as a unary operator. Unary operators have higher precedence than binary operators, so the compiler interprets

```
(float) dividend / divisor
```

as

```
((float) dividend) / divisor
```

If you find this confusing, note that there are other ways to accomplish the same effect:

```
quotient = dividend / (float) divisor;
```

or

```
quotient = (float) dividend / (float) divisor;
```

Casts are sometimes necessary to avoid overflow. Consider the following example:

```
long i;
int j = 1000;

i = j * j;    /* overflow may occur */
```

At first glance, this statement looks fine. The value of `j * j` is 1,000,000, and `i` is a `long`, so it can easily store values of this size, right? The problem is that when two `int` values are multiplied, the result will have `int` type. But `j * j` is too large to represent as an `int` on some machines, causing an overflow. Fortunately, using a cast avoids the problem:

```
i = (long) j * j;
```

Since the cast operator takes precedence over `*`, the first `j` is converted to `long` type, forcing the second `j` to be converted as well. Note that the statement

```
i = (long) (j * j);    /*** WRONG ***/
```

wouldn't work, since the overflow would already have occurred by the time of the cast.

7.5 Type Definitions

In Section 5.2, we used the `#define` directive to create a macro that could be used as a Boolean type:

```
#define BOOL int
```

Q&A There's a better way to set up a Boolean type, though, using a feature known as a ***type definition:***

```
typedef int Bool;
```

Notice that the name of the type being defined comes *last*. Note also that I've capitalized the word `Bool`. Capitalizing the first letter of a type name isn't required; it's just a convention that some C programmers employ.

Using `typedef` to define `Bool` causes the compiler to add `Bool` to the list of type names that it recognizes. `Bool` can now be used in the same way as the built-in type names—in variable declarations, cast expressions, and elsewhere. For example, we might use `Bool` to declare variables:

```
Bool flag;   /* same as int flag; */
```

The compiler treats `Bool` as a synonym for `int`; thus, `flag` is really nothing more than an ordinary `int` variable.

Advantages of Type Definitions

Type definitions can make a program more understandable (assuming that the programmer has been careful to choose meaningful type names). For example, suppose that the variables `cash_in` and `cash_out` will be used to store dollar amounts. Declaring `Dollars` as

```
typedef float Dollars;
```

and then writing

```
Dollars cash_in, cash_out;
```

is more informative than just writing

```
float cash_in, cash_out;
```

Type definitions can also make a program easier to modify. If we later decide that `Dollars` should really be defined as `double`, all we need do is change the type definition:

```
typedef double Dollars;
```

The declarations of `Dollars` variables need not be changed. Without the type definition, we would need to locate all `float` variables that store dollar amounts (not necessarily an easy task) and change their declarations.

Type Definitions and Portability

Type definitions are an important tool for writing portable programs. One of the problems with moving a program from one computer to another is that types may have different ranges on different machines. If `i` is an `int` variable, an assignment like

```
i = 100000;
```

is fine on a machine with 32-bit integers, but will fail on a machine with 16-bit integers.

portability tip

For greater portability, consider using `typedef` *to define new names for integer types.*

Suppose that we're writing a program that needs variables capable of storing product quantities in the range 0–50,000. We could use `long` variables for this purpose (since they're guaranteed to be able to hold numbers up to at least 2,147,483,647), but we'd rather use `int` variables, since arithmetic on `int` values may be faster than operations on `long` values; also, `int` variables may take up less space.

Instead of using the `int` type to declare quantity variables, we can define our own "quantity" type:

```
typedef int Quantity;
```

and use this type to declare variables:

```
Quantity q;
```

When we transport the program to a machine with shorter integers, we'll change the definition of `Quantity`:

```
typedef long Quantity;
```

This technique doesn't solve all our problems, unfortunately, since changing the definition of `Quantity` may affect the way `Quantity` variables are used. At the very least, calls of `printf` and `scanf` that use `Quantity` variables will need to be changed, with `%d` conversion specifications replaced by `%ld`.

The C library itself uses `typedef` to create names for types that can vary from one C implementation to another; these types often have names that end with `_t`, such as `ptrdiff_t`, `size_t`, and `wchar_t`. The exact definitions of these types will vary, but here are some typical examples:

```
typedef long int ptrdiff_t;
typedef unsigned long int size_t;
typedef int wchar_t;
```

C99

<stdint.h> header ➤27.1

In C99, the `<stdint.h>` header uses `typedef` to define names for integer types with a particular number of bits. For example, `int32_t` is a signed integer type with exactly 32 bits. Using these types is an effective way to make programs more portable.

7.6 The `sizeof` Operator

The `sizeof` operator allows a program to determine how much memory is required to store values of a particular type. The value of the expression

`sizeof` expression

```
sizeof ( type-name )
```

is an unsigned integer representing the number of bytes required to store a value belonging to *type-name*. `sizeof(char)` is always 1, but the sizes of the other types may vary. On a 32-bit machine, `sizeof(int)` is normally 4. Note that `sizeof` is a rather unusual operator, since the compiler itself can usually determine the value of a `sizeof` expression.

Q&A

The `sizeof` operator can also be applied to constants, variables, and expressions in general. If `i` and `j` are `int` variables, then `sizeof(i)` is 4 on a 32-bit machine, as is `sizeof(i + j)`. When applied to an expression—as opposed to a type—`sizeof` doesn't require parentheses; we could write `sizeof i` instead of `sizeof(i)`. However, parentheses may be needed anyway because of operator precedence. The compiler would interpret `sizeof i + j` as `(sizeof i) + j`, because `sizeof`—a unary operator—takes precedence over the binary + operator. To avoid problems, I always use parentheses in `sizeof` expressions.

Printing a `sizeof` value requires care, because the type of a `sizeof` expression is an implementation-defined type named `size_t`. In C89, it's best to convert the value of the expression to a known type before printing it. `size_t` is guaranteed to be an unsigned integer type, so it's safest to cast a `sizeof` expression to `unsigned long` (the largest of C89's unsigned types) and then print it using the `%lu` conversion:

```
printf("Size of int: %lu\n", (unsigned long) sizeof(int));
```

C99

In C99, the `size_t` type can be larger than `unsigned long`. However, the `printf` function in C99 is capable of displaying `size_t` values directly, without needing a cast. The trick is to use the letter `z` in the conversion specification, followed by one of the usual integer codes (typically `u`):

```
printf("Size of int: %zu\n", sizeof(int));   /* C99 only */
```

Q & A

Q: Section 7.1 says that `%o` and `%x` are used to write unsigned integers in octal and hex notation. How do I write ordinary (signed) integers in octal or hex? [p. 130]

A: You can use `%o` and `%x` to print a signed integer as long as its value isn't negative. These conversions cause `printf` to treat a signed integer as though it were unsigned; in other words, `printf` will assume that the sign bit is part of the number's magnitude. As long as the sign bit is 0, there's no problem. If the sign bit is 1, `printf` will print an unexpectedly large number.

Q: But what if the number *is* negative? How can I write it in octal or hex?

A: There's no direct way to print a negative number in octal or hex. Fortunately, the need to do so is pretty rare. You can, of course, test whether the number is negative and print a minus sign yourself:

```
if (i < 0)
  printf("-%x", -i);
else
  printf("%x", i);
```

Q: Why are floating constants stored in `double` form rather than `float` form? [p. 133]

A: For historical reasons, C gives preference to the `double` type; `float` is treated as a second-class citizen. Consider, for instance, the discussion of `float` in Kernighan and Ritchie's *The C Programming Language*: "The main reason for using `float` is to save storage in large arrays, or, less often, to save time on machines where double-precision arithmetic is particularly expensive." C originally mandated that all floating-point arithmetic be done in double precision. (C89 and C99 have no such requirement.)

***Q: What do hexadecimal floating constants look like, and what are they good for? [p. 134]**

A: A hexadecimal floating constant begins with `0x` or `0X` and must contain an exponent, which is preceded by the letter `P` (or `p`). The exponent may have a sign, and the constant may end with `f`, `F`, `l`, or `L`. The exponent is expressed in decimal, but represents a power of 2, not a power of 10. For example, `0x1.Bp3` represents the number $1.6875 \times 2^3 = 13.5$. The hex digit `B` corresponds to the bit pattern 1011. The `B` occurs to the right of the period, so each 1 bit represents a negative power of 2. Summing these powers of 2 ($2^{-1} + 2^{-3} + 2^{-4}$) yields .6875.

Hexadecimal floating constants are primarily useful for specifying constants that require great precision (including mathematical constants such as e and π). Hex numbers have a precise binary representation, whereas a constant written in decimal may be subject to a tiny rounding error when converted to binary. Hexa-

decimal numbers are also useful for defining constants with extreme values, such as the values of the macros in the `<float.h>` header. These constants are easy to write in hex but difficult to write in decimal.

***Q: Why do we use `%lf` to read a `double` value but `%f` to print it? [p. 134]**

A: This is a tough question to answer. First, notice that `scanf` and `printf` are unusual functions in that they aren't restricted to a fixed number of arguments. We say that `scanf` and `printf` have variable-length argument lists. When functions with variable-length argument lists are called, the compiler arranges for `float` arguments to be converted automatically to type `double`. As a result, `printf` can't distinguish between `float` and `double` arguments. This explains why `%f` works for both `float` and `double` arguments in calls of `printf`.

variable-length argument lists
➤26.1

`scanf`, on the other hand, is passed a *pointer* to a variable. `%f` tells `scanf` to store a `float` value at the address passed to it, while `%lf` tells `scanf` to store a `double` value at that address. The distinction between `float` and `double` is crucial here. If given the wrong conversion specification, `scanf` will likely store the wrong number of bytes (not to mention the fact that the bit pattern for a `float` isn't the same as that for a `double`).

Q: What's the proper way to pronounce `char`? [p. 134]

A: There's no universally accepted pronunciation. Some people pronounce `char` in the same way as the first syllable of "character." Others say "char," as in

```
char broiled;
```

Q: When does it matter whether a character variable is signed or unsigned? [p. 136]

A: If we store only 7-bit characters in the variable, it doesn't matter, since the sign bit will be zero. If we plan to store 8-bit characters, however, we'll probably want the variable to have `unsigned char` type. Consider the following example:

```
ch = '\xdb';
```

If `ch` has been declared to have type `char`, the compiler may choose to treat it as a signed character (many compilers do). As long as `ch` is used only as a character, there won't be any problem. But if `ch` is ever used in a context that requires the compiler to convert its value to an integer, we're likely to have trouble: the resulting integer will be negative, since `ch`'s sign bit is 1.

Here's another situation: In some kinds of programs, it's customary to use `char` variables to store one-byte integers. If we're writing such a program, we'll have to decide whether each variable should be `signed char` or `unsigned char`, just as we must decide whether ordinary integer variables should have type `int` or `unsigned int`.

Q: I don't understand how the new-line character can be the ASCII line-feed character. When a user enters input and presses the Enter key, doesn't the program read this as a carriage-return character or a carriage return plus a line feed? [p. 137]

A: Nope. As part of C's UNIX heritage, it always regards the end of a line as being marked by a single line-feed character. (In UNIX text files, a single line-feed character—but no carriage return—appears at the end of each line.) The C library takes care of translating the user's keypress into a line-feed character. When a program reads from a file, the I/O library translates the file's end-of-line marker (whatever it may be) into a single line-feed character. The same transformations occur—in reverse—when output is written to the screen or to a file. (See Section 22.1 for details.)

Although these translations may seem confusing, they serve an important purpose: insulating programs from details that may vary from one operating system to another.

***Q: What's the purpose of the \? escape sequence? [p. 138]**

A: The \? escape is related to trigraph sequences, which begin with ??. If you should put ?? in a string, there's a possibility that the compiler will mistake it for the beginning of a trigraph. Replacing the second ? by \? fixes the problem.

trigraph sequences ➤25.3

Q: If `getchar` is faster, why would we ever want to use `scanf` to read individual characters? [p. 140]

A: Although it's not as fast as `getchar`, the `scanf` function is more flexible. As we saw previously, the `"%c"` format string causes `scanf` to read the next input character; `" %c"` causes it to read the next non-white-space character. Also, `scanf` is good at reading characters that are mixed in with other kinds of data. Let's say that our input data consists of an integer, then a single nonnumeric character, then another integer. By using the format string `"%d%c%d"`, we can get `scanf` to read all three items.

***Q: Under what circumstances do the integral promotions convert a character or short integer to `unsigned int`? [p. 143]**

A: The integral promotions yield an `unsigned int` if the `int` type isn't large enough to include all possible values of the original type. Since characters are usually eight bits long, they are almost always converted to `int`, which is guaranteed to be at least 16 bits long. Signed short integers can always be converted to `int` as well. Unsigned short integers are problematic. If short integers have the same length as ordinary integers (as they do on a 16-bit machine), then unsigned short integers will have to be converted to `unsigned int`, since the largest unsigned short integer (65,535 on a 16-bit machine) is larger than the largest `int` (32,767).

Q: Exactly what happens if I assign a value to a variable that's not large enough to hold it? [p. 146]

A: Roughly speaking, if the value is of an integral type and the variable is of an unsigned type, the extra bits are thrown away; if the variable has a signed type, the result is implementation-defined. Assigning a floating-point number to a variable—integer or floating—that's too small to hold it produces undefined behavior: anything can happen, including program termination.

***Q: Why does C bother to provide type definitions? Isn't defining a `BOOL` macro just as good as defining a `Bool` type using `typedef`? [p. 149]**

A: There are two important differences between type definitions and macro definitions. First, type definitions are more powerful than macro definitions. In particular, array and pointer types can't be defined as macros. Suppose that we try to use a macro to define a "pointer to integer" type:

```
#define PTR_TO_INT int *
```

The declaration

```
PTR_TO_INT p, q, r;
```

will become

```
int * p, q, r;
```

after preprocessing. Unfortunately, only `p` is a pointer; `q` and `r` are ordinary integer variables. Type definitions don't have this problem.

Second, `typedef` names are subject to the same scope rules as variables; a `typedef` name defined inside a function body wouldn't be recognized outside the function. Macro names, on the other hand, are replaced by the preprocessor wherever they appear.

***Q: You said that compilers "can usually determine the value of a `sizeof` expression." Can't a compiler *always* determine the value of a `sizeof` expression? [p. 151]**

A: In C89, yes. In C99, however, there's one exception. The compiler can't determine the size of a variable-length array, because the number of elements in the array may change during the execution of the program.

variable-length arrays ➤8.3

Exercises

Section 7.1

1. Give the decimal value of each of the following integer constants.
   (a) `077`
   (b) `0x77`
   (c) `0XABC`

Section 7.2

2. Which of the following are not legal constants in C? Classify each legal constant as either integer or floating-point.
   (a) `010E2`
   (b) `32.1E+5`
   (c) `0790`
   (d) `100_000`
   (e) `3.978e-2`

W 3. Which of the following are not legal types in C?
(a) `short unsigned int`
(b) `short float`
(c) `long double`
(d) `unsigned long`

Section 7.3

W 4. If `c` is a variable of type `char`, which one of the following statements is illegal?
(a) `i += c;  /* i has type int */`
(b) `c = 2 * c - 1;`
(c) `putchar(c);`
(d) `printf(c);`

5. Which one of the following is not a legal way to write the number 65? (Assume that the character set is ASCII.)
(a) `'A'`
(b) `0b1000001`
(c) `0101`
(d) `0x41`

6. For each of the following items of data, specify which one of the types `char`, `short`, `int`, or `long` is the smallest one guaranteed to be large enough to store the item.
(a) Days in a month
(b) Days in a year
(c) Minutes in a day
(d) Seconds in a day

7. For each of the following character escapes, give the equivalent octal escape. (Assume that the character set is ASCII.) You may wish to consult Appendix E, which lists the numerical codes for ASCII characters.
(a) `\b`
(b) `\n`
(c) `\r`
(d) `\t`

8. Repeat Exercise 7, but give the equivalent hexadecimal escape.

Section 7.4

9. Suppose that `i` and `j` are variables of type `int`. What is the type of the expression `i / j + 'a'`?

W 10. Suppose that `i` is a variable of type `int`, `j` is a variable of type `long`, and `k` is a variable of type `unsigned int`. What is the type of the expression `i + (int) j * k`?

11. Suppose that `i` is a variable of type `int`, `f` is a variable of type `float`, and `d` is a variable of type `double`. What is the type of the expression `i * f / d`?

W 12. Suppose that `i` is a variable of type `int`, `f` is a variable of type `float`, and `d` is a variable of type `double`. Explain what conversions take place during the execution of the following statement:

```
d = i + f;
```

13. Assume that a program contains the following declarations:

```
char c = '\1';
short s = 2;
int i = -3;
long m = 5;
float f = 6.5f;
double d = 7.5;
```

Give the value and the type of each expression listed below.

(a) `c * i`	(c) `f / c`	(e) `f - d`
(b) `s + m`	(d) `d / s`	(f) `(int) f`

W 14. Does the following statement always compute the fractional part of `f` correctly (assuming that `f` and `frac_part` are `float` variables)?

```
frac_part = f - (int) f;
```

If not, what's the problem?

Section 7.5

15. Use `typedef` to create types named `Int8`, `Int16`, and `Int32`. Define the types so that they represent 8-bit, 16-bit, and 32-bit integers on your machine.

Programming Projects

W 1. The `square2.c` program of Section 6.3 will fail (usually by printing strange answers) if `i * i` exceeds the maximum `int` value. Run the program and determine the smallest value of `n` that causes failure. Try changing the type of `i` to `short` and running the program again. (Don't forget to update the conversion specifications in the call of `printf`!) Then try `long`. From these experiments, what can you conclude about the number of bits used to store integer types on your machine?

W 2. Modify the `square2.c` program of Section 6.3 so that it pauses after every 24 squares and displays the following message:

```
Press Enter to continue...
```

After displaying the message, the program should use `getchar` to read a character. `getchar` won't allow the program to continue until the user presses the Enter key.

3. Modify the `sum2.c` program of Section 7.1 to sum a series of `double` values.

4. Write a program that translates an alphabetic phone number into numeric form:

```
Enter phone number: CALLATT
2255288
```

(In case you don't have a telephone nearby, here are the letters on the keys: 2=ABC, 3=DEF, 4=GHI, 5=JKL, 6=MNO, 7=PRS, 8=TUV, 9=WXY.) If the original phone number contains nonalphabetic characters (digits or punctuation, for example), leave them unchanged:

```
Enter phone number: 1-800-COL-LECT
1-800-265-5328
```

You may assume that any letters entered by the user are upper case.

W 5. In the SCRABBLE Crossword Game, players form words using small tiles, each containing a letter and a face value. The face value varies from one letter to another, based on the letter's rarity. (Here are the face values: 1: AEILNORSTU, 2: DG, 3: BCMP, 4: FHVWY, 5: K, 8: JX, 10: QZ.) Write a program that computes the value of a word by summing the values of its letters:

```
Enter a word: pitfall
Scrabble value: 12
```

Your program should allow any mixture of lower-case and upper-case letters in the word. *Hint:* Use the `toupper` library function.

W 6. Write a program that prints the values of `sizeof(int)`, `sizeof(short)`, `sizeof(long)`, `sizeof(float)`, `sizeof(double)` and `sizeof(long double)`.

7. Modify Programming Project 6 from Chapter 3 so that the user may add, subtract, multiply, or divide two fractions (by entering either `+`, `-`, `*`, or `/` between the fractions).

8. Modify Programming Project 8 from Chapter 5 so that the user enters a time using the 12-hour clock. The input will have the form *hours*`:`*minutes* followed by either `A`, `P`, `AM`, or `PM` (either lower-case or upper-case). White space is allowed (but not required) between the numerical time and the AM/PM indicator. Examples of valid input:

```
1:15P
1:15PM
1:15p
1:15pm
1:15 P
1:15 PM
1:15 p
1:15 pm
```

You may assume that the input has one of these forms; there is no need to test for errors.

9. Write a program that asks the user for a 12-hour time, then displays the time in 24-hour form:

```
Enter a 12-hour time: 9:11 PM
Equivalent 24-hour time: 21:11
```

See Programming Project 8 for a description of the input format.

10. Write a program that counts the number of vowels (*a, e, i, o,* and *u*) in a sentence:

```
Enter a sentence: And that's the way it is.
Your sentence contains 6 vowels.
```

11. Write a program that takes a first name and last name entered by the user and displays the last name, a comma, and the first initial, followed by a period:

```
Enter a first and last name: Lloyd Fosdick
Fosdick, L.
```

The user's input may contain extra spaces before the first name, between the first and last names, and after the last name.

12. Write a program that evaluates an expression:

```
Enter an expression: 1+2.5*3
Value of expression: 10.5
```

The operands in the expression are floating-point numbers; the operators are +, -, *, and /. The expression is evaluated from left to right (no operator takes precedence over any other operator).

13. Write a program that calculates the average word length for a sentence:

```
Enter a sentence: It was deja vu all over again.
Average word length: 3.4
```

For simplicity, your program should consider a punctuation mark to be part of the word to which it is attached. Display the average word length to one decimal place.

14. Write a program that uses Newton's method to compute the square root of a positive floating-point number:

```
Enter a positive number: 3
Square root: 1.73205
```

Let x be the number entered by the user. Newton's method requires an initial guess y for the square root of x (we'll use $y = 1$). Successive guesses are found by computing the average of y and x/y. The following table shows how the square root of 3 would be found:

x	y	x/y	Average of y and x/y
3	1	3	2
3	2	1.5	1.75
3	1.75	1.71429	1.73214
3	1.73214	1.73196	1.73205
3	1.73205	1.73205	1.73205

Note that the values of y get progressively closer to the true square root of x. For greater accuracy, your program should use variables of type `double` rather than `float`. Have the program terminate when the absolute value of the difference between the old value of y and the new value of y is less than the product of .00001 and y. *Hint:* Call the `fabs` function to find the absolute value of a `double`. (You'll need to include the `<math.h>` header at the beginning of your program in order to use `fabs`.)

15. Write a program that computes the factorial of a positive integer:

```
Enter a positive integer: 6
Factorial of 6: 720
```

(a) Use a `short` variable to store the value of the factorial. What is the largest value of n for which the program correctly prints the factorial of n?
(b) Repeat part (a), using an `int` variable instead.
(c) Repeat part (a), using a `long` variable instead.
(d) Repeat part (a), using a `long long` variable instead (if your compiler supports the `long long` type).
(e) Repeat part (a), using a `float` variable instead.
(f) Repeat part (a), using a `double` variable instead.
(g) Repeat part (a), using a `long double` variable instead.

In cases (e)–(g), the program will display a close approximation of the factorial, not necessarily the exact value.

8 Arrays

If a program manipulates a large amount of data,
it does so in a small number of ways.

So far, the only variables we've seen are ***scalar:*** capable of holding a single data item. C also supports ***aggregate*** variables, which can store collections of values. There are two kinds of aggregates in C: arrays and structures. This chapter shows how to declare and use arrays, both one-dimensional (Section 8.1) and multidimensional (Section 8.2). Section 8.3 covers C99's variable-length arrays. The focus of the chapter is on one-dimensional arrays, which play a much bigger role in C than do multidimensional arrays. Later chapters (Chapter 12 in particular) provide additional information about arrays; Chapter 16 covers structures.

8.1 One-Dimensional Arrays

An ***array*** is a data structure containing a number of data values, all of which have the same type. These values, known as ***elements***, can be individually selected by their position within the array.

The simplest kind of array has just one dimension. The elements of a one-dimensional array are conceptually arranged one after another in a single row (or column, if you prefer). Here's how we might visualize a one-dimensional array named `a`:

To declare an array, we must specify the *type* of the array's elements and the *number* of elements. For example, to declare that the array `a` has 10 elements of type `int`, we would write

```
int a[10];
```

The elements of an array may be of any type; the length of the array can be specified by any (integer) constant expression. Since array lengths may need to be adjusted when the program is later changed, using a macro to define the length of an array is an excellent practice:

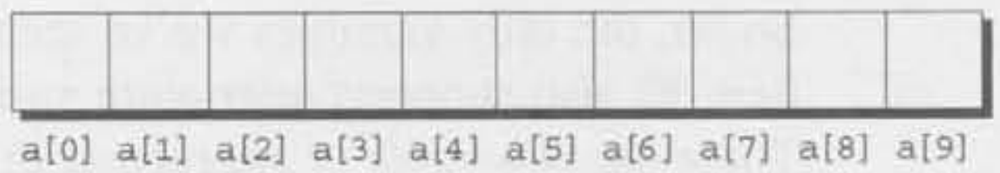
constant expressions ➤5.3

```
#define N 10
...
int a[N];
```

Array Subscripting

To access a particular element of an array, we write the array name followed by an integer value in square brackets (this is referred to as ***subscripting*** or ***indexing*** the array). Array elements are always numbered starting from 0, so the elements of an array of length n are indexed from 0 to $n - 1$. For example, if `a` is an array with 10 elements, they're designated by `a[0]`, `a[1]`, ..., `a[9]`, as the following figure shows:

Q&A



Expressions of the form `a[i]` are lvalues, so they can be used in the same way as ordinary variables:

lvalues ➤4.2

```
a[0] = 1;
printf("%d\n", a[5]);
++a[i];
```

In general, if an array contains elements of type T, then each element of the array is treated as if it were a variable of type T. In this example, the elements `a[0]`, `a[5]`, and `a[i]` behave like `int` variables.

Arrays and `for` loops go hand-in-hand. Many programs contain `for` loops whose job is to perform some operation on every element in an array. Here are a few examples of typical operations on an array `a` of length `N`:

idiom
```
for (i = 0; i < N; i++)
  a[i] = 0;                    /* clears a */
```

idiom
```
for (i = 0; i < N; i++)
  scanf("%d", &a[i]);          /* reads data into a */
```

idiom
```
for (i = 0; i < N; i++)
  sum += a[i];                 /* sums the elements of a */
```

Notice that we must use the `&` symbol when calling `scanf` to read an array element, just as we would with an ordinary variable.

C doesn't require that subscript bounds be checked; if a subscript goes out of range, the program's behavior is undefined. One cause of a subscript going out of bounds: forgetting that an array with n elements is indexed from 0 to $n - 1$, not 1 to n. (As one of my professors liked to say, "In this business, you're always off by one." He was right, of course.) The following example illustrates a bizarre effect that can be caused by this common blunder:

```
int a[10], i;

for (i = 1; i <= 10; i++)
  a[i] = 0;
```

With some compilers, this innocent-looking `for` statement causes an infinite loop! When `i` reaches 10, the program stores 0 into `a[10]`. But `a[10]` doesn't exist, so 0 goes into memory immediately after `a[9]`. If the variable `i` happens to follow `a[9]` in memory—as might be the case—then `i` will be reset to 0, causing the loop to start over.

An array subscript may be any integer expression:

```
a[i+j*10] = 0;
```

The expression can even have side effects:

```
i = 0;
while (i < N)
  a[i++] = 0;
```

Let's trace this code. After `i` is set to 0, the `while` statement checks whether `i` is less than `N`. If it is, 0 is assigned to `a[0]`, `i` is incremented, and the loop repeats. Note that `a[++i]` wouldn't be right, because 0 would be assigned to `a[1]` during the first loop iteration.

Be careful when an array subscript has a side effect. For example, the following loop—which is supposed to copy the elements of the array `b` into the array `a`—may not work properly:

```
i = 0;
while (i < N)
  a[i] = b[i++];
```

The expression `a[i] = b[i++]` accesses the value of `i` and also modifies `i` elsewhere in the expression, which—as we saw in Section 4.4—causes undefined behavior. Of course, we can easily avoid the problem by removing the increment from the subscript:

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

PROGRAM

Reversing a Series of Numbers

Our first array program prompts the user to enter a series of numbers, then writes the numbers in reverse order:

```
Enter 10 numbers: 34 82 49 102 7 94 23 11 50 31
In reverse order: 31 50 11 23 94 7 102 49 82 34
```

Our strategy will be to store the numbers in an array as they're read, then go through the array backwards, printing the elements one by one. In other words, we won't actually reverse the elements in the array, but we'll make the user think we did.

reverse.c

```
/* Reverses a series of numbers */

#include <stdio.h>

#define N 10

int main(void)
{
  int a[N], i;

  printf("Enter %d numbers: ", N);
  for (i = 0; i < N; i++)
    scanf("%d", &a[i]);

  printf("In reverse order:");
  for (i = N - 1; i >= 0; i--)
    printf(" %d", a[i]);
  printf("\n");

  return 0;
}
```

This program shows just how useful macros can be in conjunction with arrays. The macro `N` is used four times in the program: in the declaration of `a`, in the `printf` that displays a prompt, and in both `for` loops. Should we later decide to change the size of the array, we need only edit the definition of `N` and recompile the program. Nothing else will need to be altered; even the prompt will still be correct.

Array Initialization

An array, like any other variable, can be given an initial value at the time it's declared. The rules are somewhat tricky, though, so we'll cover some of them now and save others until later.

initializers ➤ 18.5

The most common form of ***array initializer*** is a list of constant expressions enclosed in braces and separated by commas:

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

If the initializer is *shorter* than the array, the remaining elements of the array are given the value 0:

```
int a[10] = {1, 2, 3, 4, 5, 6};
  /* initial value of a is {1, 2, 3, 4, 5, 6, 0, 0, 0, 0} */
```

Using this feature, we can easily initialize an array to all zeros:

```
int a[10] = {0};
  /* initial value of a is {0, 0, 0, 0, 0, 0, 0, 0, 0, 0} */
```

It's illegal for an initializer to be completely empty, so we've put a single `0` inside the braces. It's also illegal for an initializer to be *longer* than the array it initializes.

If an initializer is present, the length of the array may be omitted:

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

The compiler uses the length of the initializer to determine how long the array is. The array still has a fixed number of elements (10, in this example), just as if we had specified the length explicitly.

C99 Designated Initializers

It's often the case that relatively few elements of an array need to be initialized explicitly; the other elements can be given default values. Consider the following example:

```
int a[15] = {0, 0, 29, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 48};
```

We want element 2 of the array to be 29, element 9 to be 7, and element 14 to be 48, but the other values are just zero. For a large array, writing an initializer in this fashion is tedious and error-prone (what if there were 200 zeros between two of the nonzero values?).

C99's ***designated initializers*** can be used to solve this problem. Here's how we could redo the previous example using a designated initializer:

```
int a[15] = {[2] = 29, [9] = 7, [14] = 48};
```

Each number in brackets is said to be a ***designator.***

Besides being shorter and easier to read (at least for some arrays), designated initializers have another advantage: the order in which the elements are listed no longer matters. Thus, our previous example could also be written in the following way:

```
int a[15] = {[14] = 48, [9] = 7, [2] = 29};
```

Designators must be integer constant expressions. If the array being initialized has length n, each designator must be between 0 and $n - 1$. However, if the length of the array is omitted, a designator can be any nonnegative integer. In the latter case, the compiler will deduce the length of the array from the largest designator.

In the following example, the fact that 23 appears as a designator will force the array to have length 24:

```
int b[] = {[5] = 10, [23] = 13, [11] = 36, [15] = 29};
```

An initializer may use both the older (element-by-element) technique and the newer (designated) technique:

```
int c[10] = {5, 1, 9, [4] = 3, 7, 2, [8] = 6};
```

Q&A This initializer specifies that the array's first three elements will be 5, 1, and 9. Element 4 will have the value 3. The two elements after element 4 will be 7 and 2. Finally, element 8 will have the value 6. All elements for which no value is specified will default to zero.

PROGRAM Checking a Number for Repeated Digits

Our next program checks whether any of the digits in a number appear more than once. After the user enters a number, the program prints either `Repeated digit` or `No repeated digit`:

```
Enter a number: 28212
Repeated digit
```

The number 28212 has a repeated digit (2); a number like 9357 doesn't.

The program uses an array of Boolean values to keep track of which digits appear in a number. The array, named `digit_seen`, is indexed from 0 to 9 to correspond to the 10 possible digits. Initially, every element of the array is false. (The initializer for `digit_seen` is `{false}`, which only initializes the first element of the array. However, the compiler will automatically make the remaining elements zero, which is equivalent to false.)

When given a number `n`, the program examines `n`'s digits one at a time, storing each into the `digit` variable and then using it as an index into `digit_seen`. If `digit_seen[digit]` is true, then `digit` appears at least twice in `n`. On the other hand, if `digit_seen[digit]` is false, then `digit` has not been seen before, so the program sets `digit_seen[digit]` to `true` and keeps going.

repdigit.c

```
/* Checks numbers for repeated digits */

#include <stdbool.h>   /* C99 only */
#include <stdio.h>

int main(void)
{
  bool digit_seen[10] = {false};
  int digit;
  long n;

  printf("Enter a number: ");
  scanf("%ld", &n);
```

```
while (n > 0) {
  digit = n % 10;
  if (digit_seen[digit])
    break;
  digit_seen[digit] = true;
  n /= 10;
}

if (n > 0)
  printf("Repeated digit\n");
else
  printf("No repeated digit\n");

return 0;
}
```

C99

<stdbool.h> header ➤21.5

This program uses the names `bool`, `true`, and `false`, which are defined in C99's `<stdbool.h>` header. If your compiler doesn't support this header, you'll need to define these names yourself. One way to do so is to put the following lines above the `main` function:

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

Notice that `n` has type `long`, allowing the user to enter numbers up to 2,147,483,647 (or more, on some machines).

Using the `sizeof` Operator with Arrays

The `sizeof` operator can determine the size of an array (in bytes). If `a` is an array of 10 integers, then `sizeof(a)` is typically 40 (assuming that each integer requires four bytes).

We can also use `sizeof` to measure the size of an array element, such as `a[0]`. Dividing the array size by the element size gives the length of the array:

```
sizeof(a) / sizeof(a[0])
```

Some programmers use this expression when the length of the array is needed. To clear the array `a`, for example, we could write

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

With this technique, the loop doesn't have to be modified if the array length should change at a later date. Using a macro to represent the array length has the same advantage, of course, but the `sizeof` technique is slightly better, since there's no macro name to remember (and possibly get wrong).

One minor annoyance is that some compilers produce a warning message for the expression `i < sizeof(a) / sizeof(a[0])`. The variable `i` probably has

type `int` (a signed type), whereas `sizeof` produces a value of type `size_t` (an unsigned type). We know from Section 7.4 that comparing a signed integer with an unsigned integer is a dangerous practice, although in this case it's safe because both `i` and `sizeof(a) / sizeof(a[0])` have nonnegative values. To avoid a warning, we can add a cast that converts `sizeof(a) / sizeof(a[0])` to a signed integer:

```
for (i = 0; i < (int) (sizeof(a) / sizeof(a[0])); i++)
  a[i] = 0;
```

Writing `(int) (sizeof(a) / sizeof(a[0]))` is a bit unwieldy; defining a macro that represents it is often helpful:

```
#define SIZE ((int) (sizeof(a) / sizeof(a[0])))

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

If we're back to using a macro, though, what's the advantage of `sizeof`? We'll answer that question in a later chapter (the trick is to add a parameter to the macro).

parameterized macros ➤ 14.3

PROGRAM Computing Interest

Our next program prints a table showing the value of $100 invested at different rates of interest over a period of years. The user will enter an interest rate and the number of years the money will be invested. The table will show the value of the money at one-year intervals—at that interest rate and the next four higher rates—assuming that interest is compounded once a year. Here's what a session with the program will look like:

```
Enter interest rate: 6
Enter number of years: 5

Years     6%      7%      8%      9%     10%
  1     106.00  107.00  108.00  109.00  110.00
  2     112.36  114.49  116.64  118.81  121.00
  3     119.10  122.50  125.97  129.50  133.10
  4     126.25  131.08  136.05  141.16  146.41
  5     133.82  140.26  146.93  153.86  161.05
```

Clearly, we can use a `for` statement to print the first row. The second row is a little trickier, since its values depend on the numbers in the first row. Our solution is to store the first row in an array as it's computed, then use the values in the array to compute the second row. Of course, this process can be repeated for the third and later rows. We'll end up with two `for` statements, one nested inside the other. The outer loop will count from 1 to the number of years requested by the user. The inner loop will increment the interest rate from its lowest value to its highest value.

interest.c

```
/* Prints a table of compound interest */

#include <stdio.h>

#define NUM_RATES ((int) (sizeof(value) / sizeof(value[0])))
#define INITIAL_BALANCE 100.00

int main(void)
{
  int i, low_rate, num_years, year;
  double value[5];

  printf("Enter interest rate: ");
  scanf("%d", &low_rate);
  printf("Enter number of years: ");
  scanf("%d", &num_years);

  printf("\nYears");
  for (i = 0; i < NUM_RATES; i++) {
    printf("%6d%%", low_rate + i);
    value[i] = INITIAL_BALANCE;
  }
  printf("\n");

  for (year = 1; year <= num_years; year++) {
    printf("%3d    ", year);
    for (i = 0; i < NUM_RATES; i++) {
      value[i] += (low_rate + i) / 100.0 * value[i];
      printf("%7.2f", value[i]);
    }
    printf("\n");
  }

  return 0;
}
```

Note the use of `NUM_RATES` to control two of the `for` loops. If we later change the size of the `value` array, the loops will adjust automatically.

8.2 Multidimensional Arrays

An array may have any number of dimensions. For example, the following declaration creates a two-dimensional array (a *matrix*, in mathematical terminology):

```
int m[5][9];
```

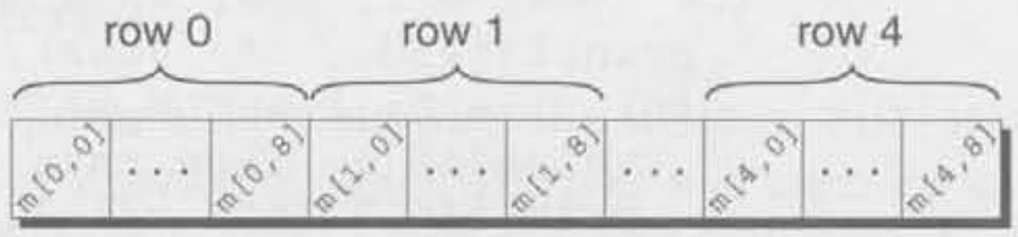
The array `m` has 5 rows and 9 columns. Both rows and columns are indexed from 0, as the following figure shows:

To access the element of `m` in row `i`, column `j`, we must write `m[i][j]`. The expression `m[i]` designates row `i` of `m`, and `m[i][j]` then selects element `j` in this row.

Resist the temptation to write `m[i,j]` instead of `m[i][j]`. C treats the comma as an operator in this context, so `m[i,j]` is the same as `m[j]`.

comma operator ➤6.3

Although we visualize two-dimensional arrays as tables, that's not the way they're actually stored in computer memory. C stores arrays in ***row-major order,*** with row 0 first, then row 1, and so forth. For example, here's how the `m` array is stored:



We'll usually ignore this detail, but sometimes it will affect our code.

Just as `for` loops go hand-in-hand with one-dimensional arrays, nested `for` loops are ideal for processing multidimensional arrays. Consider, for example, the problem of initializing an array for use as an identity matrix. (In mathematics, an *identity matrix* has 1's on the main diagonal, where the row and column index are the same, and 0's everywhere else.) We'll need to visit each element in the array in some systematic fashion. A pair of nested `for` loops—one that steps through every row index and one that steps through each column index—is perfect for the job:

```
#define N 10

double ident[N][N];
int row, col;

for (row = 0; row < N; row++)
  for (col = 0; col < N; col++)
    if (row == col)
      ident[row][col] = 1.0;
    else
      ident[row][col] = 0.0;
```

Multidimensional arrays play a lesser role in C than in many other programming languages, primarily because C provides a more flexible way to store multidimensional data: arrays of pointers.

arrays of pointers ➤13.7

Initializing a Multidimensional Array

We can create an initializer for a two-dimensional array by nesting one-dimensional initializers:

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1, 0, 1, 0},
               {0, 1, 0, 1, 1, 0, 0, 1, 0},
               {1, 1, 0, 1, 0, 0, 0, 1, 0},
               {1, 1, 0, 1, 0, 0, 1, 1, 1}};
```

Each inner initializer provides values for one row of the matrix. Initializers for higher-dimensional arrays are constructed in a similar fashion.

C provides a variety of ways to abbreviate initializers for multidimensional arrays:

- If an initializer isn't large enough to fill a multidimensional array, the remaining elements are given the value 0. For example, the following initializer fills only the first three rows of `m`; the last two rows will contain zeros:

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1, 0, 1, 0},
               {0, 1, 0, 1, 1, 0, 0, 1, 0}};
```

- If an inner list isn't long enough to fill a row, the remaining elements in the row are initialized to 0:

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1, 0, 1},
               {0, 1, 0, 1, 1, 0, 0, 1},
               {1, 1, 0, 1, 0, 0, 0, 1},
               {1, 1, 0, 1, 0, 0, 1, 1, 1}};
```

- We can even omit the inner braces:

```
int m[5][9] = {1, 1, 1, 1, 1, 0, 1, 1, 1,
               0, 1, 0, 1, 0, 1, 0, 1, 0,
               0, 1, 0, 1, 1, 0, 0, 1, 0,
               1, 1, 0, 1, 0, 0, 0, 1, 0,
               1, 1, 0, 1, 0, 0, 1, 1, 1};
```

  Once the compiler has seen enough values to fill one row, it begins filling the next.

⚠ Omitting the inner braces in a multidimensional array initializer can be risky, since an extra element (or even worse, a missing element) will affect the rest of the initializer. Leaving out the braces causes some compilers to produce a warning message such as *"missing braces around initializer."*

C99 C99's designated initializers work with multidimensional arrays. For example, we could create a 2×2 identity matrix as follows:

```
double ident[2][2] = {[0][0] = 1.0, [1][1] = 1.0};
```

As usual, all elements for which no value is specified will default to zero.

Constant Arrays

Any array, whether one-dimensional or multidimensional, can be made "constant" by starting its declaration with the word `const`:

```
const char hex_chars[] =
  {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
   'A', 'B', 'C', 'D', 'E', 'F'};
```

An array that's been declared `const` should not be modified by the program; the compiler will detect direct attempts to modify an element.

Declaring an array to be `const` has a couple of primary advantages. It documents that the program won't change the array, which can be valuable information for someone reading the code later. It also helps the compiler catch errors, by informing it that we don't intend to modify the array.

`const` type qualifier ➤ *18.3*

`const` isn't limited to arrays; it works with any variable, as we'll see later. However, `const` is particularly useful in array declarations, because arrays may contain reference information that won't change during program execution.

PROGRAM

Dealing a Hand of Cards

Our next program illustrates both two-dimensional arrays and constant arrays. The program deals a random hand from a standard deck of playing cards. (In case you haven't had time to play games recently, each card in a standard deck has a *suit*—clubs, diamonds, hearts, or spades—and a *rank*—two, three, four, five, six, seven, eight, nine, ten, jack, queen, king, or ace.) We'll have the user specify how many cards should be in the hand:

```
Enter number of cards in hand: 5
Your hand: 7c 2s 5d as 2h
```

It's not immediately obvious how we'd write such a program. How do we pick cards randomly from the deck? And how do we avoid picking the same card twice? Let's tackle these problems separately.

`time` function ➤ *26.3*

To pick cards randomly, we'll use several C library functions. The `time` function (from `<time.h>`) returns the current time, encoded in a single number. The

`srand` function ➤ *26.2*

`srand` function (from `<stdlib.h>`) initializes C's random number generator. Passing the return value of `time` to `srand` prevents the program from dealing the same cards every time we run it. The `rand` function (also from `<stdlib.h>`) produces an apparently random number each time it's called. By using the `%` operator, we can scale the return value from `rand` so that it falls between 0 and 3 (for suits) or between 0 and 12 (for ranks).

`rand` function ➤ *26.2*

To avoid picking the same card twice, we'll need to keep track of which cards have already been chosen. For that purpose, we'll use an array named `in_hand`

that has four rows (one for each suit) and 13 columns (one for each rank). In other words, each element in the array corresponds to one of the 52 cards in the deck. All elements of the array will be false to start with. Each time we pick a card at random, we'll check whether the element of `in_hand` corresponding to that card is true or false. If it's true, we'll have to pick another card. If it's false, we'll store `true` in that card's array element to remind us later that this card has already been picked.

Once we've verified that a card is "new"—not already selected—we'll need to translate its numerical rank and suit into characters and then display the card. To translate the rank and suit to character form, we'll set up two arrays of characters—one for the rank and one for the suit—and then use the numbers to subscript the arrays. These arrays won't change during program execution, so we may as well declare them to be `const`.

deal.c

```
/* Deals a random hand of cards */

#include <stdbool.h>   /* C99 only */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define NUM_SUITS 4
#define NUM_RANKS 13

int main(void)
{
  bool in_hand[NUM_SUITS][NUM_RANKS] = {false};
  int num_cards, rank, suit;
  const char rank_code[] = {'2','3','4','5','6','7','8',
                            '9','t','j','q','k','a'};
  const char suit_code[] = {'c','d','h','s'};

  srand((unsigned) time(NULL));

  printf("Enter number of cards in hand: ");
  scanf("%d", &num_cards);

  printf("Your hand:");
  while (num_cards > 0) {
    suit = rand() % NUM_SUITS;     /* picks a random suit */
    rank = rand() % NUM_RANKS;     /* picks a random rank */
    if (!in_hand[suit][rank]) {
      in_hand[suit][rank] = true;
      num_cards--;
      printf(" %c%c", rank_code[rank], suit_code[suit]);
    }
  }
  printf("\n");

  return 0;
}
```

Notice the initializer for the `in_hand` array:

```
bool in_hand[NUM_SUITS][NUM_RANKS] = {false};
```

Even though `in_hand` is a two-dimensional array, we can use a single pair of braces (at the risk of possibly incurring a warning from the compiler). Also, we've supplied only one value in the initializer, knowing that the compiler will fill in 0 (false) for the other elements.

8.3 Variable-Length Arrays (C99)

Section 8.1 stated that the length of an array variable must be specified by a constant expression. In C99, however, it's sometimes possible to use an expression that's *not* constant. The following modification of the `reverse.c` program (Section 8.1) illustrates this ability:

reverse2.c

```
/* Reverses a series of numbers using a variable-length
   array - C99 only */

#include <stdio.h>

int main(void)
{
  int i, n;

  printf("How many numbers do you want to reverse? ");
  scanf("%d", &n);

  int a[n];   /* C99 only - length of array depends on n */

  printf("Enter %d numbers: ", n);
  for (i = 0; i < n; i++)
    scanf("%d", &a[i]);

  printf("In reverse order:");
  for (i = n - 1; i >= 0; i--)
    printf(" %d", a[i]);
  printf("\n");

  return 0;
}
```

The array `a` in this program is an example of a ***variable-length array*** (or ***VLA*** for short). The length of a VLA is computed when the program is executed, not when the program is compiled. The chief advantage of a VLA is that the programmer doesn't have to pick an arbitrary length when declaring an array; instead, the program itself can calculate exactly how many elements are needed. If the programmer makes the choice, it's likely that the array will be too long (wasting memory) or too short (causing the program to fail). In the `reverse2.c` program, the num-

ber entered by the user determines the length of a; the programmer doesn't have to choose a fixed length, unlike in the original version of the program.

The length of a VLA doesn't have to be specified by a single variable. Arbitrary expressions, possibly containing operators, are also legal. For example:

```
int a[3*i+5];
int b[j+k];
```

Like other arrays, VLAs can be multidimensional:

```
int c[m][n];
```

static storage duration ➤ *18.2*

The primary restriction on VLAs is that they can't have static storage duration. (We haven't yet seen any arrays with this property.) Another restriction is that a VLA may not have an initializer.

Variable-length arrays are most often seen in functions other than `main`. One big advantage of a VLA that belongs to a function `f` is that it can have a different length each time `f` is called. We'll explore this feature in Section 9.3.

Q & A

Q: Why do array subscripts start at 0 instead of 1? [p. 162]

A: Having subscripts begin at 0 simplifies the compiler a bit. Also, it can make array subscripting marginally faster.

Q: What if I want an array with subscripts that go from 1 to 10 instead of 0 to 9?

A: Here's a common trick: declare the array to have 11 elements instead of 10. The subscripts will go from 0 to 10, but you can just ignore element 0.

Q: Is it possible to use a character as an array subscript?

A: Yes, because C treats characters as integers. You'll probably need to "scale" the character before you use it as a subscript, though. Let's say that we want the `letter_count` array to keep track of a count for each letter in the alphabet. The array will need 26 elements, so we'd declare it in the following way:

```
int letter_count[26];
```

However, we can't use letters to subscript `letter_count` directly, because their integer values don't fall between 0 and 25. To scale a lower-case letter to the proper range, we can simply subtract `'a'`; to scale an upper-case letter, we'll subtract `'A'`. For example, if `ch` contains a lower-case letter, we'd write

```
letter_count[ch-'a'] = 0;
```

to clear the count that corresponds to `ch`. A minor caveat: this technique isn't completely portable, because it assumes that letters have consecutive codes. However, it works with most character sets, including ASCII.

Q: It seems like a designated initializer could end up initializing an array element more than once. Consider the following array declaration:

```
int a[] = {4, 9, 1, 8, [0] = 5, 7};
```

Is this declaration legal, and if so, what is the length of the array? [p. 166]

A: Yes, the declaration is legal. Here's how it works: as it processes an initializer list, the compiler keeps track of which array element is to be initialized next. Normally, the next element is the one following the element that was last initialized. However, when a designator appears in the list, it forces the next element be the one represented by the designator, *even if that element has already been initialized.*

Here's a step-by-step look at how the compiler will process the initializer for the array `a`:

The 4 initializes element 0; the next element to be initialized is element 1.
The 9 initializes element 1; the next element to be initialized is element 2.
The 1 initializes element 2; the next element to be initialized is element 3.
The 8 initializes element 3; the next element to be initialized is element 4.
The [0] designator causes the next element to become 0, so the 5 initializes element 0 (replacing the 4 previously stored there). The next element to be initialized is element 1.
The 7 initializes element 1 (replacing the 9 previously stored there). The next element to be initialized is element 2 (which is irrelevant since we're at the end of the list).

The net effect is the same as if we had written

```
int a[] = {5, 7, 1, 8};
```

Thus, the length of this array is four.

Q: The compiler gives me an error message if I try to copy one array into another by using the assignment operator. What's wrong?

A: Although it looks quite plausible, the assignment

```
a = b;    /* a and b are arrays */
```

is indeed illegal. The reason for its illegality isn't obvious; it has to do with the peculiar relationship between arrays and pointers in C, a topic we'll explore in Chapter 12.

The simplest way to copy one array into another is to use a loop that copies the elements, one by one:

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

memcpy function ➤23.6

Another possibility is to use the `memcpy` ("memory copy") function from the `<string.h>` header. `memcpy` is a low-level function that simply copies bytes from one place to another. To copy the array `b` into the array `a`, use `memcpy` as follows:

```
memcpy(a, b, sizeof(a));
```

Many programmers prefer `memcpy`, especially for large arrays, because it's potentially faster than an ordinary loop.

***Q: Section 6.4 mentioned that C99 doesn't allow a `goto` statement to bypass the declaration of a variable-length array. What's the reason for this restriction?**

A: The memory used to store a variable-length array is usually allocated when the declaration of the array is reached during program execution. Bypassing the declaration using a `goto` statement could result in a program accessing the elements of an array that was never allocated.

Exercises

Section 8.1

W 1. We discussed using the expression `sizeof(a) / sizeof(a[0])` to calculate the number of elements in an array. The expression `sizeof(a) / sizeof(`*t*`)`, where *t* is the type of `a`'s elements, would also work, but it's considered an inferior technique. Why?

W 2. The Q&A section shows how to use a *letter* as an array subscript. Describe how to use a *digit* (in character form) as a subscript.

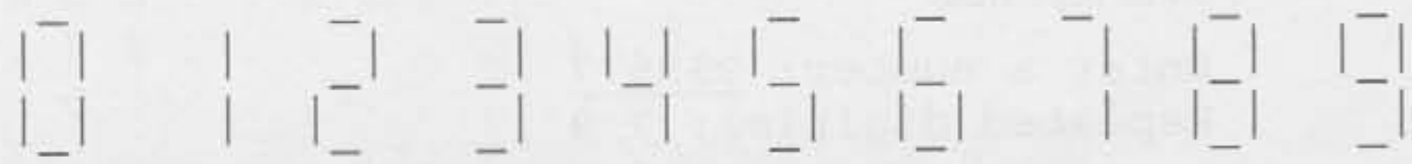
3. Write a declaration of an array named `weekend` containing seven `bool` values. Include an initializer that makes the first and last values `true`; all other values should be `false`.

4. (C99) Repeat Exercise 3, but this time use a designated initializer. Make the initializer as short as possible.

5. The Fibonacci numbers are 0, 1, 1, 2, 3, 5, 8, 13, ..., where each number is the sum of the two preceding numbers. Write a program fragment that declares an array named `fib_numbers` of length 40 and fills the array with the first 40 Fibonacci numbers. *Hint:* Fill in the first two numbers individually, then use a loop to compute the remaining numbers.

Section 8.2

6. Calculators, watches, and other electronic devices often rely on seven-segment displays for numerical output. To form a digit, such devices "turn on" some of the seven segments while leaving others "off":



Suppose that we want to set up an array that remembers which segments should be "on" for each digit. Let's number the segments as follows:

Here's what the array might look like, with each row representing one digit:

```
const int segments[10][7] = {{1, 1, 1, 1, 1, 1, 0}, ...};
```

I've given you the first row of the initializer; fill in the rest.

W 7. Using the shortcuts described in Section 8.2, shrink the initializer for the `segments` array (Exercise 6) as much as you can.

8. Write a declaration for a two-dimensional array named `temperature_readings` that stores one month of hourly temperature readings. (For simplicity, assume that a month has 30 days.) The rows of the array should represent days of the month; the columns should represent hours of the day.

9. Using the array of Exercise 8, write a program fragment that computes the average temperature for a month (averaged over all days of the month and all hours of the day).

10. Write a declaration for an 8 × 8 `char` array named `chess_board`. Include an initializer that puts the following data into the array (one character per array element):

```
r n b q k b n r
p p p p p p p p
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
P P P P P P P P
R N B Q K B N R
```

11. Write a program fragment that declares an 8 × 8 `char` array named `checker_board` and then uses a loop to store the following data into the array (one character per array element):

```
B R B R B R B R
R B R B R B R B
B R B R B R B R
R B R B R B R B
B R B R B R B R
R B R B R B R B
B R B R B R B R
R B R B R B R B
```

Hint: The element in row *i*, column *j*, should be the letter `B` if $i + j$ is an even number.

Programming Projects

1. Modify the `repdigit.c` program of Section 8.1 so that it shows which digits (if any) were repeated:

```
Enter a number: 939577
Repeated digit(s): 7 9
```

W 2. Modify the `repdigit.c` program of Section 8.1 so that it prints a table showing how many times each digit appears in the number:

```
Enter a number: 41271092
Digit:        0  1  2  3  4  5  6  7  8  9
Occurrences:  1  2  2  0  1  0  0  1  0  1
```

3. Modify the `repdigit.c` program of Section 8.1 so that the user can enter more than one number to be tested for repeated digits. The program should terminate when the user enters a number that's less than or equal to 0.

4. Modify the `reverse.c` program of Section 8.1 to use the expression `(int) (sizeof(a) / sizeof(a[0]))` (or a macro with this value) for the array length.

W 5. Modify the `interest.c` program of Section 8.1 so that it compounds interest *monthly* instead of *annually*. The form of the output shouldn't change; the balance should still be shown at annual intervals.

6. The prototypical Internet newbie is a fellow named B1FF, who has a unique way of writing messages. Here's a typical B1FF communiqué:

```
H3Y DUD3, C 15 R1LLY C00L!!!!!!!!!!
```

Write a "B1FF filter" that reads a message entered by the user and translates it into B1FF-speak:

```
Enter message: Hey dude, C is rilly cool
In B1FF-speak: H3Y DUD3, C 15 R1LLY C00L!!!!!!!!!!
```

Your program should convert the message to upper-case letters, substitute digits for certain letters (A→4, B→8, E→3, I→1, O→0, S→5), and then append 10 or so exclamation marks. *Hint:* Store the original message in an array of characters, then go back through the array, translating and printing characters one by one.

7. Write a program that reads a 5×5 array of integers and then prints the row sums and the column sums:

```
Enter row 1: 8 3 9 0 10
Enter row 2: 3 5 17 1 1
Enter row 3: 2 8 6 23 1
Enter row 4: 15 7 3 2 9
Enter row 5: 6 14 2 6 0

Row totals: 30 27 40 36 28
Column totals: 34 37 37 32 21
```

W 8. Modify Programming Project 7 so that it prompts for five quiz grades for each of five students, then computes the total score and average score for each *student*, and the average score, high score, and low score for each *quiz*.

9. Write a program that generates a "random walk" across a 10×10 array. The array will contain characters (all `'.'` initially). The program must randomly "walk" from element to element, always going up, down, left, or right by one element. The elements visited by the program will be labeled with the letters `A` through `Z`, in the order visited. Here's an example of the desired output:

```
A . . . . . . . . .
B C D . . . . . . .
. F E . . . . . . .
H G . . . . . . . .
I . . . . . . . . .
J . . . . . . . Z .
K . . R S T U V Y .
L M P Q . . . W X .
. N O . . . . . . .
. . . . . . . . . .
```

Hint: Use the `srand` and `rand` functions (see `deal.c`) to generate random numbers. After generating a number, look at its remainder when divided by 4. There are four possible values for the remainder—0, 1, 2, and 3—indicating the direction of the next move. Before performing a move, check that (a) it won't go outside the array, and (b) it doesn't take us to

an element that already has a letter assigned. If either condition is violated, try moving in another direction. If all four directions are blocked, the program must terminate. Here's an example of premature termination:

```
A B G H I . . . . .
. C F . J K . . . .
. D E . M L . . . .
. . . . N O . . . .
. . W X Y P Q . . .
. . V U T S R . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
```

`Y` is blocked on all four sides, so there's no place to put `Z`.

10. Modify Programming Project 8 from Chapter 5 so that the departure times are stored in an array and the arrival times are stored in a second array. (The times are integers, representing the number of minutes since midnight.) The program will use a loop to search the array of departure times for the one closest to the time entered by the user.

11. Modify Programming Project 4 from Chapter 7 so that the program labels its output:

```
Enter phone number: 1-800-COL-LECT
In numeric form: 1-800-265-5328
```

The program will need to store the phone number (either in its original form or in its numeric form) in an array of characters until it can be printed. You may assume that the phone number is no more than 15 characters long.

12. Modify Programming Project 5 from Chapter 7 so that the SCRABBLE values of the letters are stored in an array. The array will have 26 elements, corresponding to the 26 letters of the alphabet. For example, element 0 of the array will store 1 (because the SCRABBLE value of the letter A is 1), element 1 of the array will store 3 (because the SCRABBLE value of the letter B is 3), and so forth. As each character of the input word is read, the program will use the array to determine the SCRABBLE value of that character. Use an array initializer to set up the array.

13. Modify Programming Project 11 from Chapter 7 so that the program labels its output:

```
Enter a first and last name: Lloyd Fosdick
You entered the name: Fosdick, L.
```

The program will need to store the last name (but not the first name) in an array of characters until it can be printed. You may assume that the last name is no more than 20 characters long.

14. Write a program that reverses the words in a sentence:

```
Enter a sentence: you can cage a swallow can't you?
Reversal of sentence: you can't swallow a cage can you?
```

Hint: Use a loop to read the characters one by one and store them in a one-dimensional `char` array. Have the loop stop at a period, question mark, or exclamation point (the "terminating character"), which is saved in a separate `char` variable. Then use a second loop to search backward through the array for the beginning of the last word. Print the last word, then search backward for the next-to-last word. Repeat until the beginning of the array is reached. Finally, print the terminating character.

15. One of the oldest known encryption techniques is the Caesar cipher, attributed to Julius Caesar. It involves replacing each letter in a message with another letter that is a fixed number of

positions later in the alphabet. (If the replacement would go past the letter *Z*, the cipher "wraps around" to the beginning of the alphabet. For example, if each letter is replaced by the letter two positions after it, then *Y* would be replaced by *A*, and *Z* would be replaced by *B*.) Write a program that encrypts a message using a Caesar cipher. The user will enter the message to be encrypted and the shift amount (the number of positions by which letters should be shifted):

```
Enter message to be encrypted: Go ahead, make my day.
Enter shift amount (1-25): 3
Encrypted message: Jr dkhdg, pdnh pb gdb.
```

Notice that the program can decrypt a message if the user enters 26 minus the original key:

```
Enter message to be encrypted: Jr dkhdg, pdnh pb gdb.
Enter shift amount (1-25): 23
Encrypted message: Go ahead, make my day.
```

You may assume that the message does not exceed 80 characters. Characters other than letters should be left unchanged. Lower-case letters remain lower-case when encrypted, and upper-case letters remain upper-case. *Hint:* To handle the wrap-around problem, use the expression `((ch - 'A') + n) % 26 + 'A'` to calculate the encrypted version of an upper-case letter, where `ch` stores the letter and `n` stores the shift amount. (You'll need a similar expression for lower-case letters.)

16. Write a program that tests whether two words are anagrams (permutations of the same letters):

```
Enter first word: smartest
Enter second word: mattress
The words are anagrams.

Enter first word: dumbest
Enter second word: stumble
The words are not anagrams.
```

Write a loop that reads the first word, character by character, using an array of 26 integers to keep track of how many times each letter has been seen. (For example, after the word *smartest* has been read, the array should contain the values 1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 1 2 2 0 0 0 0 0 0 0, reflecting the fact that *smartest* contains one *a*, one *e*, one *m*, one *r*, two *s*'s and two *t*'s.) Use another loop to read the second word, except this time decrementing the corresponding array element as each letter is read. Both loops should ignore any characters that aren't letters, and both should treat upper-case letters in the same way as lower-case letters. After the second word has been read, use a third loop to check whether all the elements in the array are zero. If so, the words are anagrams. *Hint:* You may wish to use functions from `<ctype.h>`, such as `isalpha` and `tolower`.

17. Write a program that prints an $n \times n$ magic square (a square arrangement of the numbers $1, 2, \ldots, n^2$ in which the sums of the rows, columns, and diagonals are all the same). The user will specify the value of n:

```
This program creates a magic square of a specified size.
The size must be an odd number between 1 and 99.
Enter size of magic square: 5
  17  24   1   8  15
  23   5   7  14  16
   4   6  13  20  22
  10  12  19  21   3
  11  18  25   2   9
```

Store the magic square in a two-dimensional array. Start by placing the number 1 in the middle of row 0. Place each of the remaining numbers 2, 3, ..., n^2 by moving up one row and over one column. Any attempt to go outside the bounds of the array should "wrap around" to the opposite side of the array. For example, instead of storing the next number in row –1, we would store it in row $n - 1$ (the last row). Instead of storing the next number in column n, we would store it in column 0. If a particular array element is already occupied, put the number directly below the previously stored number. If your compiler supports variable-length arrays, declare the array to have n rows and n columns. If not, declare the array to have 99 rows and 99 columns.

9 Functions

If you have a procedure with ten parameters, you probably missed some.

We saw in Chapter 2 that a function is simply a series of statements that have been grouped together and given a name. Although the term "function" comes from mathematics, C functions don't always resemble math functions. In C, a function doesn't necessarily have arguments, nor does it necessarily compute a value. (In some programming languages, a "function" returns a value, whereas a "procedure" doesn't. C lacks this distinction.)

Functions are the building blocks of C programs. Each function is essentially a small program, with its own declarations and statements. Using functions, we can divide a program into small pieces that are easier for us—and others—to understand and modify. Functions can take some of the tedium out of programming by allowing us to avoid duplicating code that's used more than once. Moreover, functions are reusable: we can take a function that was originally part of one program and use it in others.

Our programs so far have consisted of just the `main` function. In this chapter, we'll see how to write functions other than `main`, and we'll learn more about `main` itself. Section 9.1 shows how to define and call functions. Section 9.2 then discusses function declarations and how they differ from function definitions. Next, Section 9.3 examines how arguments are passed to functions. The remainder of the chapter covers the `return` statement (Section 9.4), the related issue of program termination (Section 9.5), and recursion (Section 9.6).

9.1 Defining and Calling Functions

Before we go over the formal rules for defining a function, let's look at three simple programs that define functions.

PROGRAM Computing Averages

Suppose we often need to compute the average of two `double` values. The C library doesn't have an "average" function, but we can easily define our own. Here's what it would look like:

```
double average(double a, double b)
{
  return (a + b) / 2;
}
```

Q&A

The word `double` at the beginning is `average`'s ***return type:*** the type of data that the function returns each time it's called. The identifiers `a` and `b` (the function's ***parameters***) represent the two numbers that will be supplied when `average` is called. Each parameter must have a type (just like every variable has a type); in this example, both `a` and `b` have type `double`. (It may look odd, but the word `double` must appear twice, once for `a` and once for `b`.) A function parameter is essentially a variable whose initial value will be supplied later, when the function is called.

Every function has an executable part, called the ***body***, which is enclosed in braces. The body of `average` consists of a single `return` statement. Executing this statement causes the function to "return" to the place from which it was called; the value of `(a + b) / 2` will be the value returned by the function.

To call a function, we write the function name, followed by a list of ***arguments***. For example, `average(x, y)` is a call of the `average` function. Arguments are used to supply information to a function; in this case, `average` needs to know which two numbers to average. The effect of the call `average(x, y)` is to copy the values of `x` and `y` into the parameters `a` and `b`, and then execute the body of `average`. An argument doesn't have to be a variable; any expression of a compatible type will do, allowing us to write `average(5.1, 8.9)` or `average(x/2, y/3)`.

We'll put the call of `average` in the place where we need to use the return value. For example, we could write

```
printf("Average: %g\n", average(x, y));
```

to compute the average of `x` and `y` and then print it. This statement has the following effect:

1. The `average` function is called with `x` and `y` as arguments.
2. `x` and `y` are copied into `a` and `b`.
3. `average` executes its `return` statement, returning the average of `a` and `b`.
4. `printf` prints the value that `average` returns. (The return value of `average` becomes one of `printf`'s arguments.)

Note that the return value of `average` isn't saved anywhere; the program prints it and then discards it. If we had needed the return value later in the program, we could have captured it in a variable:

```
avg = average(x, y);
```

This statement calls `average`, then saves its return value in the variable `avg`.

Now, let's use the `average` function in a complete program. The following program reads three numbers and computes their averages, one pair at a time:

```
Enter three numbers: 3.5 9.6 10.2
Average of 3.5 and 9.6: 6.55
Average of 9.6 and 10.2: 9.9
Average of 3.5 and 10.2: 6.85
```

Among other things, this program shows that a function can be called as often as we need.

average.c

```
/* Computes pairwise averages of three numbers */

#include <stdio.h>

double average(double a, double b)
{
  return (a + b) / 2;
}

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}
```

Notice that I've put the definition of `average` before `main`. We'll see in Section 9.2 that putting `average` after `main` causes problems.

PROGRAM

Printing a Countdown

Not every function returns a value. For example, a function whose job is to produce output may not need to return anything. To indicate that a function has no return value, we specify that its return type is `void`. (`void` is a type with no values.) Consider the following function, which prints the message `T minus` *n* `and counting`, where *n* is supplied when the function is called:

```
void print_count(int n)
{
  printf("T minus %d and counting\n", n);
}
```

`print_count` has one parameter, `n`, of type `int`. It returns nothing, so I've specified `void` as the return type and omitted the `return` statement. Since `print_count` doesn't return a value, we can't call it in the same way we call `average`. Instead, a call of `print_count` must appear in a statement by itself:

```
print_count(i);
```

Here's a program that calls `print_count` 10 times inside a loop:

countdown.c

```
/* Prints a countdown */

#include <stdio.h>

void print_count(int n)
{
  printf("T minus %d and counting\n", n);
}

int main(void)
{
  int i;

  for (i = 10; i > 0; --i)
    print_count(i);

  return 0;
}
```

Initially, `i` has the value 10. When `print_count` is called for the first time, `i` is copied into `n`, so that `n` takes on the value 10 as well. As a result, the first call of `print_count` will print

```
T minus 10 and counting
```

`print_count` then returns to the point at which it was called, which happens to be the body of a `for` statement. The `for` statement resumes where it left off, decrementing `i` to 9 and testing whether it's greater than 0. It is, so `print_count` is called again, this time printing

```
T minus 9 and counting
```

Each time `print_count` is called, `i` is different, so `print_count` will print 10 different messages.

PROGRAM Printing a Pun (Revisited)

Some functions have no parameters at all. Consider `print_pun`, which prints a bad pun each time it's called:

```
void print_pun(void)
{
  printf("To C, or not to C: that is the question.\n");
}
```

The word `void` in parentheses indicates that `print_pun` has no arguments. (This time, we're using `void` as a placeholder that means "nothing goes here.")

To call a function with no arguments, we write the function's name, followed by parentheses:

```
print_pun();
```

The parentheses *must* be present, even though there are no arguments.

Here's a tiny program that tests the `print_pun` function:

pun2.c

```
/* Prints a bad pun */

#include <stdio.h>

void print_pun(void)
{
  printf("To C, or not to C: that is the question.\n");
}

int main(void)
{
  print_pun();
  return 0;
}
```

The execution of this program begins with the first statement in `main`, which happens to be a call of `print_pun`. When `print_pun` begins to execute, it in turn calls `printf` to display a string. When `printf` returns, `print_pun` returns to `main`.

Function Definitions

Now that we've seen several examples, let's look at the general form of a ***function definition:***

function definition

return-type function-name (*parameters*)
{
  declarations
  statements
}

The return type of a function is the type of value that the function returns. The following rules govern the return type:

- Functions may not return arrays, but there are no other restrictions on the return type.
- Specifying that the return type is `void` indicates that the function doesn't return a value.

- If the return type is omitted in C89, the function is presumed to return a value of type `int`. In C99, it's illegal to omit the return type of a function.

C99

As a matter of style, some programmers put the return type *above* the function name:

```
double
average(double a, double b)
{
  return (a + b) / 2;
}
```

Putting the return type on a separate line is especially useful if the return type is lengthy, like `unsigned long int`.

After the function name comes a list of parameters. Each parameter is preceded by a specification of its type; parameters are separated by commas. If the function has no parameters, the word `void` should appear between the parentheses. *Note:* A separate type must be specified for each parameter, even when several parameters have the same type:

Q&A

```
double average(double a, b)    /*** WRONG ***/
{
  return (a + b) / 2;
}
```

The body of a function may include both declarations and statements. For example, the `average` function could be written

```
double average(double a, double b)
{
  double sum;          /* declaration */

  sum = a + b;         /* statement */
  return sum / 2;      /* statement */
}
```

Variables declared in the body of a function belong exclusively to that function; they can't be examined or modified by other functions. In C89, variable declarations must come first, before all statements in the body of a function. In C99, variable declarations and statements can be mixed, as long as each variable is declared prior to the first statement that uses the variable. (Some pre-C99 compilers also allow mixing of declarations and statements.)

C99

The body of a function whose return type is `void` (which I'll call a "`void` function") can be empty:

```
void print_pun(void)
{
}
```

Leaving the body empty may make sense during program development; we can leave room for the function without taking the time to complete it, then come back later and write the body.

Function Calls

A function call consists of a function name followed by a list of arguments, enclosed in parentheses:

```
average(x, y)
print_count(i)
print_pun()
```

If the parentheses are missing, the function won't get called:

```
print_pun;    /*** WRONG ***/
```

Q&A The result is a legal (albeit meaningless) expression statement that looks correct, but has no effect. Some compilers issue a warning such as *"statement with no effect."*

A call of a `void` function is always followed by a semicolon to turn it into a statement:

```
print_count(i);
print_pun();
```

A call of a non-`void` function, on the other hand, produces a value that can be stored in a variable, tested, printed, or used in some other way:

```
avg = average(x, y);
if (average(x, y) > 0)
  printf("Average is positive\n");
printf("The average is %g\n", average(x, y));
```

The value returned by a non-`void` function can always be discarded if it's not needed:

```
average(x, y);    /* discards return value */
```

expression statements ➤4.5

This call of `average` is an example of an expression statement: a statement that evaluates an expression but then discards the result.

Ignoring the return value of `average` is an odd thing to do, but for some functions it makes sense. The `printf` function, for example, returns the number of characters that it prints. After the following call, `num_chars` will have the value 9:

```
num_chars = printf("Hi, Mom!\n");
```

Since we're probably not interested in the number of characters printed, we'll normally discard `printf`'s return value:

```
printf("Hi, Mom!\n");    /* discards return value */
```

To make it clear that we're deliberately discarding the return value of a function, C allows us to put `(void)` before the call:

```
(void) printf("Hi, Mom!\n");
```

casting ➤7.4 What we're doing is casting (converting) the return value of `printf` to type `void`. (In C, "casting to `void`" is a polite way of saying "throwing away.") Using `(void)` makes it clear to others that you deliberately discarded the return value, not just forgot that there was one. Unfortunately, there are a great many functions in the C library whose values are routinely ignored; using `(void)` when calling them all can get tiresome, so I haven't done so in this book.

PROGRAM Testing Whether a Number Is Prime

To see how functions can make programs easier to understand, let's write a program that tests whether a number is prime. The program will prompt the user to enter a number, then respond with a message indicating whether or not the number is prime:

```
Enter a number: 34
Not prime
```

Instead of putting the prime-testing details in `main`, we'll define a separate function that returns `true` if its parameter is a prime number and `false` if it isn't. When given a number `n`, the `is_prime` function will divide `n` by each of the numbers between 2 and the square root of `n`; if the remainder is ever 0, we know that `n` isn't prime.

prime.c

```
/* Tests whether a number is prime */

#include <stdbool.h>   /* C99 only */
#include <stdio.h>

bool is_prime(int n)
{
  int divisor;

  if (n <= 1)
    return false;
  for (divisor = 2; divisor * divisor <= n; divisor++)
    if (n % divisor == 0)
      return false;
  return true;
}

int main(void)
{
  int n;

  printf("Enter a number: ");
  scanf("%d", &n);
  if (is_prime(n))
    printf("Prime\n");
  else
    printf("Not prime\n");
```

```
  return 0;
}
```

Notice that `main` contains a variable named `n` even though `is_prime`'s parameter is also named `n`. In general, a function may declare a variable with the same name as a variable in another function. The two variables represent different locations in memory, so assigning a new value to one variable doesn't change the other. (This property extends to parameters as well.) Section 10.1 discusses this point in more detail.

As `is_prime` demonstrates, a function may have more than one `return` statement. However, we can execute just one of these statements during a given call of the function, because reaching a `return` statement causes the function to return to where it was called. We'll learn more about the `return` statement in Section 9.4.

9.2 Function Declarations

In the programs in Section 9.1, the definition of each function was always placed *above* the point at which it was called. In fact, C doesn't require that the definition of a function precede its calls. Suppose that we rearrange the `average.c` program by putting the definition of `average` *after* the definition of `main`:

```
#include <stdio.h>

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}

double average(double a, double b)
{
  return (a + b) / 2;
}
```

When the compiler encounters the first call of `average` in `main`, it has no information about `average`: it doesn't know how many parameters `average` has, what the types of these parameters are, or what kind of value `average` returns. Instead of producing an error message, though, the compiler assumes that `average` returns an `int` value (recall from Section 9.1 that the return type of a

function is `int` by default). We say that the compiler has created an ***implicit declaration*** of the function. The compiler is unable to check that we're passing `average` the right number of arguments and that the arguments have the proper type. Instead, it performs the default argument promotions and hopes for the best. When it encounters the definition of `average` later in the program, the compiler notices that the function's return type is actually `double`, not `int`, and so we get an error message.

default argument promotions ➤9.3

One way to avoid the problem of call-before-definition is to arrange the program so that the definition of each function precedes all its calls. Unfortunately, such an arrangement doesn't always exist, and even when it does, it may make the program harder to understand by putting its function definitions in an unnatural order.

Fortunately, C offers a better solution: declare each function before calling it. A ***function declaration*** provides the compiler with a brief glimpse at a function whose full definition will appear later. A function declaration resembles the first line of a function definition with a semicolon added at the end:

function declaration

return-type function-name (*parameters*) ;

Needless to say, the declaration of a function must be consistent with the function's definition.

Q&A

Here's how our program would look with a declaration of `average` added:

```
#include <stdio.h>

double average(double a, double b);   /* DECLARATION */

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}

double average(double a, double b)     /* DEFINITION */
{
  return (a + b) / 2;
}
```

Function declarations of the kind we've been discussing are known as ***function prototypes*** to distinguish them from an older style of function declaration in which the parentheses are left empty. A prototype provides a complete description

Q&A

of how to call a function: how many arguments to supply, what their types should be, and what type of result will be returned.

Incidentally, a function prototype doesn't have to specify the *names* of the function's parameters, as long as their *types* are present:

```
double average(double, double);
```

It's usually best not to omit parameter names, since they help document the purpose of each parameter and remind the programmer of the order in which arguQ&A ments must appear when the function is called. However, there are legitimate reasons for omitting parameter names, and some programmers prefer to do so.

C99 C99 has adopted the rule that either a declaration or a definition of a function must be present prior to any call of the function. Calling a function for which the compiler has not yet seen a declaration or definition is an error.

9.3 Arguments

Let's review the difference between a parameter and an argument. *Parameters* appear in function *definitions;* they're dummy names that represent values to be supplied when the function is called. *Arguments* are expressions that appear in function *calls*. When the distinction between *argument* and *parameter* isn't important, I'll sometimes use *argument* to mean either.

In C, arguments are ***passed by value:*** when a function is called, each argument is evaluated and its value assigned to the corresponding parameter. Since the parameter contains a copy of the argument's value, any changes made to the parameter during the execution of the function don't affect the argument. In effect, each parameter behaves like a variable that's been initialized to the value of the matching argument.

The fact that arguments are passed by value has both advantages and disadvantages. Since a parameter can be modified without affecting the corresponding argument, we can use parameters as variables within the function, thereby reducing the number of genuine variables needed. Consider the following function, which raises a number `x` to a power `n`:

```
int power(int x, int n)
{
  int i, result = 1;

  for (i = 1; i <= n; i++)
    result = result * x;

  return result;
}
```

Since `n` is a *copy* of the original exponent, we can modify it inside the function, thus removing the need for `i`:

```
int power(int x, int n)
{
  int result = 1;

  while (n-- > 0)
    result = result * x;

  return result;
}
```

Unfortunately, C's requirement that arguments be passed by value makes it difficult to write certain kinds of functions. For example, suppose that we need a function that will decompose a `double` value into an integer part and a fractional part. Since a function can't *return* two numbers, we might try passing a pair of variables to the function and having it modify them:

```
void decompose(double x, long int_part, double frac_part)
{
  int_part = (long) x;    /* drops the fractional part of x */
  frac_part = x - int_part;
}
```

Suppose that we call the function in the following way:

```
decompose(3.14159, i, d);
```

At the beginning of the call, 3.14159 is copied into `x`, `i`'s value is copied into `int_part`, and `d`'s value is copied into `frac_part`. The statements inside `decompose` then assign 3 to `int_part` and .14159 to `frac_part`, and the function returns. Unfortunately, `i` and `d` weren't affected by the assignments to `int_part` and `frac_part`, so they have the same values after the call as they did before the call. With a little extra effort, `decompose` can be made to work, as we'll see in Section 11.4. However, we'll need to cover more of C's features first.

Argument Conversions

C allows function calls in which the types of the arguments don't match the types of the parameters. The rules governing how the arguments are converted depend on whether or not the compiler has seen a prototype for the function (or the function's full definition) prior to the call:

- ***The compiler has encountered a prototype prior to the call.*** The value of each argument is implicitly converted to the type of the corresponding parameter as if by assignment. For example, if an `int` argument is passed to a function that was expecting a `double`, the argument is converted to `double` automatically.
- ***The compiler has not encountered a prototype prior to the call.*** The compiler performs the ***default argument promotions:*** (1) `float` arguments are converted to `double`. (2) The integral promotions are performed, causing `char`

and `short` arguments to be converted to `int`. (In C99, the integer promotions are performed.)

C99

⚠

Relying on the default argument promotions is dangerous. Consider the following program:

```
#include <stdio.h>

int main(void)
{
  double x = 3.0;
  printf("Square: %d\n", square(x));

  return 0;
}

int square(int n)
{
  return n * n;
}
```

At the time `square` is called, the compiler hasn't seen a prototype yet, so it doesn't know that `square` expects an argument of type `int`. Instead, the compiler performs the default argument promotions on `x`, with no effect. Since it's expecting an argument of type `int` but has been given a `double` value instead, the effect of calling `square` is undefined. The problem can be fixed by casting `square`'s argument to the proper type:

```
printf("Square: %d\n", square((int) x));
```

Of course, a much better solution is to provide a prototype for `square` before calling it. In C99, calling `square` without first providing a declaration or definition of the function is an error.

C99

Array Arguments

Arrays are often used as arguments. When a function parameter is a one-dimensional array, the length of the array can be (and is normally) left unspecified:

Q&A

```
int f(int a[])   /* no length specified */
{
  ...
}
```

The argument can be any one-dimensional array whose elements are of the proper type. There's just one problem: how will `f` know how long the array is? Unfortunately, C doesn't provide any easy way for a function to determine the length of an array passed to it. Instead, we'll have to supply the length—if the function needs it—as an additional argument.

Although we can use the `sizeof` operator to help determine the length of an array *variable*, it doesn't give the correct answer for an array *parameter*:

```
int f(int a[])
{
  int len = sizeof(a) / sizeof(a[0]);
    /*** WRONG: not the number of elements in a ***/
  ...
}
```

Section 12.3 explains why.

The following function illustrates the use of one-dimensional array arguments. When given an array `a` of `int` values, `sum_array` returns the sum of the elements in `a`. Since `sum_array` needs to know the length of `a`, we must supply it as a second argument.

```
int sum_array(int a[], int n)
{
  int i, sum = 0;

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

The prototype for `sum_array` has the following appearance:

```
int sum_array(int a[], int n);
```

As usual, we can omit the parameter names if we wish:

```
int sum_array(int [], int);
```

When `sum_array` is called, the first argument will be the name of an array, and the second will be its length. For example:

```
#define LEN 100

int main(void)
{
  int b[LEN], total;
  ...
  total = sum_array(b, LEN);
  ...
}
```

Notice that we don't put brackets after an array name when passing it to a function:

```
total = sum_array(b[], LEN);    /*** WRONG ***/
```

An important point about array arguments: A function has no way to check that we've passed it the correct array length. We can exploit this fact by telling the function that the array is smaller than it really is. Suppose that we've only stored 50 numbers in the b array, even though it can hold 100. We can sum just the first 50 elements by writing

```
total = sum_array(b, 50);    /* sums first 50 elements */
```

sum_array will ignore the other 50 elements. (Indeed, it won't know that they even exist!)

Be careful not to tell a function that an array argument is *larger* than it really is:

```
total = sum_array(b, 150);    /*** WRONG ***/
```

In this example, sum_array will go past the end of the array, causing undefined behavior.

Another important thing to know is that a function is allowed to change the elements of an array parameter, and the change is reflected in the corresponding argument. For example, the following function modifies an array by storing zero into each of its elements:

```
void store_zeros(int a[], int n)
{
  int i;

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

The call

```
store_zeros(b, 100);
```

will store zero into the first 100 elements of the array b. This ability to modify the elements of an array argument may seem to contradict the fact that C passes arguments by value. In fact, there's no contradiction, but I won't be able to explain why until Section 12.3.

Q&A If a parameter is a multidimensional array, only the length of the first dimension may be omitted when the parameter is declared. For example, if we revise the sum_array function so that a is a two-dimensional array, we must specify the number of columns in a, although we don't have to indicate the number of rows:

```
#define LEN 10

int sum_two_dimensional_array(int a[][LEN], int n)
{
  int i, j, sum = 0;
```

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

  return sum;
}
```

Not being able to pass multidimensional arrays with an arbitrary number of columns can be a nuisance. Fortunately, we can often work around this difficulty by using arrays of pointers. C99's variable-length array parameters provide an even better solution to the problem.

arrays of pointers ➤13.7

C99 Variable-Length Array Parameters

C99 adds several new twists to array arguments. The first has to do with variable-length arrays (VLAs), a feature of C99 that allows the length of an array to be specified using a non-constant expression. Variable-length arrays can also be parameters, as it turns out.

variable-length arrays ➤8.3

Consider the `sum_array` function discussed earlier in this section. Here's the definition of `sum_array`, with the body omitted:

```
int sum_array(int a[], int n)
{
  ...
}
```

As it stands now, there's no direct link between `n` and the length of the array `a`. Although the function body treats `n` as `a`'s length, the actual length of the array could in fact be larger than `n` (or smaller, in which case the function won't work correctly).

Using a variable-length array parameter, we can explicitly state that `a`'s length is `n`:

```
int sum_array(int n, int a[n])
{
  ...
}
```

The value of the first parameter (`n`) specifies the length of the second parameter (`a`). Note that the order of the parameters has been switched; order is important when variable-length array parameters are used.

The following version of `sum_array` is illegal:

```
int sum_array(int a[n], int n)    /*** WRONG ***/
{
  ...
}
```

The compiler will issue an error message at `int a[n]`, because it hasn't yet seen `n`.

There are several ways to write the prototype for our new version of `sum_array`. One possibility is to make it look exactly like the function definition:

```
int sum_array(int n, int a[n]);    /* Version 1 */
```

Another possibility is to replace the array length by an asterisk (`*`):

```
int sum_array(int n, int a[*]);    /* Version 2a */
```

The reason for using the `*` notation is that parameter names are optional in function declarations. If the name of the first parameter is omitted, it wouldn't be possible to specify that the length of the array is `n`, but the `*` provides a clue that the length of the array is related to parameters that come earlier in the list:

```
int sum_array(int, int [*]);       /* Version 2b */
```

It's also legal to leave the brackets empty, as we normally do when declaring an array parameter:

```
int sum_array(int n, int a[]);     /* Version 3a */
int sum_array(int, int []);        /* Version 3b */
```

Leaving the brackets empty isn't a good choice, because it doesn't expose the relationship between `n` and `a`.

In general, the length of a variable-length array parameter can be any expression. For example, suppose that we were to write a function that concatenates two arrays `a` and `b` by copying the elements of `a`, followed by the elements of `b`, into a third array named `c`:

```
int concatenate(int m, int n, int a[m], int b[n], int c[m+n])
{
  ...
}
```

The length of `c` is the sum of the lengths of `a` and `b`. The expression used to specify the length of `c` involves two other parameters, but in general it could refer to variables outside the function or even call other functions.

Variable-length array parameters with a single dimension—as in all our examples so far—have limited usefulness. They make a function declaration or definition more descriptive by stating the desired length of an array argument. However, no additional error-checking is performed; it's still possible for an array argument to be too long or too short.

It turns out that variable-length array parameters are most useful for multidimensional arrays. Earlier in this section, we tried to write a function that sums the elements in a two-dimensional array. Our original function was limited to arrays with a fixed number of columns. If we use a variable-length array parameter, we can generalize the function to any number of columns:

```
int sum_two_dimensional_array(int n, int m, int a[n][m])
{
  int i, j, sum = 0;

  for (i = 0; i < n; i++)
    for (j = 0; j < m; j++)
      sum += a[i][j];

  return sum;
}
```

Prototypes for this function include the following:

```
int sum_two_dimensional_array(int n, int m, int a[n][m]);
int sum_two_dimensional_array(int n, int m, int a[*][*]);
int sum_two_dimensional_array(int n, int m, int a[][m]);
int sum_two_dimensional_array(int n, int m, int a[][*]);
```

C99 Using `static` in Array Parameter Declarations

C99 allows the use of the keyword `static` in the declaration of array parameters. (The keyword itself existed before C99. Section 18.2 discusses its traditional uses.)

In the following example, putting `static` in front of the number `3` indicates that the length of `a` is guaranteed to be at least 3:

```
int sum_array(int a[static 3], int n)
{
  ...
}
```

Using `static` in this way has no effect on the behavior of the program. The presence of `static` is merely a "hint" that may allow a C compiler to generate faster instructions for accessing the array. (If the compiler knows that an array will always have a certain minimum length, it can arrange to "prefetch" these elements from memory when the function is called, before the elements are actually needed by statements within the function.)

One last note about `static`: If an array parameter has more than one dimension, `static` can be used only in the first dimension (for example, when specifying the number of rows in a two-dimensional array).

C99 Compound Literals

Let's return to the original `sum_array` function one last time. When `sum_array` is called, the first argument is usually the name of an array (the one whose elements are to be summed). For example, we might call `sum_array` in the following way:

```
int b[] = {3, 0, 3, 4, 1};
total = sum_array(b, 5);
```

The only problem with this arrangement is that b must be declared as a variable and then initialized prior to the call. If b isn't needed for any other purpose, it can be mildly annoying to create it solely for the purpose of calling sum_array.

In C99, we can avoid this annoyance by using a ***compound literal:*** an unnamed array that's created "on the fly" by simply specifying which elements it contains. The following call of sum_array has a compound literal (shown in **bold**) as its first argument:

```
total = sum_array((int []){3, 0, 3, 4, 1}, 5);
```

In this example, the compound literal creates an array containing the five integers 3, 0, 3, 4, and 1. We didn't specify the length of the array, so it's determined by the number of elements in the literal. We also have the option of specifying a length explicitly: (int [4]){1, 9, 2, 1} is equivalent to (int []){1, 9, 2, 1}.

In general, a compound literal consists of a type name within parentheses, followed by a set of values enclosed by braces. A compound literal resembles a cast applied to an initializer. In fact, compound literals and initializers obey the same rules. A compound literal may contain designators, just like a designated initializer, and it may fail to provide full initialization (in which case any uninitialized elements default to zero). For example, the literal (int [10]){8, 6} has 10 elements; the first two have the values 8 and 6, and the remaining elements have the value 0.

designated initializers ➤8.1

Compound literals created inside a function may contain arbitrary expressions, not just constants. For example, we could write

```
total = sum_array((int []){2 * i, i + j, j * k}, 3);
```

where i, j, and k are variables. This aspect of compound literals greatly enhances their usefulness.

lvalues ➤4.2

A compound literal is an lvalue, so the values of its elements can be changed. If desired, a compound literal can be made "read-only" by adding the word const to its type, as in (const int []){5, 4}.

9.4 The return Statement

A non-void function must use the return statement to specify what value it will return. The return statement has the form

return statement

```
return expression ;
```

The expression is often just a constant or variable:

```
return 0;
return status;
```

conditional operator ➤5.2

More complex expressions are possible. For example, it's not unusual to see the conditional operator used in a return expression:

```
return n >= 0 ? n : 0;
```

When this statement is executed, the expression `n >= 0 ? n : 0` is evaluated first. The statement returns the value of `n` if it's not negative; otherwise, it returns 0.

If the type of the expression in a `return` statement doesn't match the function's return type, the expression will be implicitly converted to the return type. For example, if a function is declared to return an `int`, but the `return` statement contains a `double` expression, the value of the expression is converted to `int`.

`return` statements may appear in functions whose return type is `void`, provided that no expression is given:

```
return;   /* return in a void function */
```

Q&A

Putting an expression in such a `return` statement will get you a compile-time error. In the following example, the `return` statement causes the function to return immediately when given a negative argument:

```
void print_int(int i)
{
  if (i < 0)
    return;
  printf("%d", i);
}
```

If `i` is less than 0, `print_int` will return without calling `printf`.

A `return` statement may appear at the end of a `void` function:

```
void print_pun(void)
{
  printf("To C, or not to C: that is the question.\n");
  return;   /* OK, but not needed */
}
```

Using `return` is unnecessary, though, since the function will return automatically after its last statement has been executed.

If a non-`void` function reaches the end of its body—that is, it fails to execute a `return` statement—the behavior of the program is undefined if it attempts to use the value returned by the function. Some compilers will issue a warning such as *"control reaches end of non-void function"* if they detect the possibility of a non-`void` function "falling off" the end of its body.

9.5 Program Termination

Since `main` is a function, it must have a return type. Normally, the return type of `main` is `int`, which is why the programs we've seen so far have defined `main` in the following way:

```
int main(void)
{
  ...
}
```

Older C programs often omit `main`'s return type, taking advantage of the fact that it traditionally defaults to `int`:

```
main()
{
  ...
}
```

C99 Omitting the return type of a function isn't legal in C99, so it's best to avoid this practice. Omitting the word `void` in `main`'s parameter list remains legal, but—as a matter of style—it's best to be explicit about the fact that `main` has no parameters. (We'll see later that `main` sometimes *does* have two parameters, usually named `argc` and `argv`.)

`argc` and `argv` ➤*13.7*

Q&A The value returned by `main` is a status code that—in some operating systems—can be tested when the program terminates. `main` should return 0 if the program terminates normally; to indicate abnormal termination, `main` should return a value other than 0. (Actually, there's no rule to prevent us from using the return value for other purposes.) It's good practice to make sure that every C program returns a status code, even if there are no plans to use it, since someone running the program later may decide to test it.

The `exit` Function

Executing a `return` statement in `main` is one way to terminate a program. Another is calling the `exit` function, which belongs to `<stdlib.h>`. The argument passed to `exit` has the same meaning as `main`'s return value: both indicate the program's status at termination. To indicate normal termination, we'd pass `0`:

`<stdlib.h>` header ➤*26.2*

```
exit(0);                    /* normal termination */
```

Since `0` is a bit cryptic, C allows us to pass `EXIT_SUCCESS` instead (the effect is the same):

```
exit(EXIT_SUCCESS);    /* normal termination */
```

Passing `EXIT_FAILURE` indicates abnormal termination:

```
exit(EXIT_FAILURE);    /* abnormal termination */
```

`EXIT_SUCCESS` and `EXIT_FAILURE` are macros defined in `<stdlib.h>`. The values of `EXIT_SUCCESS` and `EXIT_FAILURE` are implementation-defined; typical values are 0 and 1, respectively.

As methods of terminating a program, `return` and `exit` are closely related. In fact, the statement

`return` *expression*`;`

in `main` is equivalent to

`exit(`*expression*`);`

The difference between `return` and `exit` is that `exit` causes program termination regardless of which function calls it. The `return` statement causes program termination only when it appears in the `main` function. Some programmers use `exit` exclusively to make it easier to locate all exit points in a program.

9.6 Recursion

A function is ***recursive*** if it calls itself. For example, the following function computes $n!$ recursively, using the formula $n! = n \times (n - 1)!$:

```
int fact(int n)
{
  if (n <= 1)
    return 1;
  else
    return n * fact(n - 1);
}
```

Some programming languages rely heavily on recursion, while others don't even allow it. C falls somewhere in the middle: it allows recursion, but most C programmers don't use it that often.

To see how recursion works, let's trace the execution of the statement

```
i = fact(3);
```

Here's what happens:

`fact(3)` finds that 3 is not less than or equal to 1, so it calls
  `fact(2)`, which finds that 2 is not less than or equal to 1, so it calls
    `fact(1)`, which finds that 1 *is* less than or equal to 1, so it returns 1, causing
  `fact(2)` to return $2 \times 1 = 2$, causing
`fact(3)` to return $3 \times 2 = 6$.

Notice how the unfinished calls of `fact` "pile up" until `fact` is finally passed 1. At that point, the old calls of `fact` begin to "unwind" one by one, until the original call—`fact(3)`—finally returns with the answer, 6.

Here's another example of recursion: a function that computes x^n, using the formula $x^n = x \times x^{n-1}$.

```
int power(int x, int n)
{
  if (n == 0)
    return 1;
  else
    return x * power(x, n - 1);
}
```

The call `power(5, 3)` would be executed as follows:

`power(5, 3)` finds that 3 is not equal to 0, so it calls
  `power(5, 2)`, which finds that 2 is not equal to 0, so it calls
    `power(5, 1)`, which finds that 1 is not equal to 0, so it calls
      `power(5, 0)`, which finds that 0 *is* equal to 0, so it returns 1, causing
    `power(5, 1)` to return $5 \times 1 = 5$, causing
  `power(5, 2)` to return $5 \times 5 = 25$, causing
`power(5, 3)` to return $5 \times 25 = 125$.

Incidentally, we can condense the `power` function a bit by putting a conditional expression in the `return` statement:

```
int power(int x, int n)
{
  return n == 0 ? 1 : x * power(x, n - 1);
}
```

Both `fact` and `power` are careful to test a "termination condition" as soon as they're called. When `fact` is called, it immediately checks whether its parameter is less than or equal to 1. When `power` is called, it first checks whether its second parameter is equal to 0. All recursive functions need some kind of termination condition in order to prevent infinite recursion.

The Quicksort Algorithm

At this point, you may wonder why we're bothering with recursion; after all, neither `fact` nor `power` really needs it. Well, you've got a point. Neither function makes much of a case for recursion, because each calls itself just once. Recursion is much more helpful for sophisticated algorithms that require a function to call itself two or more times.

In practice, recursion often arises naturally as a result of an algorithm design technique known as ***divide-and-conquer***, in which a large problem is divided into smaller pieces that are then tackled by the same algorithm. A classic example of the divide-and-conquer strategy can be found in the popular sorting algorithm known as ***Quicksort***. The Quicksort algorithm goes as follows (for simplicity, we'll assume that the array being sorted is indexed from 1 to n):

1. Choose an array element e (the "partitioning element"), then rearrange the array so that elements $1, \ldots, i-1$ are less than or equal to e, element i contains e, and elements $i+1, \ldots, n$ are greater than or equal to e.
2. Sort elements $1, \ldots, i-1$ by using Quicksort recursively.
3. Sort elements $i+1, \ldots, n$ by using Quicksort recursively.

After step 1, the element e is in its proper location. Since the elements to the left of e are all less than or equal to it, they'll be in their proper places once they've been sorted in step 2; similar reasoning applies to the elements to the right of e.

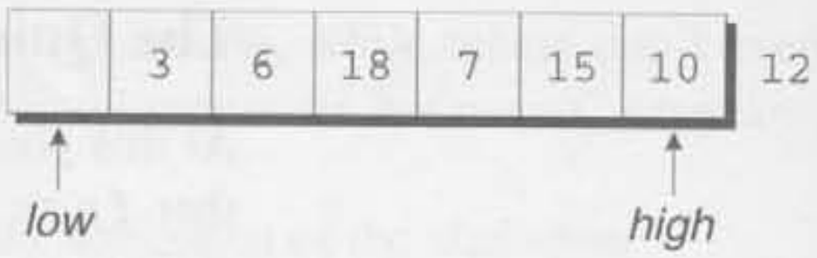
Step 1 of the Quicksort algorithm is obviously critical. There are various methods to partition an array, some much better than others. We'll use a technique

that's easy to understand but not particularly efficient. I'll first describe the partitioning algorithm informally; later, we'll translate it into C code.

The algorithm relies on two "markers" named *low* and *high*, which keep track of positions within the array. Initially, *low* points to the first element of the array and *high* points to the last element. We start by copying the first element (the partitioning element) into a temporary location elsewhere, leaving a "hole" in the array. Next, we move *high* across the array from right to left until it points to an element that's smaller than the partitioning element. We then copy the element into the hole that *low* points to, which creates a new hole (pointed to by *high*). We now move *low* from left to right, looking for an element that's larger than the partitioning element. When we find one, we copy it into the hole that *high* points to. The process repeats, with *low* and *high* taking turns, until they meet somewhere in the middle of the array. At that time, both will point to a hole; all we need do is copy the partitioning element into the hole. The following diagrams illustrate how Quicksort would sort an array of integers:

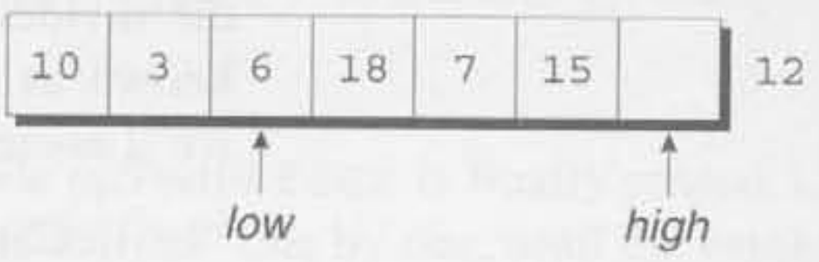
Let's start with an array containing seven elements. *low* points to the first element; *high* points to the last one.

The first element, 12, is the partitioning element. Copying it somewhere else leaves a hole at the beginning of the array.

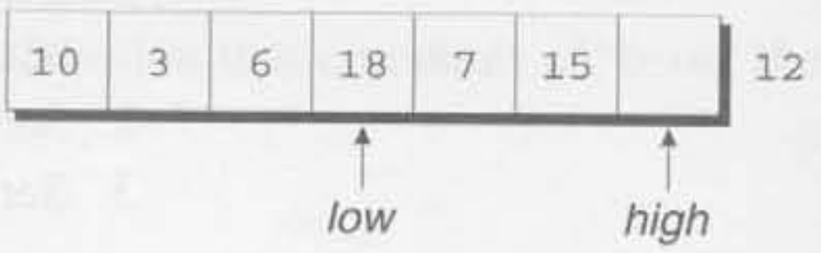


We now compare the element pointed to by *high* with 12. Since 10 is smaller than 12, it's on the wrong side of the array, so we move it to the hole and shift *low* to the right.

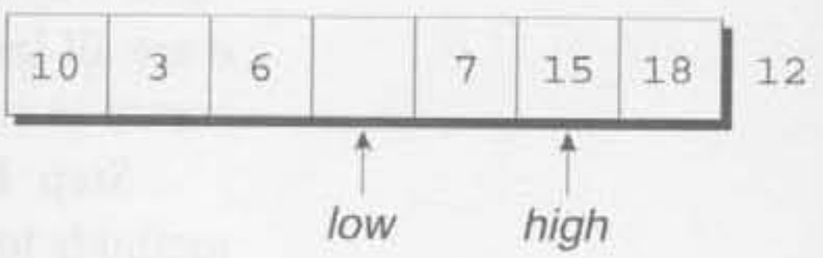
low points to the number 3, which is less than 12 and therefore doesn't need to be moved. We shift *low* to the right instead.



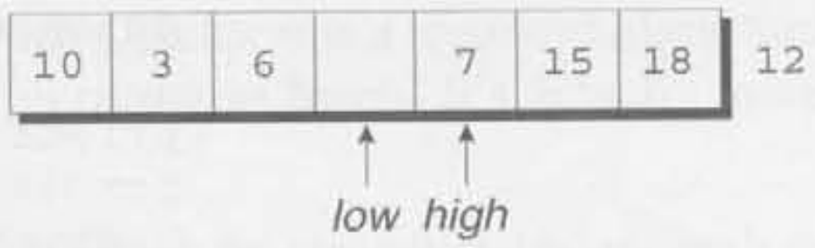
Since 6 is also less than 12, we shift *low* again.



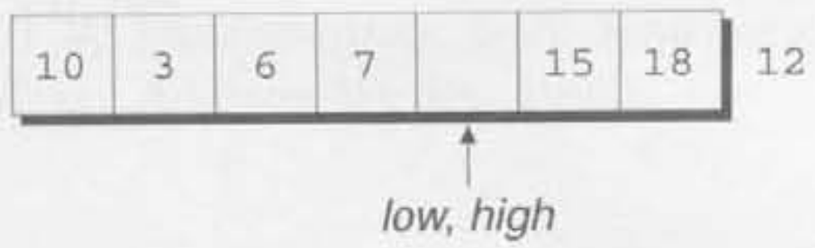
low now points to 18, which is larger than 12 and therefore out of position. After moving 18 to the hole, we shift *high* to the left.



high points to 15, which is greater than 12 and thus doesn't need to be moved. We shift *high* to the left and continue.



high points to 7, which is out of position. After moving 7 to the hole, we shift *low* to the right.



low and *high* are now equal, so we move the partitioning element to the hole.

At this point, we've accomplished our objective: all elements to the left of the partitioning element are less than or equal to 12, and all elements to the right are greater than or equal to 12. Now that the array has been partitioned, we can use Quicksort recursively to sort the first four elements of the array (10, 3, 6, and 7) and the last two (15 and 18).

PROGRAM Quicksort

Let's develop a recursive function named `quicksort` that uses the Quicksort algorithm to sort an array of integers. To test the function, we'll have `main` read 10 numbers into an array, call `quicksort` to sort the array, then print the elements in the array:

```
Enter 10 numbers to be sorted: 9 16 47 82 4 66 12 3 25 51
In sorted order: 3 4 9 12 16 25 47 51 66 82
```

Since the code for partitioning the array is a bit lengthy, I'll put it in a separate function named `split`.

qsort.c

```
/* Sorts an array of integers using Quicksort algorithm */

#include <stdio.h>

#define N 10

void quicksort(int a[], int low, int high);
int split(int a[], int low, int high);

int main(void)
{
  int a[N], i;

  printf("Enter %d numbers to be sorted: ", N);
  for (i = 0; i < N; i++)
    scanf("%d", &a[i]);
```

```
  quicksort(a, 0, N - 1);

  printf("In sorted order: ");
  for (i = 0; i < N; i++)
    printf("%d ", a[i]);
  printf("\n");

  return 0;
}

void quicksort(int a[], int low, int high)
{
  int middle;

  if (low >= high) return;
  middle = split(a, low, high);
  quicksort(a, low, middle - 1);
  quicksort(a, middle + 1, high);
}

int split(int a[], int low, int high)
{
  int part_element = a[low];

  for (;;) {
    while (low < high && part_element <= a[high])
      high--;
    if (low >= high) break;
    a[low++] = a[high];

    while (low < high && a[low] <= part_element)
      low++;
    if (low >= high) break;
    a[high--] = a[low];
  }

  a[high] = part_element;
  return high;
}
```

Although this version of Quicksort works, it's not the best. There are numerous ways to improve the program's performance, including:

- ***Improving the partitioning algorithm.*** Our method isn't the most efficient. Instead of choosing the first element in the array as the partitioning element, it's better to take the median of the first element, the middle element, and the last element. The partitioning process itself can also be sped up. In particular, it's possible to avoid the `low < high` tests in the two `while` loops.
- ***Using a different method to sort small arrays.*** Instead of using Quicksort recursively all the way down to arrays with one element, it's better to use a simpler method for small arrays (those with fewer than, say, 25 elements).

- ***Making Quicksort nonrecursive.*** Although Quicksort is a recursive algorithm by nature—and is easiest to understand in recursive form—it's actually more efficient if the recursion is removed.

For details about improving Quicksort, consult a book on algorithm design, such as Robert Sedgewick's *Algorithms in C, Parts 1–4: Fundamentals, Data Structures, Sorting, Searching,* Third Edition (Boston, Mass.: Addison-Wesley, 1998).

Q & A

Q: Some C books appear to use terms other than *parameter* and *argument*. Is there any standard terminology? [p. 184]

A: As with many other aspects of C, there's no general agreement on terminology, although the C89 and C99 standards use *parameter* and *argument*. The following table should help you translate:

This book:	*Other books:*
parameter	formal argument, formal parameter
argument	actual argument, actual parameter

Keep in mind that—when no confusion would result—I sometimes deliberately blur the distinction between the two terms, using *argument* to mean either.

Q: I've seen programs in which parameter types are specified in separate declarations after the parameter list, as in the following example:

```
double average(a, b)
double a, b;
{
  return (a + b) / 2;
}
```

Is this practice legal? [p. 188]

A: This method of defining functions comes from K&R C, so you may encounter it in older books and programs. C89 and C99 support this style so that older programs will still compile. I'd avoid using it in new programs, however, for a couple of reasons.

First, functions that are defined in the older way aren't subject to the same degree of error-checking. When a function is defined in the older way—and no prototype is present—the compiler won't check that the function is called with the right number of arguments, nor will it check that the arguments have the proper types. Instead, it will perform the default argument promotions.

default argument promotions ➤9.3

Second, the C standard says that the older style is "obsolescent," meaning that its use is discouraged and that it may be dropped from C eventually.

Q: Some programming languages allow procedures and functions to be nested within each other. Does C allow function definitions to be nested?

A: No. C does not permit the definition of one function to appear in the body of another. Among other things, this restriction simplifies the compiler.

***Q: Why does the compiler allow the use of function names that aren't followed by parentheses? [p. 189]**

A: We'll see in a later chapter that the compiler treats a function name not followed by parentheses as a *pointer* to the function. Pointers to functions have legitimate uses, so the compiler can't automatically assume that a function name without parentheses is an error. The statement

pointers to functions ➤17.7

```
print_pun;
```

is legal because the compiler treats `print_pun` as a pointer and therefore an expression, making this a valid (although pointless) expression statement.

expression statements ➤4.5

***Q: In the function call `f(a, b)`, how does the compiler know whether the comma is punctuation or whether it's an operator?**

A: It turns out that the arguments in a function call can't be arbitrary expressions. Instead, they must be "assignment expressions," which can't contain commas used as operators unless they're enclosed in parentheses. In other words, in the call `f(a, b)` the comma is punctuation; in the call `f((a, b))` it's an operator.

Q: Do the names of parameters in a function prototype have to match the names given later in the function's definition? [p. 192]

A: No. Some programmers take advantage of this fact by giving long names to parameters in the prototype, then using shorter names in the actual definition. Or a French-speaking programmer might use English names in prototypes, then switch to more familiar French names in function definitions.

Q: I still don't understand why we bother with function prototypes. If we just put definitions of all the functions before `main`, we're covered, right?

A: Wrong. First, you're assuming that only `main` calls the other functions, which is unrealistic. In practice, some of the functions will call each other. If we put all function definitions above `main`, we'll have to watch their order carefully. Calling a function that hasn't been defined yet can lead to big problems.

But that's not all. Suppose that two functions call each other (which isn't as far-fetched as it may sound). No matter which function we define first, it will end up calling a function that hasn't been defined yet.

But there's still more! Once programs reach a certain size, it won't be feasible to put all the functions in one file anymore. When we reach that point, we'll need prototypes to tell the compiler about functions in other files.

Q: I've seen function declarations that omit all information about parameters:

```
double average();
```

Is this practice legal? [p. 192]

A: Yes. This declaration informs the compiler that `average` returns a `double` value but provides no information about the number and types of its parameters. (Leaving the parentheses empty doesn't necessarily mean that `average` has no parameters.)

In K&R C, this form of function declaration is the only one allowed; the form that we've been using—the function prototype, in which parameter information *is* included—was introduced in C89. The older kind of function declaration is now obsolescent, although still allowed.

Q: Why would a programmer deliberately omit parameter names in a function prototype? Isn't it easier to just leave the names? [p. 193]

A: Omitting parameter names in prototypes is typically done for defensive purposes. If a macro happens to have the same name as a parameter, the parameter name will be replaced during preprocessing, thereby damaging the prototype in which it appears. This isn't likely to be a problem in a small program written by one person but can occur in large applications written by many people.

Q: Is it legal to put a function declaration inside the body of another function?

A: Yes. Here's an example:

```
int main(void)
{
  double average(double a, double b);
  …
}
```

This declaration of `average` is valid only for the body of `main`; if other functions need to call `average`, they'll each have to declare it.

The advantage of this practice is that it's clearer to the reader which functions call which other functions. (In this example, we see that `main` will be calling `average`.) On the other hand, it can be a nuisance if several functions need to call the same function. Even worse, trying to add and remove declarations during program maintenance can be a real pain. For these reasons, I'll always put function declarations outside function bodies.

Q: If several functions have the same return type, can their declarations be combined? For example, since both `print_pun` and `print_count` have `void` as their return type, is the following declaration legal?

```
void print_pun(void), print_count(int n);
```

A: Yes. In fact, C even allows us to combine function declarations with variable declarations:

```
double x, y, average(double a, double b);
```

Combining declarations in this way usually isn't a good idea, though; it can easily cause confusion.

Q: What happens if I specify a length for a one-dimensional array parameter? [p. 195]

A: The compiler ignores it. Consider the following example:

```
double inner_product(double v[3], double w[3]);
```

Other than documenting that `inner_product`'s arguments are supposed to be arrays of length 3, specifying a length doesn't buy us much. The compiler won't check that the arguments actually have length 3, so there's no added security. In fact, the practice is misleading in that it suggests that `inner_product` can only be passed arrays of length 3, when in fact we can pass arrays of arbitrary length.

***Q: Why can the first dimension in an array parameter be left unspecified, but not the other dimensions? [p. 197]**

A: First, we need to discuss how arrays are passed in C. As Section 12.3 explains, when an array is passed to a function, the function is given a *pointer* to the first element in the array.

Next, we need to know how the subscripting operator works. Suppose that `a` is a one-dimensional array passed to a function. When we write

```
a[i] = 0;
```

the compiler generates instructions that compute the address of `a[i]` by multiplying `i` by the size of an array element and adding the result to the address that `a` represents (the pointer passed to the function). This calculation doesn't depend on the length of `a`, which explains why we can omit it when defining the function.

What about multidimensional arrays? Recall that C stores arrays in row-major order, with the elements in row 0 stored first, then the elements in row 1, and so forth. Suppose that `a` is a two-dimensional array parameter and we write

```
a[i][j] = 0;
```

The compiler generates instructions to do the following: (1) multiply `i` by the size of a single row of `a`; (2) add this result to the address that `a` represents; (3) multiply `j` by the size of an array element; and (4) add this result to the address computed in step 2. To generate these instructions, the compiler must know the size of a row in the array, which is determined by the number of columns. The bottom line: the programmer must declare the number of columns in `a`.

Q: Why do some programmers put parentheses around the expression in a `return` statement?

A: The examples in the first edition of Kernighan and Ritchie's *The C Programming Language* always have parentheses in `return` statements, even though they aren't required. Programmers (and authors of subsequent books) picked up the habit from K&R. I don't use these parentheses, since they're unnecessary and

contribute nothing to readability. (Kernighan and Ritchie apparently agree: the `return` statements in the second edition of *The C Programming Language* lack parentheses.)

Q: What happens if a non-`void` function attempts to execute a `return` statement that has no expression? [p. 202]

A: That depends on the version of C. In C89, executing a `return` statement without an expression in a non-`void` function causes undefined behavior (but only if the program attempts to use the value returned by the function). In C99, such a statement is illegal and should be detected as an error by the compiler.

Q: How can I test `main`'s return value to see if a program has terminated normally? [p. 203]

A: That depends on your operating system. Many operating systems allow this value to be tested within a "batch file" or "shell script" that contains commands to run several programs. For example, the line

```
if errorlevel 1 command
```

in a Windows batch file will execute *command* if the last program terminated with a status code greater than or equal to 1.

In UNIX, each shell has its own method for testing the status code. In the Bourne shell, the variable `$?` contains the status of the last program run. The C shell has a similar variable, but its name is `$status`.

Q: Why does my compiler produce a *"control reaches end of non-void function"* warning when it compiles `main`?

A: The compiler has noticed that `main`, despite having `int` as its return type, doesn't have a `return` statement. Putting the statement

```
return 0;
```

at the end of `main` will keep the compiler happy. Incidentally, this is good practice even if your compiler doesn't object to the lack of a `return` statement.

C99 When a program is compiled using a C99 compiler, this warning shouldn't occur. In C99, it's OK to "fall off" the end of `main` without returning a value; the standard states that `main` automatically returns 0 in this situation.

Q: With regard to the previous question: Why not just define `main`'s return type to be `void`?

A: Although this practice is fairly common, it's illegal according to the C89 standard. Even if it weren't illegal, it wouldn't be a good idea, since it presumes that no one will ever test the program's status upon termination.

C99 C99 opens the door to legalizing this practice, by allowing `main` to be declared "in some other implementation-defined manner" (with a return type other than `int` or parameters other than those specified by the standard). However, any such usage isn't portable, so it's best to declare `main`'s return type to be `int`.

Q: Is it legal for a function `f1` to call a function `f2`, which then calls `f1`?

A: Yes. This is just an indirect form of recursion in which one call of `f1` leads to another. (But make sure that either `f1` or `f2` eventually terminates!)

Exercises

Section 9.1

1. The following function, which computes the area of a triangle, contains two errors. Locate the errors and show how to fix them. (*Hint:* There are no errors in the formula.)

```
double triangle_area(double base, height)
double product;
{
  product = base * height;
  return product / 2;
}
```

W 2. Write a function `check(x, y, n)` that returns 1 if both `x` and `y` fall between 0 and `n` – 1, inclusive. The function should return 0 otherwise. Assume that `x`, `y`, and `n` are all of type `int`.

3. Write a function `gcd(m, n)` that calculates the greatest common divisor of the integers `m` and `n`. (Programming Project 2 in Chapter 6 describes Euclid's algorithm for computing the GCD.)

W 4. Write a function `day_of_year(month, day, year)` that returns the day of the year (an integer between 1 and 366) specified by the three arguments.

5. Write a function `num_digits(n)` that returns the number of digits in `n` (a positive integer). *Hint:* To determine the number of digits in a number *n*, divide it by 10 repeatedly. When *n* reaches 0, the number of divisions indicates how many digits *n* originally had.

W 6. Write a function `digit(n, k)` that returns the k^{th} digit (from the right) in `n` (a positive integer). For example, `digit(829, 1)` returns 9, `digit(829, 2)` returns 2, and `digit(829, 3)` returns 8. If `k` is greater than the number of digits in `n`, have the function return 0.

7. Suppose that the function `f` has the following definition:

```
int f(int a, int b) { ... }
```

Which of the following statements are legal? (Assume that `i` has type `int` and `x` has type `double`.)

```
(a) i = f(83, 12);
(b) x = f(83, 12);
(c) i = f(3.15, 9.28);
(d) x = f(3.15, 9.28);
(e) f(83, 12);
```

Section 9.2

W 8. Which of the following would be valid prototypes for a function that returns nothing and has one `double` parameter?

```
(a) void f(double x);
```

(b) `void f(double);`
(c) `void f(x);`
(d) `f(double x);`

Section 9.3

*9. What will be the output of the following program?

```
#include <stdio.h>

void swap(int a, int b);

int main(void)
{
  int i = 1, j = 2;

  swap(i, j);
  printf("i = %d, j = %d\n", i, j);
  return 0;
}

void swap(int a, int b)
{
  int temp = a;
  a = b;
  b = temp;
}
```

W 10. Write functions that return the following values. (Assume that `a` and `n` are parameters, where `a` is an array of `int` values and `n` is the length of the array.)
(a) The largest element in `a`.
(b) The average of all elements in `a`.
(c) The number of positive elements in `a`.

11. Write the following function:

```
float compute_GPA(char grades[], int n);
```

The `grades` array will contain letter grades (`A`, `B`, `C`, `D`, or `F`, either upper-case or lower-case); `n` is the length of the array. The function should return the average of the grades (assume that `A` = 4, `B` = 3, `C` = 2, `D` = 1, and `F` = 0).

12. Write the following function:

```
double inner_product(double a[], double b[], int n);
```

The function should return `a[0] * b[0] + a[1] * b[1] + ... + a[n-1] * b[n-1]`.

13. Write the following function, which evaluates a chess position:

```
int evaluate_position(char board[8][8]);
```

`board` represents a configuration of pieces on a chessboard, where the letters `K`, `Q`, `R`, `B`, `N`, `P` represent White pieces, and the letters `k`, `q`, `r`, `b`, `n`, and `p` represent Black pieces. `evaluate_position` should sum the values of the White pieces (`Q` = 9, `R` = 5, `B` = 3, `N` = 3, `P` = 1). It should also sum the values of the Black pieces (done in a similar way). The function will return the difference between the two numbers. This value will be positive if White has an advantage in material and negative if Black has an advantage.

Section 9.4

14. The following function is supposed to return `true` if any element of the array `a` has the value 0 and `false` if all elements are nonzero. Sadly, it contains an error. Find the error and show how to fix it:

```
bool has_zero(int a[], int n)
{
  int i;

  for (i = 0; i < n; i++)
    if (a[i] == 0)
      return true;
    else
      return false;
}
```

W 15. The following (rather confusing) function finds the median of three numbers. Rewrite the function so that it has just one `return` statement.

```
double median(double x, double y, double z)
{
  if (x <= y)
    if (y <= z) return y;
    else if (x <= z) return z;
    else return x;
  if (z <= y) return y;
  if (x <= z) return x;
  return z;
}
```

Section 9.6

16. Condense the `fact` function in the same way we condensed `power`.

W 17. Rewrite the `fact` function so that it's no longer recursive.

18. Write a recursive version of the `gcd` function (see Exercise 3). Here's the strategy to use for computing `gcd(m, n)`: If `n` is 0, return `m`. Otherwise, call `gcd` recursively, passing `n` as the first argument and `m % n` as the second.

W *19. Consider the following "mystery" function:

```
void pb(int n)
{
  if (n != 0) {
    pb(n / 2);
    putchar('0' + n % 2);
  }
}
```

Trace the execution of the function by hand. Then write a program that calls the function, passing it a number entered by the user. What does the function do?

Programming Projects

1. Write a program that asks the user to enter a series of integers (which it stores in an array), then sorts the integers by calling the function `selection_sort`. When given an array with n elements, `selection_sort` must do the following:

   1. Search the array to find the largest element, then move it to the last position in the array.
   2. Call itself recursively to sort the first $n - 1$ elements of the array.

2. Modify Programming Project 5 from Chapter 5 so that it uses a function to compute the amount of income tax. When passed an amount of taxable income, the function will return the tax due.

3. Modify Programming Project 9 from Chapter 8 so that it includes the following functions:

```
void generate_random_walk(char walk[10][10]);
void print_array(char walk[10][10]);
```

`main` first calls `generate_random_walk`, which initializes the array to contain `'.'` characters and then replaces some of these characters by the letters `A` through `Z`, as described in the original project. `main` then calls `print_array` to display the array on the screen.

4. Modify Programming Project 16 from Chapter 8 so that it includes the following functions:

```
void read_word(int counts[26]);
bool equal_array(int counts1[26], int counts2[26]);
```

`main` will call `read_word` twice, once for each of the two words entered by the user. As it reads a word, `read_word` will use the letters in the word to update the `counts` array, as described in the original project. (`main` will declare two arrays, one for each word. These arrays are used to track how many times each letter occurs in the words.) `main` will then call `equal_array`, passing it the two arrays. `equal_array` will return `true` if the elements in the two arrays are identical (indicating that the words are anagrams) and `false` otherwise.

5. Modify Programming Project 17 from Chapter 8 so that it includes the following functions:

```
void create_magic_square(int n, char magic_square[n][n]);
void print_magic_square(int n, char magic_square[n][n]);
```

After obtaining the number n from the user, `main` will call `create_magic_square`, passing it an $n \times n$ array that is declared inside `main`. `create_magic_square` will fill the array with the numbers 1, 2, ..., n^2 as described in the original project. `main` will then call `print_magic_square`, which will display the array in the format described in the original project. *Note:* If your compiler doesn't support variable-length arrays, declare the array in `main` to be 99×99 instead of $n \times n$ and use the following prototypes instead:

```
void create_magic_square(int n, char magic_square[99][99]);
void print_magic_square(int n, char magic_square[99][99]);
```

6. Write a function that computes the value of the following polynomial:

$3x^5 + 2x^4 - 5x^3 - x^2 + 7x - 6$

Write a program that asks the user to enter a value for x, calls the function to compute the value of the polynomial, and then displays the value returned by the function.

7. The `power` function of Section 9.6 can be made faster by having it calculate x^n in a different way. We first notice that if n is a power of 2, then x^n can be computed by squaring. For example, x^4 is the square of x^2, so x^4 can be computed using only two multiplications instead of three. As it happens, this technique can be used even when n is not a power of 2. If n is even, we use the formula $x^n = (x^{n/2})^2$. If n is odd, then $x^n = x \times x^{n-1}$. Write a recursive function that computes x^n. (The recursion ends when $n = 0$, in which case the function returns 1.) To test your function, write a program that asks the user to enter values for x and n, calls `power` to compute x^n, and then displays the value returned by the function.

8. Write a program that simulates the game of craps, which is played with two dice. On the first roll, the player wins if the sum of the dice is 7 or 11. The player loses if the sum is 2, 3,

or 12. Any other roll is called the "point" and the game continues. On each subsequent roll, the player wins if he or she rolls the point again. The player loses by rolling 7. Any other roll is ignored and the game continues. At the end of each game, the program will ask the user whether or not to play again. When the user enters a response other than `y` or `Y`, the program will display the number of wins and losses and then terminate.

```
You rolled: 8
Your point is 8
You rolled: 3
You rolled: 10
You rolled: 8
You win!

Play again? y

You rolled: 6
Your point is 6
You rolled: 5
You rolled: 12
You rolled: 3
You rolled: 7
You lose!

Play again? y

You rolled: 11
You win!

Play again? n

Wins: 2  Losses: 1
```

Write your program as three functions: `main`, `roll_dice`, and `play_game`. Here are the prototypes for the latter two functions:

```
int roll_dice(void);
bool play_game(void);
```

`roll_dice` should generate two random numbers, each between 1 and 6, and return their sum. `play_game` should play one craps game (calling `roll_dice` to determine the outcome of each dice roll); it will return `true` if the player wins and `false` if the player loses. `play_game` is also responsible for displaying messages showing the results of the player's dice rolls. `main` will call `play_game` repeatedly, keeping track of the number of wins and losses and displaying the "you win" and "you lose" messages. *Hint:* Use the `rand` function to generate random numbers. See the `deal.c` program in Section 8.2 for an example of how to call `rand` and the related `srand` function.

10 Program Organization

As Will Rogers would have said, "There is no such thing as a free variable."

Having covered functions in Chapter 9, we're ready to confront several issues that arise when a program contains more than one function. The chapter begins with a discussion of the differences between local variables (Section 10.1) and external variables (Section 10.2). Section 10.3 then considers blocks (compound statements containing declarations). Section 10.4 tackles the scope rules that apply to local names, external names, and names declared in blocks. Finally, Section 10.5 suggests a way to organize function prototypes, function definitions, variable declarations, and the other parts of a C program.

10.1 Local Variables

A variable declared in the body of a function is said to be ***local*** to the function. In the following function, `sum` is a local variable:

```
int sum_digits(int n)
{
  int sum = 0;    /* local variable */

  while (n > 0) {
    sum += n % 10;
    n /= 10;
  }

  return sum;
}
```

By default, local variables have the following properties:

- ***Automatic storage duration.*** The ***storage duration*** (or ***extent***) of a variable is the portion of program execution during which storage for the variable exists. Storage for a local variable is "automatically" allocated when the enclosing function is called and deallocated when the function returns, so the variable is said to have ***automatic storage duration.*** A local variable doesn't retain its value when its enclosing function returns. When the function is called again, there's no guarantee that the variable will still have its old value.
- ***Block scope.*** The ***scope*** of a variable is the portion of the program text in which the variable can be referenced. A local variable has ***block scope:*** it is visible from its point of declaration to the end of the enclosing function body. Since the scope of a local variable doesn't extend beyond the function to which it belongs, other functions can use the same name for other purposes.

Section 18.2 covers these and other related concepts in more detail.

C99 Since C99 doesn't require variable declarations to come at the beginning of a function, it's possible for a local variable to have a very small scope. In the following example, the scope of `i` doesn't begin until the line on which it's declared, which could be near the end of the function body:

```
void f(void)
{
  ...
  int i;
  ...
}
```

scope of `i`

Static Local Variables

Putting the word `static` in the declaration of a local variable causes it to have ***static storage duration*** instead of automatic storage duration. A variable with static storage duration has a permanent storage location, so it retains its value throughout the execution of the program. Consider the following function:

```
void f(void)
{
  static int i;    /* static local variable */
  ...
}
```

Since the local variable `i` has been declared `static`, it occupies the same memory location throughout the execution of the program. When `f` returns, `i` won't

Q&A lose its value.

A static local variable still has block scope, so it's not visible to other functions. In a nutshell, a static variable is a place to hide data from other functions but retain it for future calls of the same function.

Parameters

Parameters have the same properties—automatic storage duration and block scope—as local variables. In fact, the only real difference between parameters and local variables is that each parameter is initialized automatically when a function is called (by being assigned the value of the corresponding argument).

10.2 External Variables

Passing arguments is one way to transmit information to a function. Functions can also communicate through ***external variables***—variables that are declared outside the body of any function.

The properties of external variables (or ***global variables,*** as they're sometimes called) are different from those of local variables:

- ***Static storage duration.*** External variables have static storage duration, just like local variables that have been declared `static`. A value stored in an external variable will stay there indefinitely.
- ***File scope.*** An external variable has ***file scope:*** it is visible from its point of declaration to the end of the enclosing file. As a result, an external variable can be accessed (and potentially modified) by all functions that follow its declaration.

Example: Using External Variables to Implement a Stack

To illustrate how external variables might be used, let's look at a data structure known as a ***stack***. (Stacks are an abstract concept, not a C feature; they can be implemented in most programming languages.) A stack, like an array, can store multiple data items of the same type. However, the operations on a stack are limited: we can either ***push*** an item onto the stack (add it to one end—the "stack top") or ***pop*** it from the stack (remove it from the same end). Examining or modifying an item that's not at the top of the stack is forbidden.

One way to implement a stack in C is to store its items in an array, which we'll call `contents`. A separate integer variable named `top` marks the position of the stack top. When the stack is empty, `top` has the value 0. To push an item on the stack, we simply store the item in `contents` at the position indicated by `top`, then increment `top`. Popping an item requires decrementing `top`, then using it as an index into `contents` to fetch the item that's being popped.

Based on this outline, here's a program fragment (not a complete program) that declares the `contents` and `top` variables for a stack and provides a set of functions that represent operations on the stack. All five functions need access to the `top` variable, and two functions need access to `contents`, so we'll make `contents` and `top` external.

```
#include <stdbool.h>   /* C99 only */

#define STACK_SIZE 100

/* external variables */
int contents[STACK_SIZE];
int top = 0;

void make_empty(void)
{
  top = 0;
}

bool is_empty(void)
{
  return top == 0;
}

bool is_full(void)
{
  return top == STACK_SIZE;
}

void push(int i)
{
  if (is_full())
    stack_overflow();
  else
    contents[top++] = i;
}

int pop(void)
{
  if (is_empty())
    stack_underflow();
  else
    return contents[--top];
}
```

Pros and Cons of External Variables

External variables are convenient when many functions must share a variable or when a few functions share a large number of variables. In most cases, however, it's better for functions to communicate through parameters rather than by sharing variables. Here's why:

- If we change an external variable during program maintenance (by altering its type, say), we'll need to check every function in the same file to see how the change affects it.

- If an external variable is assigned an incorrect value, it may be difficult to identify the guilty function. It's like trying to solve a murder committed at a crowded party—there's no easy way to narrow the list of suspects.
- Functions that rely on external variables are hard to reuse in other programs. A function that depends on external variables isn't self-contained; to reuse the function, we'll have to drag along any external variables that it needs.

Many C programmers rely far too much on external variables. One common abuse: using the same external variable for different purposes in different functions. Suppose that several functions need a variable named `i` to control a `for` statement. Instead of declaring `i` in each function that uses it, some programmers declare it at the top of the program, thereby making the variable visible to all functions. This practice is poor not only for the reasons listed earlier, but also because it's misleading; someone reading the program later may think that the uses of the variable are related, when in fact they're not.

When you use external variables, make sure they have meaningful names. (Local variables don't always need meaningful names: it's often hard to think of a better name than `i` for the control variable in a `for` loop.) If you find yourself using names like `i` and `temp` for external variables, that's a clue that perhaps they should really be local variables.

Making variables external when they should be local can lead to some rather frustrating bugs. Consider the following example, which is supposed to display a 10 × 10 arrangement of asterisks:

```
int i;

void print_one_row(void)
{
  for (i = 1; i <= 10; i++)
    printf("*");
}

void print_all_rows(void)
{
  for (i = 1; i <= 10; i++) {
    print_one_row();
    printf("\n");
  }
}
```

Instead of printing 10 rows, `print_all_rows` prints only one row. When `print_one_row` returns after being called the first time, `i` will have the value 11. The `for` statement in `print_all_rows` then increments `i` and tests whether it's less than or equal to 10. It's not, so the loop terminates and the function returns.

PROGRAM

Guessing a Number

To get more experience with external variables, we'll write a simple game-playing program. The program generates a random number between 1 and 100, which the user attempts to guess in as few tries as possible. Here's what the user will see when the program is run:

```
Guess the secret number between 1 and 100.

A new number has been chosen.
Enter guess: 55
Too low; try again.
Enter guess: 65
Too high; try again.
Enter guess: 60
Too high; try again.
Enter guess: 58
You won in 4 guesses!

Play again? (Y/N) y

A new number has been chosen.
Enter guess: 78
Too high; try again.
Enter guess: 34
You won in 2 guesses!

Play again? (Y/N) n
```

This program will need to carry out several different tasks: initializing the random number generator, choosing a secret number, and interacting with the user until the correct number is picked. If we write a separate function to handle each task, we might end up with the following program.

guess.c

```
/* Asks user to guess a hidden number */

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

#define MAX_NUMBER 100

/* external variable */
int secret_number;

/* prototypes */
void initialize_number_generator(void);
void choose_new_secret_number(void);
void read_guesses(void);

int main(void)
{
  char command;
```

```
  printf("Guess the secret number between 1 and %d.\n\n",
         MAX_NUMBER);
  initialize_number_generator();
  do {
    choose_new_secret_number();
    printf("A new number has been chosen.\n");
    read_guesses();
    printf("Play again? (Y/N) ");
    scanf(" %c", &command);
    printf("\n");
  } while (command == 'y' || command == 'Y');

  return 0;
}

/**********************************************************
 * initialize_number_generator: Initializes the random    *
 *                              number generator using    *
 *                              the time of day.          *
 **********************************************************/
void initialize_number_generator(void)
{
  srand((unsigned) time(NULL));
}

/**********************************************************
 * choose_new_secret_number: Randomly selects a number    *
 *                           between 1 and MAX_NUMBER and *
 *                           stores it in secret_number.  *
 **********************************************************/
void choose_new_secret_number(void)
{
  secret_number = rand() % MAX_NUMBER + 1;
}

/**********************************************************
 * read_guesses: Repeatedly reads user guesses and tells  *
 *               the user whether each guess is too low,  *
 *               too high, or correct. When the guess is  *
 *               correct, prints the total number of      *
 *               guesses and returns.                     *
 **********************************************************/
void read_guesses(void)
{
  int guess, num_guesses = 0;

  for (;;) {
    num_guesses++;
    printf("Enter guess: ");
    scanf("%d", &guess);
    if (guess == secret_number) {
      printf("You won in %d guesses!\n\n", num_guesses);
      return;
    } else if (guess < secret_number)
```

```
      printf("Too low; try again.\n");
    else
      printf("Too high; try again.\n");
  }
}
```

time function ➤26.3
srand function ➤26.2
rand function ➤26.2

For random number generation, the `guess.c` program relies on the `time`, `srand`, and `rand` functions, which we first used in `deal.c` (Section 8.2). This time, we're scaling the return value of `rand` so that it falls between 1 and `MAX_NUMBER`.

Although `guess.c` works fine, it relies on an external variable. We made `secret_number` external so that both `choose_new_secret_number` and `read_guesses` could access it. If we alter `choose_new_secret_number` and `read_guesses` just a little, we should be able to move `secret_number` into the `main` function. We'll modify `choose_new_secret_number` so that it returns the new number, and we'll rewrite `read_guesses` so that `secret_number` can be passed to it as an argument.

Here's our new program, with changes in **bold**:

guess2.c

```
/* Asks user to guess a hidden number */

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

#define MAX_NUMBER 100

/* prototypes */
void initialize_number_generator(void);
int new_secret_number(void);
void read_guesses(int secret_number);

int main(void)
{
  char command;
  int secret_number;

  printf("Guess the secret number between 1 and %d.\n\n",
         MAX_NUMBER);
  initialize_number_generator();
  do {
    secret_number = new_secret_number();
    printf("A new number has been chosen.\n");
    read_guesses(secret_number);
    printf("Play again? (Y/N) ");
    scanf(" %c", &command);
    printf("\n");
  } while (command == 'y' || command == 'Y');

  return 0;
}
```

```
/*************************************************************
 * initialize_number_generator: Initializes the random       *
 *                              number generator using       *
 *                              the time of day.             *
 *************************************************************/
void initialize_number_generator(void)
{
  srand((unsigned) time(NULL));
}

/*************************************************************
 * new_secret_number: Returns a randomly chosen number       *
 *                    between 1 and MAX_NUMBER.              *
 *************************************************************/
int new_secret_number(void)
{
  return rand() % MAX_NUMBER + 1;
}

/*************************************************************
 * read_guesses: Repeatedly reads user guesses and tells     *
 *               the user whether each guess is too low,     *
 *               too high, or correct. When the guess is     *
 *               correct, prints the total number of         *
 *               guesses and returns.                        *
 *************************************************************/
void read_guesses(int secret_number)
{
  int guess, num_guesses = 0;

  for (;;) {
    num_guesses++;
    printf("Enter guess: ");
    scanf("%d", &guess);
    if (guess == secret_number) {
      printf("You won in %d guesses!\n\n", num_guesses);
      return;
    } else if (guess < secret_number)
      printf("Too low; try again.\n");
    else
      printf("Too high; try again.\n");
  }
}
```

10.3 Blocks

In Section 5.2, we encountered compound statements of the form

{ *statements* }

It turns out that C allows compound statements to contain declarations as well:

block { *declarations statements* }

I'll use the term ***block*** to describe such a compound statement. Here's an example of a block:

```
if (i > j) {
  /* swap values of i and j */
  int temp = i;
  i = j;
  j = temp;
}
```

By default, the storage duration of a variable declared in a block is automatic: storage for the variable is allocated when the block is entered and deallocated when the block is exited. The variable has block scope; it can't be referenced outside the block. A variable that belongs to a block can be declared `static` to give it static storage duration.

The body of a function is a block. Blocks are also useful inside a function body when we need variables for temporary use. In our last example, we needed a variable temporarily so that we could swap the values of `i` and `j`. Putting temporary variables in blocks has two advantages: (1) It avoids cluttering the declarations at the beginning of the function body with variables that are used only briefly. (2) It reduces name conflicts. In our example, the name `temp` can be used elsewhere in the same function for different purposes—the `temp` variable is strictly local to the block in which it's declared.

C99 C99 allows variables to be declared anywhere within a block, just as it allows variables to be declared anywhere within a function.

10.4 Scope

In a C program, the same identifier may have several different meanings. C's scope rules enable the programmer (and the compiler) to determine which meaning is relevant at a given point in the program.

Here's the most important scope rule: When a declaration inside a block names an identifier that's already visible (because it has file scope or because it's declared in an enclosing block), the new declaration temporarily "hides" the old one, and the identifier takes on a new meaning. At the end of the block, the identifier regains its old meaning.

Consider the (somewhat extreme) example at the top of the next page, in which the identifier `i` has four different meanings:

- In Declaration 1, `i` is a variable with static storage duration and file scope.

```
int i;                  /* Declaration 1 */

void f(int i)           /* Declaration 2 */
{
  i = 1;
}

void g(void)
{
  int i = 2;            /* Declaration 3 */
  if (i > 0) {
    int i;              /* Declaration 4 */
    i = 3;
  }
  i = 4;
}

void h(void)
{
  i = 5;
}
```

- In Declaration 2, `i` is a parameter with block scope.
- In Declaration 3, `i` is an automatic variable with block scope.
- In Declaration 4, `i` is also automatic and has block scope.

`i` is used five times. C's scope rules allow us to determine the meaning of `i` in each case:

- The `i = 1` assignment refers to the parameter in Declaration 2, not the variable in Declaration 1, since Declaration 2 hides Declaration 1.
- The `i > 0` test refers to the variable in Declaration 3, since Declaration 3 hides Declaration 1 and Declaration 2 is out of scope.
- The `i = 3` assignment refers to the variable in Declaration 4, which hides Declaration 3.
- The `i = 4` assignment refers to the variable in Declaration 3. It can't refer to Declaration 4, which is out of scope.
- The `i = 5` assignment refers to the variable in Declaration 1.

10.5 Organizing a C Program

Now that we've seen the major elements that make up a C program, it's time to develop a strategy for their arrangement. For now, we'll assume that a program

always fits into a single file. Chapter 15 shows how to organize a program that's split over several files.

So far, we've seen that a program may contain the following:

Preprocessing directives such as `#include` and `#define`
Type definitions
Declarations of external variables
Function prototypes
Function definitions

C imposes only a few rules on the order of these items: A preprocessing directive doesn't take effect until the line on which it appears. A type name can't be used until it's been defined. A variable can't be used until it's declared. Although C isn't as picky about functions, I strongly recommend that every function be defined or declared prior to its first call. (C99 makes this a requirement anyway.)

C99

There are several ways to organize a program so that these rules are obeyed. Here's one possible ordering:

`#include` directives
`#define` directives
Type definitions
Declarations of external variables
Prototypes for functions other than `main`
Definition of `main`
Definitions of other functions

It makes sense to put `#include` directives first, since they bring in information that will likely be needed in several places within the program. `#define` directives create macros, which are generally used throughout the program. Putting type definitions above the declarations of external variables is logical, since the declarations of these variables may refer to the type names just defined. Declaring external variables next makes them available to all the functions that follow. Declaring all functions except for `main` avoids the problems that arise when a function is called before the compiler has seen its prototype. This practice also makes it possible to arrange the function definitions in any order whatsoever: alphabetically by function name or with related functions grouped together, for example. Defining `main` before the other functions makes it easier for a reader to locate the program's starting point.

A final suggestion: Precede each function definition by a boxed comment that gives the name of the function, explains its purpose, discusses the meaning of each parameter, describes its return value (if any), and lists any side effects it has (such as modifying external variables).

PROGRAM Classifying a Poker Hand

To show how a C program might be organized, let's attempt a program that's a little more complex than our previous examples. The program will read and classify

a poker hand. Each card in the hand will have both a *suit* (clubs, diamonds, hearts, or spades) and a *rank* (two, three, four, five, six, seven, eight, nine, ten, jack, queen, king, or ace). We won't allow the use of jokers, and we'll assume that aces are high. The program will read a hand of five cards, then classify the hand into one of the following categories (listed in order from best to worst):

straight flush (both a straight and a flush)
four-of-a-kind (four cards of the same rank)
full house (a three-of-a-kind and a pair)
flush (five cards of the same suit)
straight (five cards with consecutive ranks)
three-of-a-kind (three cards of the same rank)
two pairs
pair (two cards of the same rank)
high card (any other hand)

If a hand falls into two or more categories, the program will choose the best one.

For input purposes, we'll abbreviate ranks and suits as follows (letters may be either upper- or lower-case):

Ranks: `2 3 4 5 6 7 8 9 t j q k a`
Suits: `c d h s`

If the user enters an illegal card or tries to enter the same card twice, the program will ignore the card, issue an error message, and then request another card. Entering the number 0 instead of a card will cause the program to terminate.

A session with the program will have the following appearance:

```
Enter a card: 2s
Enter a card: 5s
Enter a card: 4s
Enter a card: 3s
Enter a card: 6s
Straight flush

Enter a card: 8c
Enter a card: as
Enter a card: 8c
Duplicate card; ignored.
Enter a card: 7c
Enter a card: ad
Enter a card: 3h
Pair

Enter a card: 6s
Enter a card: d2
Bad card; ignored.
Enter a card: 2d
Enter a card: 9c
Enter a card: 4h
Enter a card: ts
```

```
High card

Enter a card: 0
```

From this description of the program, we see that it has three tasks:

Read a hand of five cards.
Analyze the hand for pairs, straights, and so forth.
Print the classification of the hand.

We'll divide the program into three functions—`read_cards`, `analyze_hand`, and `print_result`—that perform these three tasks. `main` does nothing but call these functions inside an endless loop. The functions will need to share a fairly large amount of information, so we'll have them communicate through external variables. `read_cards` will store information about the hand into several external variables. `analyze_hand` will then examine these variables, storing its findings into other external variables for the benefit of `print_result`.

Based on this preliminary design, we can begin to sketch an outline of the program:

```
/* #include directives go here */

/* #define directives go here */

/* declarations of external variables go here */

/* prototypes */
void read_cards(void);
void analyze_hand(void);
void print_result(void);

/**********************************************************
 * main: Calls read_cards, analyze_hand, and print_result *
 *       repeatedly.                                      *
 **********************************************************/
int main(void)
{
  for (;;) {
    read_cards();
    analyze_hand();
    print_result();
  }
}

/**********************************************************
 * read_cards:  Reads the cards into external variables;  *
 *              checks for bad cards and duplicate cards. *
 **********************************************************/
void read_cards(void)
{
  ...
}
```

```
/**********************************************************
 * analyze_hand: Determines whether the hand contains a   *
 *               straight, a flush, four-of-a-kind,       *
 *               and/or three-of-a-kind; determines the   *
 *               number of pairs; stores the results into *
 *               external variables.                      *
 **********************************************************/
void analyze_hand(void)
{
  ...
}

/**********************************************************
 * print_result: Notifies the user of the result, using   *
 *               the external variables set by            *
 *               analyze_hand.                            *
 **********************************************************/
void print_result(void)
{
  ...
}
```

The most pressing question that remains is how to represent the hand of cards. Let's see what operations `read_cards` and `analyze_hand` will perform on the hand. During the analysis of the hand, `analyze_hand` will need to know how many cards are in each rank and each suit. This suggests that we use two arrays, `num_in_rank` and `num_in_suit`. The value of `num_in_rank[r]` will be the number of cards with rank `r`, and the value of `num_in_suit[s]` will be the number of cards with suit `s`. (We'll encode ranks as numbers between 0 and 12, and suits as numbers between 0 and 3.) We'll also need a third array, `card_exists`, so that `read_cards` can detect duplicate cards. Each time `read_cards` reads a card with rank `r` and suit `s`, it checks whether the value of `card_exists[r][s]` is `true`. If so, the card was previously entered; if not, `read_cards` assigns `true` to `card_exists[r][s]`.

Both the `read_cards` function and the `analyze_hand` function will need access to the `num_in_rank` and `num_in_suit` arrays, so I'll make them external variables. The `card_exists` array is used only by `read_cards`, so it can be local to that function. As a rule, variables should be made external only if necessary.

Having decided on the major data structures, we can now finish the program:

poker.c

```
/* Classifies a poker hand */

#include <stdbool.h>   /* C99 only */
#include <stdio.h>
#include <stdlib.h>

#define NUM_RANKS 13
#define NUM_SUITS 4
#define NUM_CARDS 5
```

```
/* external variables */
int num_in_rank[NUM_RANKS];
int num_in_suit[NUM_SUITS];
bool straight, flush, four, three;
int pairs;   /* can be 0, 1, or 2 */

/* prototypes */
void read_cards(void);
void analyze_hand(void);
void print_result(void);

/*************************************************************
 * main: Calls read_cards, analyze_hand, and print_result    *
 *       repeatedly.                                         *
 *************************************************************/
int main(void)
{
  for (;;) {
    read_cards();
    analyze_hand();
    print_result();
  }
}

/*************************************************************
 * read_cards: Reads the cards into the external             *
 *             variables num_in_rank and num_in_suit;        *
 *             checks for bad cards and duplicate cards.     *
 *************************************************************/
void read_cards(void)
{
  bool card_exists[NUM_RANKS][NUM_SUITS];
  char ch, rank_ch, suit_ch;
  int rank, suit;
  bool bad_card;
  int cards_read = 0;

  for (rank = 0; rank < NUM_RANKS; rank++) {
    num_in_rank[rank] = 0;
    for (suit = 0; suit < NUM_SUITS; suit++)
      card_exists[rank][suit] = false;
  }

for (suit = 0; suit < NUM_SUITS; suit++)
  num_in_suit[suit] = 0;

while (cards_read < NUM_CARDS) {
  bad_card = false;

  printf("Enter a card: ");

  rank_ch = getchar();
  switch (rank_ch) {
```

```
      case '0':           exit(EXIT_SUCCESS);
      case '2':           rank = 0; break;
      case '3':           rank = 1; break;
      case '4':           rank = 2; break;
      case '5':           rank = 3; break;
      case '6':           rank = 4; break;
      case '7':           rank = 5; break;
      case '8':           rank = 6; break;
      case '9':           rank = 7; break;
      case 't': case 'T': rank = 8; break;
      case 'j': case 'J': rank = 9; break;
      case 'q': case 'Q': rank = 10; break;
      case 'k': case 'K': rank = 11; break;
      case 'a': case 'A': rank = 12; break;
      default:            bad_card = true;
    }

    suit_ch = getchar();
    switch (suit_ch) {
      case 'c': case 'C': suit = 0; break;
      case 'd': case 'D': suit = 1; break;
      case 'h': case 'H': suit = 2; break;
      case 's': case 'S': suit = 3; break;
      default:            bad_card = true;
    }

    while ((ch = getchar()) != '\n')
      if (ch != ' ') bad_card = true;

    if (bad_card)
      printf("Bad card; ignored.\n");
    else if (card_exists[rank][suit])
      printf("Duplicate card; ignored.\n");
    else {
      num_in_rank[rank]++;
      num_in_suit[suit]++;
      card_exists[rank][suit] = true;
      cards_read++;
    }
  }
}

/**********************************************************
 * analyze_hand: Determines whether the hand contains a   *
 *               straight, a flush, four-of-a-kind,       *
 *               and/or three-of-a-kind; determines the   *
 *               number of pairs; stores the results into *
 *               the external variables straight, flush,  *
 *               four, three, and pairs.                  *
 **********************************************************/
void analyze_hand(void)
{
  int num_consec = 0;
  int rank, suit;
```

```
  straight = false;
  flush = false;
  four = false;
  three = false;
  pairs = 0;

  /* check for flush */
  for (suit = 0; suit < NUM_SUITS; suit++)
    if (num_in_suit[suit] == NUM_CARDS)
      flush = true;

  /* check for straight */
  rank = 0;
  while (num_in_rank[rank] == 0) rank++;
  for (; rank < NUM_RANKS && num_in_rank[rank] > 0; rank++)
    num_consec++;
  if (num_consec == NUM_CARDS) {
    straight = true;
    return;
  }

  /* check for 4-of-a-kind, 3-of-a-kind, and pairs */
  for (rank = 0; rank < NUM_RANKS; rank++) {
    if (num_in_rank[rank] == 4) four = true;
    if (num_in_rank[rank] == 3) three = true;
    if (num_in_rank[rank] == 2) pairs++;
  }
}

/**********************************************************
 * print_result: Prints the classification of the hand,   *
 *               based on the values of the external      *
 *               variables straight, flush, four, three,  *
 *               and pairs.                               *
 **********************************************************/
void print_result(void)
{
  if (straight && flush) printf("Straight flush");
  else if (four)         printf("Four of a kind");
  else if (three &&
           pairs == 1)   printf("Full house");
  else if (flush)        printf("Flush");
  else if (straight)     printf("Straight");
  else if (three)        printf("Three of a kind");
  else if (pairs == 2)   printf("Two pairs");
  else if (pairs == 1)   printf("Pair");
  else                   printf("High card");

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

Notice the use of the `exit` function in `read_cards` (in case `'0'` of the first `switch` statement). `exit` is convenient for this program because of its ability to terminate execution from anywhere in the program.

Q & A

Q: What impact do local variables with static storage duration have on recursive functions? [p. 220]

A: When a function is called recursively, fresh copies are made of its automatic variables for each call. This doesn't occur for static variables, though. Instead, all calls of the function share the *same* static variables.

Q: In the following example, `j` is initialized to the same value as `i`, but there are two variables named `i`:

```
int i = 1;

void f(void)
{
  int j = i;
  int i = 2;
  ...
}
```

Is this code legal? If so, what is `j`'s initial value, 1 or 2?

A: The code is indeed legal. The scope of a local variable doesn't begin until its declaration. Therefore, the declaration of `j` refers to the external variable named `i`. The initial value of `j` will be 1.

Exercises

Section 10.4

W 1. The following program outline shows only function definitions and variable declarations.

```
int a;

void f(int b)
{
  int c;
}

void g(void)
{
  int d;
  {
    int e;
  }
}

int main(void)
{
  int f;
}
```

For each of the following scopes, list all variable and parameter names visible in that scope:

(a) The `f` function
(b) The `g` function
(c) The block in which `e` is declared
(d) The `main` function

2. The following program outline shows only function definitions and variable declarations.

```
int b, c;

void f(void)
{
  int b, d;
}

void g(int a)
{
  int c;
  {
    int a, d;
  }
}

int main(void)
{
  int c, d;
}
```

For each of the following scopes, list all variable and parameter names visible in that scope. If there's more than one variable or parameter with the same name, indicate which one is visible.

(a) The `f` function
(b) The `g` function
(c) The block in which `a` and `d` are declared
(d) The `main` function

*3. Suppose that a program has only one function (`main`). How many different variables named `i` could this program contain?

Programming Projects

1. Modify the stack example of Section 10.2 so that it stores characters instead of integers. Next, add a `main` function that asks the user to enter a series of parentheses and/or braces, then indicates whether or not they're properly nested:

```
Enter parentheses and/or braces: (){}{()})
Parentheses/braces are nested properly
```

Hint: As the program reads characters, have it push each left parenthesis or left brace. When it reads a right parenthesis or brace, have it pop the stack and check that the item popped is a matching parenthesis or brace. (If not, the parentheses/braces aren't nested properly.) When the program reads the new-line character, have it check whether the stack is empty; if so, the parentheses/braces are matched. If the stack *isn't* empty (or if `stack_underflow` is ever

called), the parentheses/braces aren't matched. If `stack_overflow` is called, have the program print the message `Stack overflow` and terminate immediately.

2. Modify the `poker.c` program of Section 10.5 by moving the `num_in_rank` and `num_in_suit` arrays into `main`, which will pass them as arguments to `read_cards` and `analyze_hand`.

W 3. Remove the `num_in_rank`, `num_in_suit`, and `card_exists` arrays from the `poker.c` program of Section 10.5. Have the program store the cards in a 5 × 2 array instead. Each row of the array will represent a card. For example, if the array is named `hand`, then `hand[0][0]` will store the rank of the first card and `hand[0][1]` will store the suit of the first card.

4. Modify the `poker.c` program of Section 10.5 by having it recognize an additional category, "royal flush" (ace, king, queen, jack, ten of the same suit). A royal flush ranks higher than all other hands.

W 5. Modify the `poker.c` program of Section 10.5 by allowing "ace-low" straights (ace, two, three, four, five).

6. Some calculators (notably those from Hewlett-Packard) use a system of writing mathematical expressions known as Reverse Polish Notation (RPN). In this notation, operators are placed *after* their operands instead of *between* their operands. For example, 1 + 2 would be written 1 2 + in RPN, and 1 + 2 * 3 would be written 1 2 3 * +. RPN expressions can easily be evaluated using a stack. The algorithm involves reading the operators and operands in an expression from left to right, performing the following actions:

   When an operand is encountered, push it onto the stack.

   When an operator is encountered, pop its operands from the stack, perform the operation on those operands, and then push the result onto the stack.

   Write a program that evaluates RPN expressions. The operands will be single-digit integers. The operators are +, –, *, /, and =. The = operator causes the top stack item to be displayed; afterwards, the stack is cleared and the user is prompted to enter another expression. The process continues until the user enters a character that is not an operator or operand:

```
Enter an RPN expression: 1 2 3 * + =
Value of expression: 7
Enter an RPN expression: 5 8 * 4 9 - / =
Value of expression: -8
Enter an RPN expression: q
```

   If the stack overflows, the program will display the message `Expression is too complex` and terminate. If the stack underflows (because of an expression such as 1 2 + +), the program will display the message `Not enough operands in expression` and terminate. *Hints:* Incorporate the stack code from Section 10.2 into your program. Use `scanf(" %c", &ch)` to read the operators and operands.

7. Write a program that prompts the user for a number and then displays the number, using characters to simulate the effect of a seven-segment display:

```
Enter a number: 491-9014
```

```
 _   _       _   _   _       _   _
|_| |_|   | |_| |_  |_|   | |_| |_|
  |  _|   | |_|  _| |_|   | |   |
```

   Characters other than digits should be ignored. Write the program so that the maximum number of digits is controlled by a macro named `MAX_DIGITS`, which has the value 10. If

the number contains more than this number of digits, the extra digits are ignored. *Hints:* Use two external arrays. One is the `segments` array (see Exercise 6 in Chapter 8), which stores data representing the correspondence between digits and segments. The other array, `digits`, will be an array of characters with 4 rows (since each segmented digit is four characters high) and `MAX_DIGITS * 4` columns (digits are three characters wide, but a space is needed between digits for readability). Write your program as four functions: `main`, `clear_digits_array`, `process_digit`, and `print_digits_array`. Here are the prototypes for the latter three functions:

```
void clear_digits_array(void);
void process_digit(int digit, int position);
void print_digits_array(void);
```

`clear_digits_array` will store blank characters into all elements of the `digits` array. `process_digit` will store the seven-segment representation of `digit` into a specified position in the `digits` array (positions range from 0 to `MAX_DIGITS – 1`). `print_digits_array` will display the rows of the `digits` array, each on a single line, producing output such as that shown in the example.

11 Pointers

The 11th commandment was "Thou Shalt Compute"
or "Thou Shalt Not Compute"—I forget which.

Pointers are one of C's most important—and most often misunderstood—features. Because of their importance, we'll devote three chapters to pointers. In this chapter, we'll concentrate on the basics; Chapters 12 and 17 cover more advanced uses of pointers.

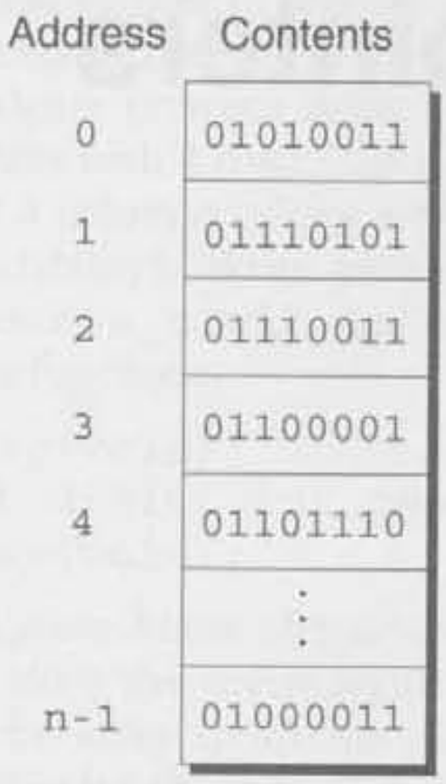
We'll start with a discussion of memory addresses and their relationship to pointer variables (Section 11.1). Section 11.2 then introduces the address and indirection operators. Section 11.3 covers pointer assignment. Section 11.4 explains how to pass pointers to functions, while Section 11.5 discusses returning pointers from functions.

11.1 Pointer Variables

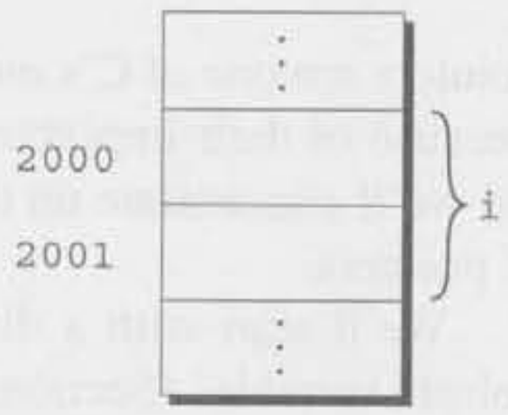
The first step in understanding pointers is visualizing what they represent at the machine level. In most modern computers, main memory is divided into ***bytes***, with each byte capable of storing eight bits of information:

Each byte has a unique ***address*** to distinguish it from the other bytes in memory. If there are n bytes in memory, we can think of addresses as numbers that range from 0 to $n - 1$ (see the figure at the top of the next page).

An executable program consists of both code (machine instructions corresponding to statements in the original C program) and data (variables in the original program). Each variable in the program occupies one or more bytes of memory;



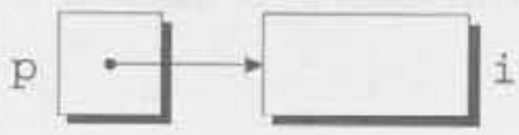
the address of the first byte is said to be the address of the variable. In the following figure, the variable `i` occupies the bytes at addresses 2000 and 2001, so `i`'s address is 2000:



Here's where pointers come in. Although addresses are represented by numbers, their range of values may differ from that of integers, so we can't necessarily store them in ordinary integer variables. We can, however, store them in special ***pointer variables***. When we store the address of a variable `i` in the pointer variable `p`, we say that `p` "points to" `i`. In other words, a pointer is nothing more than an address, and a pointer variable is just a variable that can store an address.

Q&A

Instead of showing addresses as numbers in our examples, I'll use a simpler notation. To indicate that a pointer variable `p` stores the address of a variable `i`, I'll show the contents of `p` as an arrow directed toward `i`:



Declaring Pointer Variables

A pointer variable is declared in much the same way as an ordinary variable. The only difference is that the name of a pointer variable must be preceded by an asterisk:

```
int *p;
```

This declaration states that p is a pointer variable capable of pointing to ***objects*** of type int. I'm using the term *object* instead of *variable* since—as we'll see in Chapter 17—p might point to an area of memory that doesn't belong to a variable. (Be aware that "object" will have a different meaning when we discuss program design in Chapter 19.)

abstract objects ➤ 19.1

Pointer variables can appear in declarations along with other variables:

```
int i, j, a[10], b[20], *p, *q;
```

In this example, i and j are ordinary integer variables, a and b are arrays of integers, and p and q are pointers to integer objects.

C requires that every pointer variable point only to objects of a particular type (the ***referenced type***):

```
int *p;       /* points only to integers   */
double *q;    /* points only to doubles    */
char *r;      /* points only to characters */
```

There are no restrictions on what the referenced type may be. In fact, a pointer variable can even point to another pointer.

pointers to pointers ➤ 17.6

11.2 The Address and Indirection Operators

C provides a pair of operators designed specifically for use with pointers. To find the address of a variable, we use the & (address) operator. If x is a variable, then &x is the address of x in memory. To gain access to the object that a pointer points to, we use the * (***indirection***) operator. If p is a pointer, then *p represents the object to which p currently points.

The Address Operator

Declaring a pointer variable sets aside space for a pointer but doesn't make it point to an object:

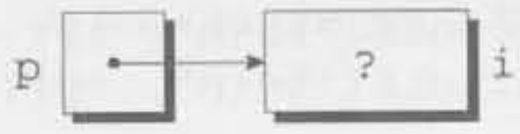
```
int *p;   /* points nowhere in particular */
```

It's crucial to initialize p before we use it. One way to initialize a pointer variable is to assign it the address of some variable—or, more generally, lvalue—using the & operator:

lvalues ➤ 4.2

```
int i, *p;
...
p = &i;
```

By assigning the address of i to the variable p, this statement makes p point to i:



It's also possible to initialize a pointer variable at the time we declare it:

```
int i;
int *p = &i;
```

Q&A

We can even combine the declaration of `i` with the declaration of `p`, provided that `i` is declared first:

```
int i, *p = &i;
```

The Indirection Operator

Once a pointer variable points to an object, we can use the `*` (indirection) operator to access what's stored in the object. If `p` points to `i`, for example, we can print the value of `i` as follows:

```
printf("%d\n", *p);
```

Q&A

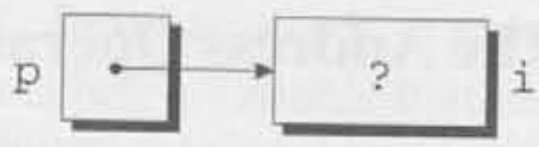
`printf` will display the *value* of `i`, not the *address* of `i`.

The mathematically inclined reader may wish to think of `*` as the inverse of `&`. Applying `&` to a variable produces a pointer to the variable; applying `*` to the pointer takes us back to the original variable:

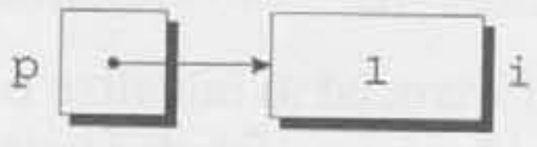
```
j = *&i;    /* same as j = i; */
```

As long as `p` points to `i`, `*p` is an ***alias*** for `i`. Not only does `*p` have the same value as `i`, but changing the value of `*p` also changes the value of `i`. (`*p` is an lvalue, so assignment to it is legal.) The following example illustrates the equivalence of `*p` and `i`; diagrams show the values of `p` and `i` at various points in the computation.

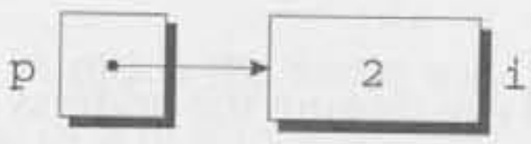
```
p = &i;
```




```
i = 1;
```




```
printf("%d\n", i);     /* prints 1 */
printf("%d\n", *p);    /* prints 1 */
*p = 2;
```




```
printf("%d\n", i);     /* prints 2 */
printf("%d\n", *p);    /* prints 2 */
```

Never apply the indirection operator to an uninitialized pointer variable. If a pointer variable `p` hasn't been initialized, attempting to use the value of `p` in any way causes undefined behavior. In the following example, the call of `printf` may print garbage, cause the program to crash, or have some other effect:

```
int *p;
printf("%d", *p);    /*** WRONG ***/
```

Assigning a value to `*p` is particularly dangerous. If `p` happens to contain a valid memory address, the following assignment will attempt to modify the data stored at that address:

```
int *p;
*p = 1;    /*** WRONG ***/
```

If the location modified by this assignment belongs to the program, it may behave erratically; if it belongs to the operating system, the program will most likely crash. Your compiler may issue a warning that `p` is uninitialized, so pay close attention to any warning messages you get.

11.3 Pointer Assignment

C allows the use of the assignment operator to copy pointers, provided that they have the same type. Suppose that `i`, `j`, `p`, and `q` have been declared as follows:

```
int i, j, *p, *q;
```

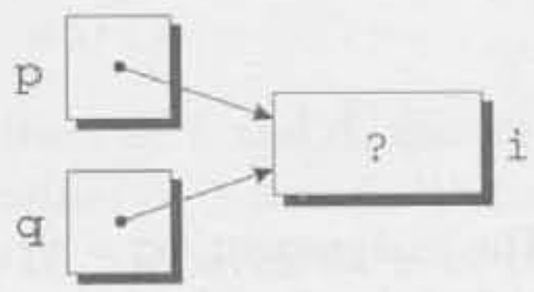
The statement

```
p = &i;
```

is an example of pointer assignment; the address of `i` is copied into `p`. Here's another example of pointer assignment:

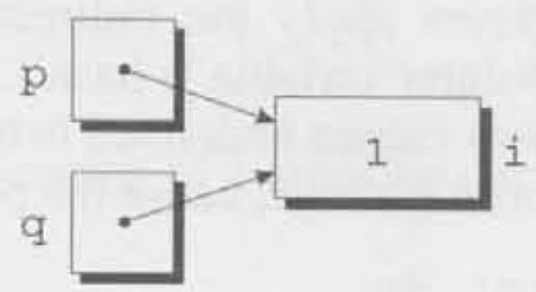
```
q = p;
```

This statement copies the contents of `p` (the address of `i`) into `q`, in effect making `q` point to the same place as `p`:

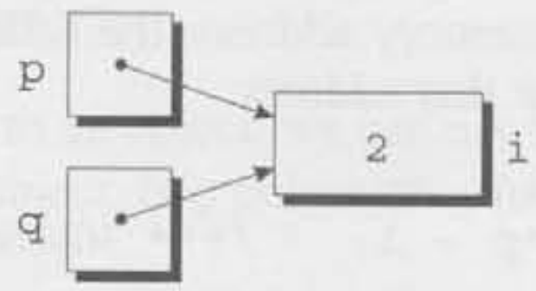


Both `p` and `q` now point to `i`, so we can change `i` by assigning a new value to either `*p` or `*q`:

```
*p = 1;
```




```
*q = 2;
```



Any number of pointer variables may point to the same object.

Be careful not to confuse

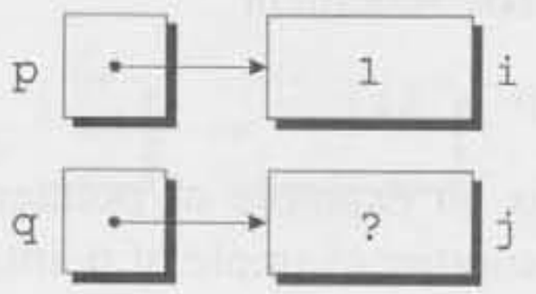
```
q = p;
```

with

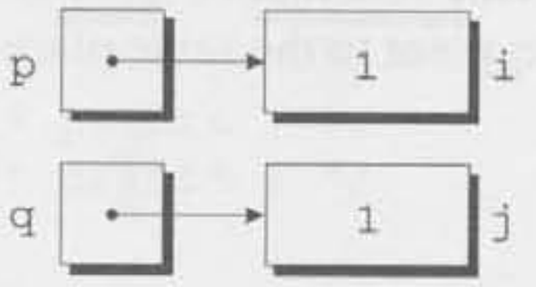
```
*q = *p;
```

The first statement is a pointer assignment; the second isn't, as the following example shows:

```
p = &i;
q = &j;
i = 1;
```




```
*q = *p;
```



The assignment `*q = *p` copies the value that `p` points to (the value of `i`) into the object that `q` points to (the variable `j`).

11.4 Pointers as Arguments

So far, we've managed to avoid a rather important question: What are pointers good for? There's no single answer to that question, since pointers have several distinct uses in C. In this section, we'll see how a pointer to a variable can be useful as a function argument. We'll discover other uses for pointers in Section 11.5 and in Chapters 12 and 17.

We saw in Section 9.3 that a variable supplied as an argument in a function call is protected against change, because C passes arguments by value. This property of C can be a nuisance if we want the function to be able to modify the variable. In Section 9.3, we tried—and failed—to write a `decompose` function that could modify two of its arguments.

Pointers offer a solution to this problem: instead of passing a variable `x` as the argument to a function, we'll supply `&x`, a pointer to `x`. We'll declare the corresponding parameter `p` to be a pointer. When the function is called, `p` will have the value `&x`, hence `*p` (the object that `p` points to) will be an alias for `x`. Each appearance of `*p` in the body of the function will be an indirect reference to `x`, allowing the function both to read `x` and to modify it.

To see this technique in action, let's modify the `decompose` function by declaring the parameters `int_part` and `frac_part` to be pointers. The definition of `decompose` will now look like this:

```
void decompose(double x, long *int_part, double *frac_part)
{
  *int_part = (long) x;
  *frac_part = x - *int_part;
}
```

The prototype for `decompose` could be either

```
void decompose(double x, long *int_part, double *frac_part);
```

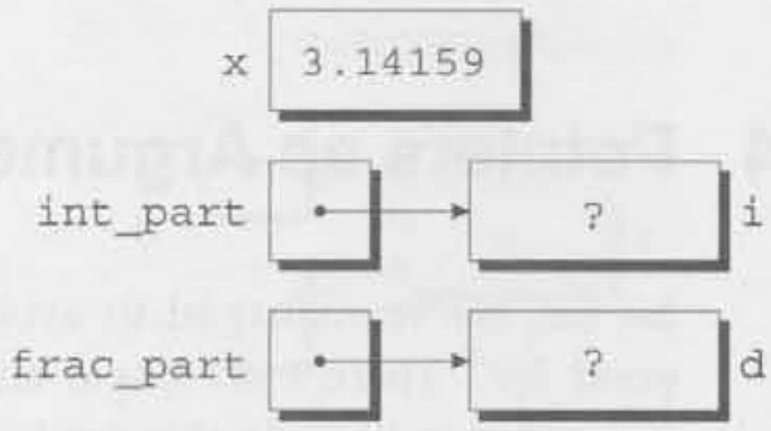
or

```
void decompose(double, long *, double *);
```

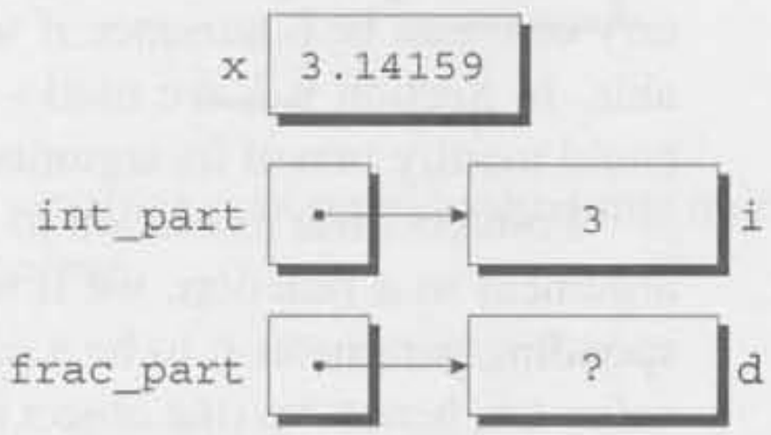
We'll call `decompose` in the following way:

```
decompose(3.14159, &i, &d);
```

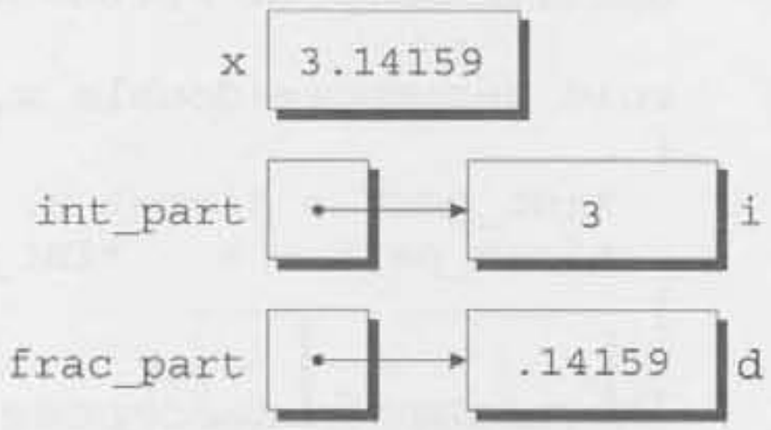
Because of the `&` operator in front of `i` and `d`, the arguments to `decompose` are *pointers* to `i` and `d`, not the *values* of `i` and `d`. When `decompose` is called, the value 3.14159 is copied into `x`, a pointer to `i` is stored in `int_part`, and a pointer to `d` is stored in `frac_part`:



The first assignment in the body of `decompose` converts the value of `x` to type `long` and stores it in the object pointed to by `int_part`. Since `int_part` points to `i`, the assignment puts the value 3 in `i`:



The second assignment fetches the value that `int_part` points to (the value of `i`), which is 3. This value is converted to type `double` and subtracted from `x`, giving .14159, which is then stored in the object that `frac_part` points to:



When `decompose` returns, `i` and `d` will have the values 3 and .14159, just as we originally wanted.

Using pointers as arguments to functions is actually nothing new; we've been doing it in calls of `scanf` since Chapter 2. Consider the following example:

```
int i;
...
scanf("%d", &i);
```

We must put the `&` operator in front of `i` so that `scanf` is given a *pointer* to `i`; that pointer tells `scanf` where to put the value that it reads. Without the `&`, `scanf` would be supplied with the *value* of `i`.

Although `scanf`'s arguments must be pointers, it's not always true that every argument needs the `&` operator. In the following example, `scanf` is passed a pointer variable:

```
int i, *p;
...
p = &i;
scanf("%d", p);
```

Since `p` contains the address of `i`, `scanf` will read an integer and store it in `i`. Using the `&` operator in the call would be wrong:

```
scanf("%d", &p);    /*** WRONG ***/
```

`scanf` would read an integer and store it in `p` instead of in `i`.

Failing to pass a pointer to a function when one is expected can have disastrous results. Suppose that we call `decompose` without the `&` operator in front of `i` and `d`:

```
decompose(3.14159, i, d);
```

`decompose` is expecting pointers as its second and third arguments, but it's been given the *values* of `i` and `d` instead. `decompose` has no way to tell the difference, so it will use the values of `i` and `d` as though they were pointers. When `decompose` stores values in `*int_part` and `*frac_part`, it will attempt to change unknown memory locations instead of modifying `i` and `d`.

If we've provided a prototype for `decompose` (as we should always do, of course), the compiler will let us know that we're attempting to pass arguments of the wrong type. In the case of `scanf`, however, failing to pass pointers often goes undetected by the compiler, making `scanf` an especially error-prone function.

PROGRAM

Finding the Largest and Smallest Elements in an Array

To illustrate how pointers are passed to functions, let's look at a function named `max_min` that finds the largest and smallest elements in an array. When we call `max_min`, we'll pass it pointers to two variables; `max_min` will then store its answers in these variables. `max_min` has the following prototype:

```
void max_min(int a[], int n, int *max, int *min);
```

A call of `max_min` might have the following appearance:

```
max_min(b, N, &big, &small);
```

`b` is an array of integers; `N` is the number of elements in `b`. `big` and `small` are ordinary integer variables. When `max_min` finds the largest element in `b`, it stores the value in `big` by assigning it to `*max`. (Since `max` points to `big`, an assignment to `*max` will modify the value of `big`.) `max_min` stores the smallest element of `b` in `small` by assigning it to `*min`.

To test `max_min`, we'll write a program that reads 10 numbers into an array, passes the array to `max_min`, and prints the results:

```
Enter 10 numbers: 34 82 49 102 7 94 23 11 50 31
Largest: 102
Smallest: 7
```

Here's the complete program:

maxmin.c

```
/* Finds the largest and smallest elements in an array */

#include <stdio.h>

#define N 10

void max_min(int a[], int n, int *max, int *min);

int main(void)
{
  int b[N], i, big, small;

  printf("Enter %d numbers: ", N);
  for (i = 0; i < N; i++)
    scanf("%d", &b[i]);

  max_min(b, N, &big, &small);

  printf("Largest: %d\n", big);
  printf("Smallest: %d\n", small);

  return 0;
}

void max_min(int a[], int n, int *max, int *min)
{
  int i;

  *max = *min = a[0];
  for (i = 1; i < n; i++) {
    if (a[i] > *max)
      *max = a[i];
    else if (a[i] < *min)
      *min = a[i];
  }
}
```

Using `const` to Protect Arguments

When we call a function and pass it a pointer to a variable, we normally assume that the function will modify the variable (otherwise, why would the function require a pointer?). For example, if we see a statement like

```
f(&x);
```

in a program, we'd probably expect `f` to change the value of `x`. It's possible, though, that `f` merely needs to examine the value of `x`, not change it. The reason for the pointer might be efficiency: passing the value of a variable can waste time and space if the variable requires a large amount of storage. (Section 12.3 covers this point in more detail.)

We can use the word `const` to document that a function won't change an object whose address is passed to the function. `const` goes in the parameter's declaration, just before the specification of its type:

Q&A

```
void f(const int *p)
{
  *p = 0;    /*** WRONG ***/
}
```

This use of `const` indicates that `p` is a pointer to a "constant integer." Attempting to modify `*p` is an error that the compiler will detect.

11.5 Pointers as Return Values

We can not only pass pointers to functions but also write functions that *return* pointers. Such functions are relatively common; we'll encounter several in Chapter 13.

The following function, when given pointers to two integers, returns a pointer to whichever integer is larger:

```
int *max(int *a, int *b)
{
  if (*a > *b)
    return a;
  else
    return b;
}
```

When we call `max`, we'll pass pointers to two `int` variables and store the result in a pointer variable:

```
int *p, i, j;
...
p = max(&i, &j);
```

During the call of `max`, `*a` is an alias for `i`, while `*b` is an alias for `j`. If `i` has a larger value than `j`, `max` returns the address of `i`; otherwise, it returns the address of `j`. After the call, `p` points to either `i` or `j`.

Although the `max` function returns one of the pointers passed to it as an argument, that's not the only possibility. A function could also return a pointer to an external variable or to a local variable that's been declared `static`.

Never return a pointer to an *automatic* local variable:

```
int *f(void)
{
  int i;
  …
  return &i;
}
```

The variable `i` doesn't exist once `f` returns, so the pointer to it will be invalid. Some compilers issue a warning such as "*function returns address of local variable*" in this situation.

Pointers can point to array elements, not just ordinary variables. If `a` is an array, then `&a[i]` is a pointer to element `i` of `a`. When a function has an array argument, it's sometimes useful for the function to return a pointer to one of the elements in the array. For example, the following function returns a pointer to the middle element of the array `a`, assuming that `a` has `n` elements:

```
int *find_middle(int a[], int n) {
  return &a[n/2];
}
```

Chapter 12 explores the relationship between pointers and arrays in considerable detail.

Q & A

***Q: Is a pointer always the same as an address? [p. 242]**

A: Usually, but not always. Consider a computer whose main memory is divided into ***words*** rather than bytes. A word might contain 36 bits, 60 bits, or some other number of bits. If we assume 36-bit words, memory will have the following appearance:

Address	Contents
0	001010011001010011001010011001010011
1	001110101001110101001110101001110101
2	001110011001110011001110011001110011
3	001100001001100001001100001001100001
4	001101110001101110001101110001101110
	⋮
n-1	001000011001000011001000011001000011

When memory is divided into words, each word has an address. An integer usually occupies one word, so a pointer to an integer can just be an address. However, a word can store more than one character. For example, a 36-bit word might store six 6-bit characters:

or four 9-bit characters:

For this reason, a pointer to a character may need to be stored in a different form than other pointers. A pointer to a character might consist of an address (the word in which the character is stored) plus a small integer (the position of the character within the word).

On some computers, pointers may be "offsets" rather than complete addresses. For example, CPUs in the Intel x86 family (used in many personal computers) can execute programs in several modes. The oldest of these, which dates back to the 8086 processor of 1978, is called ***real mode***. In this mode, addresses are sometimes represented by a single 16-bit number (an ***offset***) and sometimes by two 16-bit numbers (a ***segment:offset pair***). An offset isn't a true memory address; the CPU must combine it with a segment value stored in a special register. To support real mode, older C compilers often provide two kinds of pointers: ***near pointers*** (16-bit offsets) and ***far pointers*** (32-bit segment:offset pairs). These compilers usually reserve the words `near` and `far` as nonstandard keywords that can be used to declare pointer variables.

Q: If a pointer can point to *data* in a program, is it possible to have a pointer to program *code?

A: Yes. We'll cover pointers to functions in Section 17.7.

Q: It seems to me that there's an inconsistency between the declaration

```
int *p = &i;
```

and the statement

```
p = &i;
```

Why isn't `p` preceded by a `*` symbol in the statement, as it is in the declaration? [p. 244]

A: The source of the confusion is the fact that the `*` symbol can have different meanings in C, depending on the context in which it's used. In the declaration

```
int *p = &i;
```

the `*` symbol is *not* the indirection operator. Instead, it helps specify the type of `p`, informing the compiler that `p` is a *pointer* to an `int`. When it appears in a statement,

however, the * symbol performs indirection (when used as a unary operator). The statement

```
*p = &i;    /*** WRONG ***/
```

would be wrong, because it assigns the address of `i` to the object that `p` points to, not to `p` itself.

Q: Is there some way to print the address of a variable? [p. 244]

A: Any pointer, including the address of a variable, can be displayed by calling the `printf` function and using `%p` as the conversion specification. See Section 22.3 for details.

Q: The following declaration is confusing:

```
void f(const int *p);
```

Does this say that `f` can't modify `p`? [p. 251]

A: No. It says that `f` can't change the integer that `p` *points to*; it doesn't prevent `f` from changing `p` itself.

```
void f(const int *p)
{
  int j;

  *p = 0;    /*** WRONG ***/
  p = &j;    /* legal */
}
```

Since arguments are passed by value, assigning `p` a new value—by making it point somewhere else—won't have any effect outside the function.

***Q: When declaring a parameter of a pointer type, is it legal to put the word `const` in front of the parameter's name, as in the following example?**

```
void f(int * const p);
```

A: Yes, although the effect isn't the same as if `const` precedes `p`'s type. We saw in Section 11.4 that putting `const` *before* `p`'s type protects the object that `p` points to. Putting `const` *after* `p`'s type protects `p` itself:

```
void f(int * const p)
{
  int j;

  *p = 0;    /* legal */
  p = &j;    /*** WRONG ***/
}
```

This feature isn't used very often. Since `p` is merely a copy of another pointer (the argument when the function is called), there's rarely any reason to protect it.

An even greater rarity is the need to protect both `p` *and* the object it points to, which can be done by putting `const` both before and after `p`'s type:

```
void f(const int * const p)
{
  int j;

  *p = 0;    /*** WRONG ***/
  p = &j;    /*** WRONG ***/
}
```

Exercises

Section 11.2

1. If `i` is a variable and `p` points to `i`, which of the following expressions are aliases for `i`?

   (a) `*p` (c) `*&p` (e) `*i` (g) `*&i`
   (b) `&p` (d) `&*p` (f) `&i` (h) `&*i`

Section 11.3

W 2. If `i` is an `int` variable and `p` and `q` are pointers to `int`, which of the following assignments are legal?

   (a) `p = i;` (d) `p = &q;` (g) `p = *q;`
   (b) `*p = &i;` (e) `p = *&q;` (h) `*p = q;`
   (c) `&p = q;` (f) `p = q;` (i) `*p = *q;`

Section 11.4

3. The following function supposedly computes the sum and average of the numbers in the array `a`, which has length `n`. `avg` and `sum` point to variables that the function should modify. Unfortunately, the function contains several errors; find and correct them.

```
void avg_sum(double a[], int n, double *avg, double *sum)
{
  int i;

  sum = 0.0;
  for (i = 0; i < n; i++)
    sum += a[i];
  avg = sum / n;
}
```

W 4. Write the following function:

```
void swap(int *p, int *q);
```

When passed the addresses of two variables, `swap` should exchange the values of the variables:

```
swap(&i, &j);   /* exchanges values of i and j */
```

5. Write the following function:

```
void split_time(long total_sec, int *hr, int *min, int *sec);
```

`total_sec` is a time represented as the number of seconds since midnight. `hr`, `min`, and `sec` are pointers to variables in which the function will store the equivalent time in hours (0–23), minutes (0–59), and seconds (0–59), respectively.

W 6. Write the following function:

```
void find_two_largest(int a[], int n, int *largest,
                      int *second_largest);
```

When passed an array `a` of length `n`, the function will search `a` for its largest and second-largest elements, storing them in the variables pointed to by `largest` and `second_largest`, respectively.

7. Write the following function:

```
void split_date(int day_of_year, int year,
                int *month, int *day);
```

`day_of_year` is an integer between 1 and 366, specifying a particular day within the year designated by `year`. `month` and `day` point to variables in which the function will store the equivalent month (1–12) and day within that month (1–31).

Section 11.5

8. Write the following function:

```
int *find_largest(int a[], int n);
```

When passed an array `a` of length `n`, the function will return a pointer to the array's largest element.

Programming Projects

1. Modify Programming Project 7 from Chapter 2 so that it includes the following function:

```
void pay_amount(int dollars, int *twenties, int *tens,
                int *fives, int *ones);
```

The function determines the smallest number of $20, $10, $5, and $1 bills necessary to pay the amount represented by the `dollars` parameter. The `twenties` parameter points to a variable in which the function will store the number of $20 bills required. The `tens`, `fives`, and `ones` parameters are similar.

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

```
void find_closest_flight(int desired_time,
                         int *departure_time,
                         int *arrival_time);
```

This function will find the flight whose departure time is closest to `desired_time` (expressed in minutes since midnight). It will store the departure and arrival times of this flight (also expressed in minutes since midnight) in the variables pointed to by `departure_time` and `arrival_time`, respectively.

3. Modify Programming Project 3 from Chapter 6 so that it includes the following function:

```
void reduce(int numerator, int denominator,
            int *reduced_numerator,
            int *reduced_denominator);
```

`numerator` and `denominator` are the numerator and denominator of a fraction. `reduced_numerator` and `reduced_denominator` are pointers to variables in which the function will store the numerator and denominator of the fraction once it has been reduced to lowest terms.

4. Modify the `poker.c` program of Section 10.5 by moving all external variables into `main` and modifying functions so that they communicate by passing arguments. The `analyze_hand` function needs to change the `straight`, `flush`, `four`, `three`, and `pairs` variables, so it will have to be passed pointers to those variables.

12 Pointers and Arrays

Optimization hinders evolution.

Chapter 11 introduced pointers and showed how they're used as function arguments and as values returned by functions. This chapter covers another application for pointers. When pointers point to array elements, C allows us to perform arithmetic—addition and subtraction—on the pointers, which leads to an alternative way of processing arrays in which pointers take the place of array subscripts.

The relationship between pointers and arrays in C is a close one, as we'll soon see. We'll exploit this relationship in subsequent chapters, including Chapter 13 (Strings) and Chapter 17 (Advanced Uses of Pointers). Understanding the connection between pointers and arrays is critical for mastering C: it will give you insight into how C was designed and help you understand existing programs. Be aware, however, that one of the primary reasons for using pointers to process arrays—efficiency—is no longer as important as it once was, thanks to improved compilers.

Section 12.1 discusses pointer arithmetic and shows how pointers can be compared using the relational and equality operators. Section 12.2 then demonstrates how we can use pointer arithmetic for processing array elements. Section 12.3 reveals a key fact about arrays—an array name can serve as a pointer to the array's first element—and uses it to show how array arguments really work. Section 12.4 shows how the topics of the first three sections apply to multidimensional arrays. Section 12.5 wraps up the chapter by exploring the relationship between pointers and variable-length arrays, a C99 feature.

12.1 Pointer Arithmetic

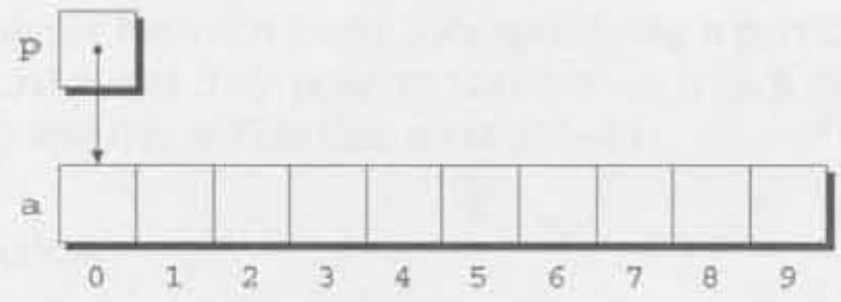
We saw in Section 11.5 that pointers can point to array elements. For example, suppose that `a` and `p` have been declared as follows:

```
int a[10], *p;
```

We can make p point to a[0] by writing

```
p = &a[0];
```

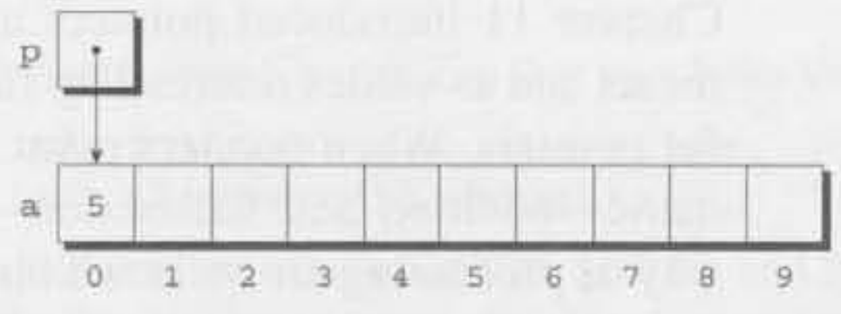
Graphically, here's what we've just done:



We can now access a[0] through p; for example, we can store the value 5 in a[0] by writing

```
*p = 5;
```

Here's our picture now:



Making a pointer p point to an element of an array a isn't particularly exciting. However, by performing ***pointer arithmetic*** (or ***address arithmetic***) on p, we can access the other elements of a. C supports three (and only three) forms of pointer arithmetic:

Adding an integer to a pointer
Subtracting an integer from a pointer
Subtracting one pointer from another

Let's take a close look at each of these operations. Our examples assume that the following declarations are in effect:

```
int a[10], *p, *q, i;
```

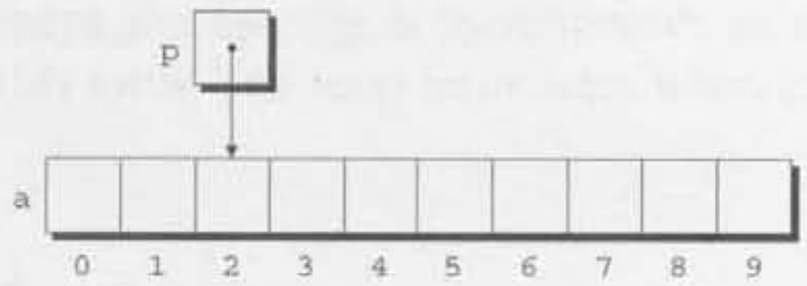
Adding an Integer to a Pointer

Adding an integer j to a pointer p yields a pointer to the element j places after the one that p points to. More precisely, if p points to the array element a[i], then p + j points to a[i+j] (provided, of course, that a[i+j] exists).

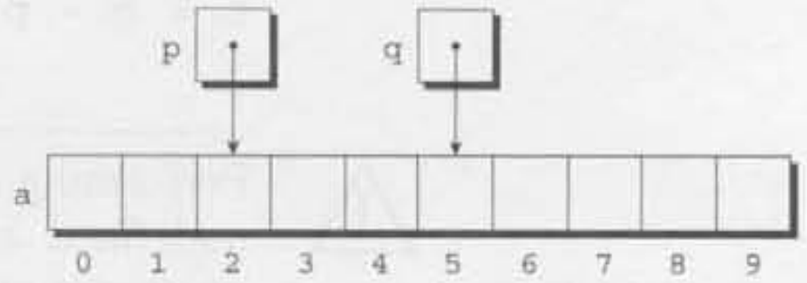
Q&A

The following example illustrates pointer addition; diagrams show the values of p and q at various points in the computation.

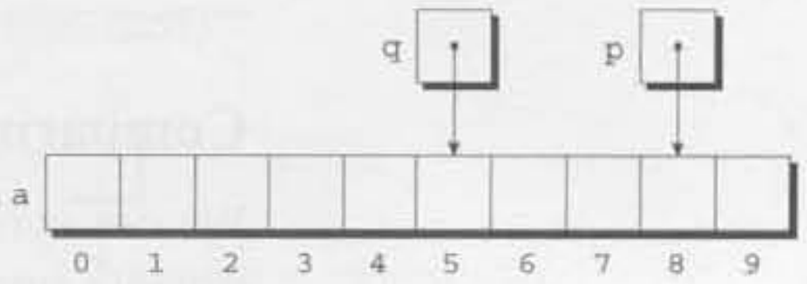
```
p = &a[2];
```




```
q = p + 3;
```



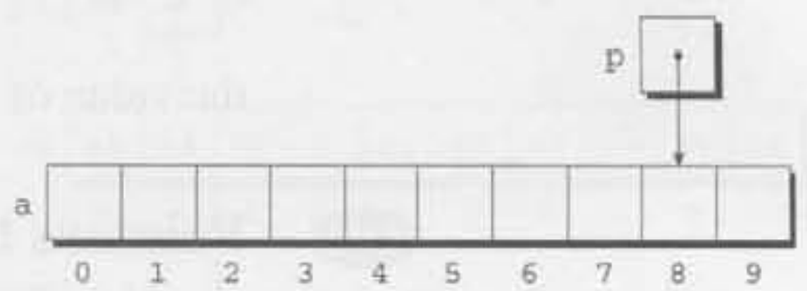

```
p += 6;
```



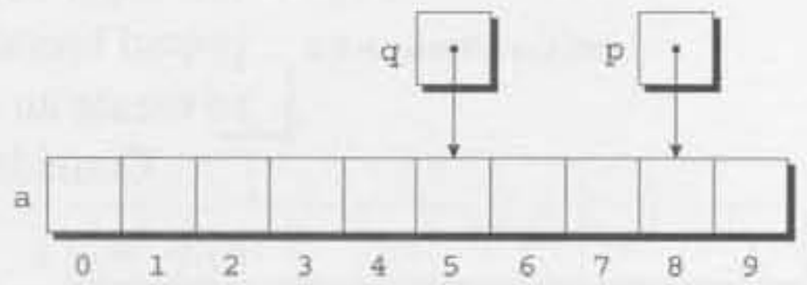
Subtracting an Integer from a Pointer

If `p` points to the array element `a[i]`, then `p - j` points to `a[i-j]`. For example:

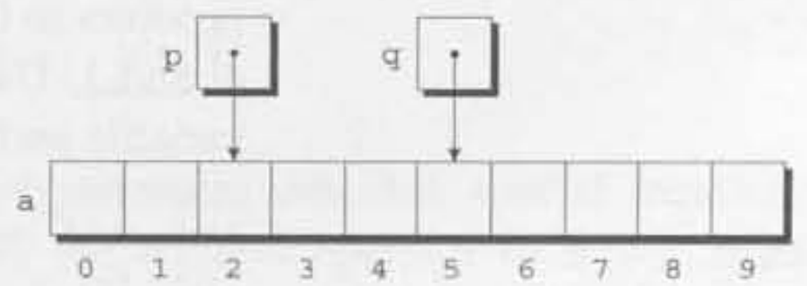
```
p = &a[8];
```




```
q = p - 3;
```



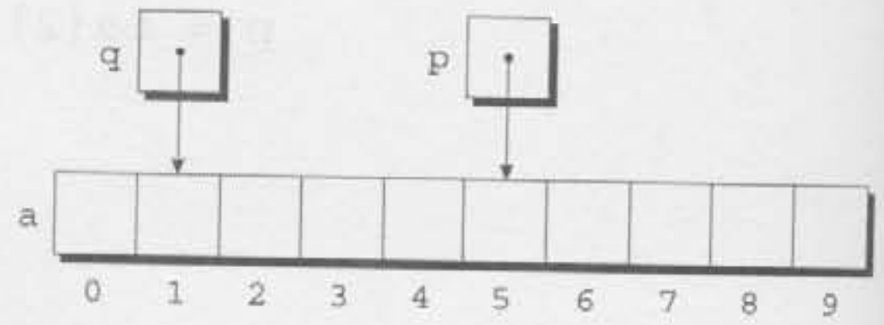

```
p -= 6;
```



Subtracting One Pointer from Another

When one pointer is subtracted from another, the result is the distance (measured in array elements) between the pointers. Thus, if `p` points to `a[i]` and `q` points to `a[j]`, then `p - q` is equal to `i - j`. For example:

```
p = &a[5];
q = &a[1];
```




```
i = p - q;    /* i is 4 */
i = q - p;    /* i is -4 */
```

Performing arithmetic on a pointer that doesn't point to an array element causes undefined behavior. Furthermore, the effect of subtracting one pointer from another is undefined unless both point to elements of the *same* array.

Comparing Pointers

We can compare pointers using the relational operators (`<`, `<=`, `>`, `>=`) and the equality operators (`==` and `!=`). Using the relational operators to compare two pointers is meaningful only when both point to elements of the same array. The outcome of the comparison depends on the relative positions of the two elements in the array. For example, after the assignments

```
p = &a[5];
q = &a[1];
```

the value of `p <= q` is 0 and the value of `p >= q` is 1.

C99 Pointers to Compound Literals

compound literals ➤9.3

It's legal for a pointer to point to an element within an array created by a compound literal. A compound literal, you may recall, is a C99 feature that can be used to create an array with no name.

Consider the following example:

```
int *p = (int []){3, 0, 3, 4, 1};
```

`p` points to the first element of a five-element array containing the integers 3, 0, 3, 4, and 1. Using a compound literal saves us the trouble of first declaring an array variable and then making `p` point to the first element of that array:

```
int a[] = {3, 0, 3, 4, 1};
int *p = &a[0];
```

12.2 Using Pointers for Array Processing

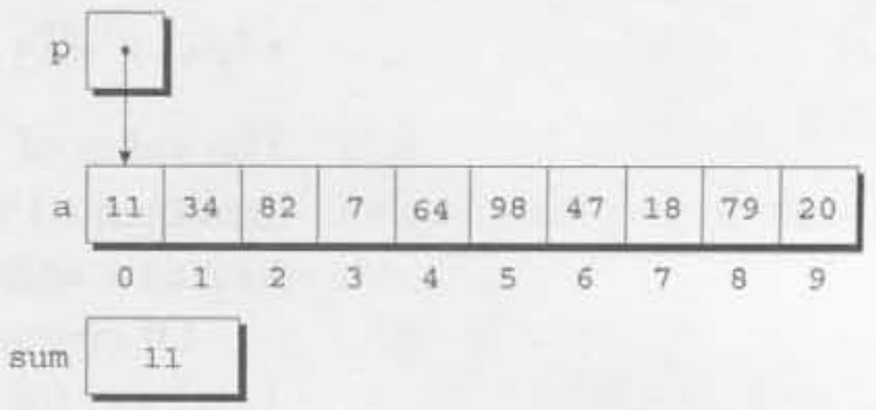
Pointer arithmetic allows us to visit the elements of an array by repeatedly incrementing a pointer variable. The following program fragment, which sums the elements of an array `a`, illustrates the technique. In this example, the pointer variable

p initially points to a[0]. Each time through the loop, p is incremented; as a result, it points to a[1], then a[2], and so forth. The loop terminates when p steps past the last element of a.

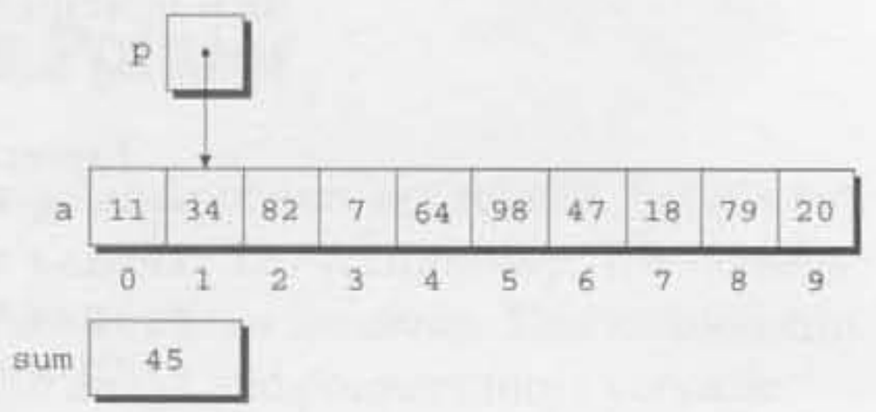
```
#define N 10
...
int a[N], sum, *p;
...
sum = 0;
for (p = &a[0]; p < &a[N]; p++)
  sum += *p;
```

The following figures show the contents of a, sum, and p at the end of the first three loop iterations (before p has been incremented).

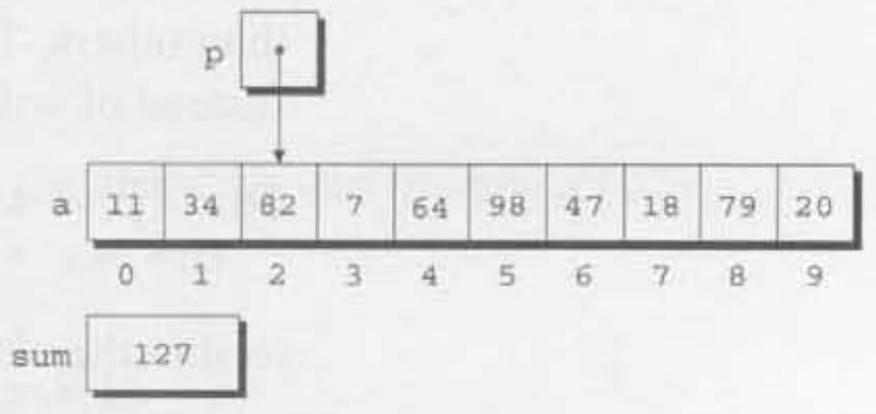
At the end of the first iteration:



At the end of the second iteration:



At the end of the third iteration:



The condition p < &a[N] in the for statement deserves special mention. Strange as it may seem, it's legal to apply the address operator to a[N], even though this element doesn't exist (a is indexed from 0 to N – 1). Using a[N] in this fashion is perfectly safe, since the loop doesn't attempt to examine its value. The body of the loop will be executed with p equal to &a[0], &a[1], ..., &a[N-1], but when p is equal to &a[N], the loop terminates.

We could just as easily have written the loop without pointers, of course, using subscripting instead. The argument most often cited in support of pointer arithmetic is that it can save execution time. However, that depends on the implementation—some C compilers actually produce better code for loops that rely on subscripting.

Q&A

Combining the * and ++ Operators

C programmers often combine the * (indirection) and ++ operators in statements that process array elements. Consider the simple case of storing a value into an array element and then advancing to the next element. Using array subscripting, we might write

```
a[i++] = j;
```

If p is pointing to an array element, the corresponding statement would be

```
*p++ = j;
```

Because the postfix version of ++ takes precedence over *, the compiler sees this as

```
*(p++) = j;
```

The value of p++ is p. (Since we're using the postfix version of ++, p won't be incremented until after the expression has been evaluated.) Thus, the value of *(p++) will be *p—the object to which p is pointing.

Of course, *p++ isn't the only legal combination of * and ++. We could write (*p)++, for example, which returns the value of the object that p points to, and then increments that object (p itself is unchanged). If you find this confusing, the following table may help:

Expression	*Meaning*
*p++ or *(p++)	Value of expression is *p before increment; increment p later
(*p)++	Value of expression is *p before increment; increment *p later
*++p or *(++p)	Increment p first; value of expression is *p after increment
++*p or ++(*p)	Increment *p first; value of expression is *p after increment

All four combinations appear in programs, although some are far more common than others. The one we'll see most frequently is *p++, which is handy in loops. Instead of writing

```
for (p = &a[0]; p < &a[N]; p++)
  sum += *p;
```

to sum the elements of the array a, we could write

```
p = &a[0];
while (p < &a[N])
  sum += *p++;
```

The * and -- operators mix in the same way as * and ++. For an application that combines * and --, let's return to the stack example of Section 10.2. The original version of the stack relied on an integer variable named top to keep track of the "top-of-stack" position in the contents array. Let's replace top by a pointer variable that points initially to element 0 of the contents array:

```
int *top_ptr = &contents[0];
```

Here are the new `push` and `pop` functions (updating the other stack functions is left as an exercise):

```
void push(int i)
{
  if (is_full())
    stack_overflow();
  else
    *top_ptr++ = i;
}

int pop(void)
{
  if (is_empty())
    stack_underflow();
  else
    return *--top_ptr;
}
```

Note that I've written `*--top_ptr`, not `*top_ptr--`, since I want `pop` to decrement `top_ptr` *before* fetching the value to which it points.

12.3 Using an Array Name as a Pointer

Pointer arithmetic is one way in which arrays and pointers are related, but it's not the only connection between the two. Here's another key relationship: *The name of an array can be used as a pointer to the first element in the array.* This relationship simplifies pointer arithmetic and makes both arrays and pointers more versatile.

For example, suppose that `a` is declared as follows:

```
int a[10];
```

Using `a` as a pointer to the first element in the array, we can modify `a[0]`:

```
*a = 7;   /* stores 7 in a[0] */
```

We can modify `a[1]` through the pointer `a + 1`:

```
*(a+1) = 12;   /* stores 12 in a[1] */
```

In general, `a + i` is the same as `&a[i]` (both represent a pointer to element `i` of `a`) and `*(a+i)` is equivalent to `a[i]` (both represent element `i` itself). In other words, array subscripting can be viewed as a form of pointer arithmetic.

The fact that an array name can serve as a pointer makes it easier to write loops that step through an array. Consider the following loop from Section 12.2:

```
for (p = &a[0]; p < &a[N]; p++)
  sum += *p;
```

To simplify the loop, we can replace `&a[0]` by `a` and `&a[N]` by `a + N`:

idiom
```
for (p = a; p < a + N; p++)
  sum += *p;
```

Although an array name can be used as a pointer, it's not possible to assign it a new value. Attempting to make it point elsewhere is an error:

```
while (*a != 0)
  a++;                  /*** WRONG ***/
```

This is no great loss; we can always copy `a` into a pointer variable, then change the pointer variable:

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

PROGRAM

Reversing a Series of Numbers (Revisited)

The `reverse.c` program of Section 8.1 reads 10 numbers, then writes the numbers in reverse order. As the program reads the numbers, it stores them in an array. Once all the numbers are read, the program steps through the array backwards as it prints the numbers.

The original program used subscripting to access elements of the array. Here's a new version in which I've replaced subscripting with pointer arithmetic.

reverse3.c
```
/* Reverses a series of numbers (pointer version) */

#include <stdio.h>

#define N 10

int main(void)
{
  int a[N], *p;

  printf("Enter %d numbers: ", N);
  for (p = a; p < a + N; p++)
    scanf("%d", p);

  printf("In reverse order:");
  for (p = a + N - 1; p >= a; p--)
    printf(" %d", *p);
  printf("\n");

  return 0;
}
```

In the original program, an integer variable `i` kept track of the current position within the array. The new version replaces `i` with `p`, a pointer variable. The num-

bers are still stored in an array; we're simply using a different technique to keep track of where we are in the array.

Note that the second argument to `scanf` is `p`, not `&p`. Since `p` points to an array element, it's a satisfactory argument for `scanf`; `&p`, on the other hand, would be a pointer to a pointer to an array element.

Array Arguments (Revisited)

When passed to a function, an array name is always treated as a pointer. Consider the following function, which returns the largest element in an array of integers:

```
int find_largest(int a[], int n)
{
  int i, max;

  max = a[0];
  for (i = 1; i < n; i++)
    if (a[i] > max)
      max = a[i];
  return max;
}
```

Suppose that we call `find_largest` as follows:

```
largest = find_largest(b, N);
```

This call causes a pointer to the first element of `b` to be assigned to `a`; the array itself isn't copied.

The fact that an array argument is treated as a pointer has some important consequences:

- When an ordinary variable is passed to a function, its value is copied; any changes to the corresponding parameter don't affect the variable. In contrast, an array used as an argument isn't protected against change, since no copy is made of the array itself. For example, the following function (which we first saw in Section 9.3) modifies an array by storing zero into each of its elements:

```
void store_zeros(int a[], int n)
{
  int i;

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

To indicate that an array parameter won't be changed, we can include the word `const` in its declaration:

```
int find_largest(const int a[], int n)
{
  ...
}
```

If `const` is present, the compiler will check that no assignment to an element of `a` appears in the body of `find_largest`.

- The time required to pass an array to a function doesn't depend on the size of the array. There's no penalty for passing a large array, since no copy of the array is made.
- An array parameter can be declared as a pointer if desired. For example, `find_largest` could be defined as follows:

```
int find_largest(int *a, int n)
{
  ...
}
```

Declaring `a` to be a pointer is equivalent to declaring it to be an array; the compiler treats the declarations as though they were identical.

Q&A

Although declaring a *parameter* to be an array is the same as declaring it to be a pointer, the same isn't true for a *variable*. The declaration

```
int a[10];
```

causes the compiler to set aside space for 10 integers. In contrast, the declaration

```
int *a;
```

causes the compiler to allocate space for a pointer variable. In the latter case, `a` is not an array; attempting to use it as an array can have disastrous results. For example, the assignment

```
*a = 0;    /*** WRONG ***/
```

will store 0 where `a` is pointing. Since we don't know where `a` is pointing, the effect on the program is undefined.

- A function with an array parameter can be passed an array "slice"—a sequence of consecutive elements. Suppose that we want `find_largest` to locate the largest element in some portion of an array `b`, say elements `b[5]`, ..., `b[14]`. When we call `find_largest`, we'll pass it the address of `b[5]` and the number `10`, indicating that we want `find_largest` to examine 10 array elements, starting at `b[5]`:

```
largest = find_largest(&b[5], 10);
```

Using a Pointer as an Array Name

If we can use an array name as a pointer, will C allow us to subscript a pointer as though it were an array name? By now, you'd probably expect the answer to be yes, and you'd be right. Here's an example:

```
#define N 10
…
int a[N], i, sum = 0, *p = a;
…
for (i = 0; i < N; i++)
  sum += p[i];
```

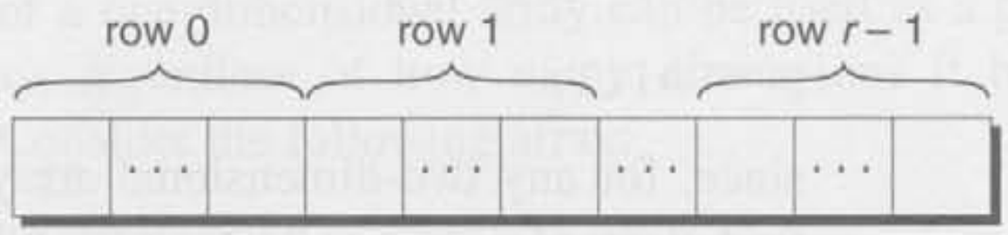
The compiler treats `p[i]` as `*(p+i)`, which is a perfectly legal use of pointer arithmetic. Although the ability to subscript a pointer may seem to be little more than a curiosity, we'll see in Section 17.3 that it's actually quite useful.

12.4 Pointers and Multidimensional Arrays

Just as pointers can point to elements of one-dimensional arrays, they can also point to elements of multidimensional arrays. In this section, we'll explore common techniques for using pointers to process the elements of multidimensional arrays. For simplicity, I'll stick to two-dimensional arrays, but everything we'll do applies equally to higher-dimensional arrays.

Processing the Elements of a Multidimensional Array

We saw in Section 8.2 that C stores two-dimensional arrays in row-major order; in other words, the elements of row 0 come first, followed by the elements of row 1, and so forth. An array with *r* rows would have the following appearance:



We can take advantage of this layout when working with pointers. If we make a pointer `p` point to the first element in a two-dimensional array (the element in row 0, column 0), we can visit every element in the array by incrementing `p` repeatedly.

As an example, let's look at the problem of initializing all elements of a two-dimensional array to zero. Suppose that the array has been declared as follows:

```
int a[NUM_ROWS][NUM_COLS];
```

The obvious technique would be to use nested `for` loops:

```
int row, col;
…
for (row = 0; row < NUM_ROWS; row++)
  for (col = 0; col < NUM_COLS; col++)
    a[row][col] = 0;
```

But if we view a as a one-dimensional array of integers (which is how it's stored), we can replace the pair of loops by a single loop:

```
int *p;
...
for (p = &a[0][0]; p <= &a[NUM_ROWS-1][NUM_COLS-1]; p++)
  *p = 0;
```

The loop begins with p pointing to a[0][0]. Successive increments of p make it point to a[0][1], a[0][2], a[0][3], and so on. When p reaches a[0][NUM_COLS-1] (the last element in row 0), incrementing it again makes p point to a[1][0], the first element in row 1. The process continues until p goes past a[NUM_ROWS-1][NUM_COLS-1], the last element in the array.

Q&A Although treating a two-dimensional array as one-dimensional may seem like cheating, it works with most C compilers. Whether it's a good idea to do so is another matter. Techniques like this one definitely hurt program readability, but—at least with some older compilers—produce a compensating increase in efficiency. With many modern compilers, though, there's often little or no speed advantage.

Processing the Rows of a Multidimensional Array

What about processing the elements in just one *row* of a two-dimensional array? Again, we have the option of using a pointer variable p. To visit the elements of row i, we'd initialize p to point to element 0 in row i in the array a:

```
p = &a[i][0];
```

Or we could simply write

```
p = a[i];
```

since, for any two-dimensional array a, the expression a[i] is a pointer to the first element in row i. To see why this works, recall the magic formula that relates array subscripting to pointer arithmetic: for any array a, the expression a[i] is equivalent to *(a + i). Thus, &a[i][0] is the same as &(*(a[i] + 0)), which is equivalent to &*a[i], which is the same as a[i], since the & and * operators cancel. We'll use this simplification in the following loop, which clears row i of the array a:

```
int a[NUM_ROWS][NUM_COLS], *p, i;
...
for (p = a[i]; p < a[i] + NUM_COLS; p++)
  *p = 0;
```

Since a[i] is a pointer to row i of the array a, we can pass a[i] to a function that's expecting a one-dimensional array as its argument. In other words, a function that's designed to work with one-dimensional arrays will also work with a row belonging to a two-dimensional array. As a result, functions such as

`find_largest` and `store_zeros` are more versatile than you might expect. Consider `find_largest`, which we originally designed to find the largest element of a one-dimensional array. We can just as easily use `find_largest` to determine the largest element in row `i` of the two-dimensional array `a`:

```
largest = find_largest(a[i], NUM_COLS);
```

Processing the Columns of a Multidimensional Array

Processing the elements in a *column* of a two-dimensional array isn't as easy, because arrays are stored by row, not by column. Here's a loop that clears column `i` of the array `a`:

```
int a[NUM_ROWS][NUM_COLS], (*p)[NUM_COLS], i;
...
for (p = &a[0]; p < &a[NUM_ROWS]; p++)
  (*p)[i] = 0;
```

I've declared `p` to be a pointer to an array of length `NUM_COLS` whose elements are integers. The parentheses around `*p` in `(*p)[NUM_COLS]` are required; without them, the compiler would treat `p` as an array of pointers instead of a pointer to an array. The expression `p++` advances `p` to the beginning of the next row. In the expression `(*p)[i]`, `*p` represents an entire row of `a`, so `(*p)[i]` selects the element in column `i` of that row. The parentheses in `(*p)[i]` are essential, because the compiler would interpret `*p[i]` as `*(p[i])`.

Using the Name of a Multidimensional Array as a Pointer

Just as the name of a one-dimensional array can be used as a pointer, so can the name of *any* array, regardless of how many dimensions it has. Some care is required, though. Consider the following array:

```
int a[NUM_ROWS][NUM_COLS];
```

`a` is *not* a pointer to `a[0][0]`; instead, it's a pointer to `a[0]`. This makes more sense if we look at it from the standpoint of C, which regards `a` not as a two-dimensional array but as a one-dimensional array whose elements are one-dimensional arrays. When used as a pointer, `a` has type `int (*)[NUM_COLS]` (pointer to an integer array of length `NUM_COLS`).

Knowing that `a` points to `a[0]` is useful for simplifying loops that process the elements of a two-dimensional array. For example, instead of writing

```
for (p = &a[0]; p < &a[NUM_ROWS]; p++)
  (*p)[i] = 0;
```

to clear column `i` of the array `a`, we can write

```
for (p = a; p < a + NUM_ROWS; p++)
  (*p)[i] = 0;
```

Another situation in which this knowledge comes in handy is when we want to "trick" a function into thinking that a multidimensional array is really one-dimensional. For example, consider how we might use `find_largest` to find the largest element in `a`. As the first argument to `find_largest`, let's try passing `a` (the address of the array); as the second, we'll pass `NUM_ROWS * NUM_COLS` (the total number of elements in `a`):

```
largest = find_largest(a, NUM_ROWS * NUM_COLS);   /* WRONG */
```

Unfortunately, the compiler will object to this statement, because the type of `a` is `int (*)[NUM_COLS]` but `find_largest` is expecting an argument of type `int *`. The correct call is

```
largest = find_largest(a[0], NUM_ROWS * NUM_COLS);
```

`a[0]` points to element 0 in row 0, and it has type `int *` (after conversion by the compiler), so the latter call will work correctly.

Q&A

12.5 Pointers and Variable-Length Arrays (C99)

Pointers are allowed to point to elements of variable-length arrays (VLAs), a feature of C99. An ordinary pointer variable would be used to point to an element of a one-dimensional VLA:

variable-length arrays ➤8.3

```
void f(int n)
{
  int a[n], *p;
  p = a;
  …
}
```

When the VLA has more than one dimension, the type of the pointer depends on the length of each dimension except for the first. Let's look at the two-dimensional case:

```
void f(int m, int n)
{
  int a[m][n], (*p)[n];
  p = a;
  …
}
```

Since the type of `p` depends on `n`, which isn't constant, `p` is said to have a ***variably modified type.*** Note that the validity of an assignment such as `p = a` can't always be determined by the compiler. For example, the following code will compile but is correct only if `m` and `n` are equal:

```
int a[m][n], (*p)[m];
p = a;
```

If m ≠ n, any subsequent use of p will cause undefined behavior.

Variably modified types are subject to certain restrictions, just as variable-length arrays are. The most important restriction is that the declaration of a variably modified type must be inside the body of a function or in a function prototype.

Pointer arithmetic works with VLAs just as it does for ordinary arrays. Returning to the example of Section 12.4 that clears a single column of a two-dimensional array a, let's declare a as a VLA this time:

```
int a[m][n];
```

A pointer capable of pointing to a row of a would be declared as follows:

```
int (*p)[n];
```

The loop that clears column i is almost identical to the one we used in Section 12.4:

```
for (p = a; p < a + m; p++)
  (*p)[i] = 0;
```

Q & A

Q: I don't understand pointer arithmetic. If a pointer is an address, does that mean that an expression like p + j adds j to the address stored in p? [p. 258]

A: No. Integers used in pointer arithmetic are scaled depending on the type of the pointer. If p is of type int *, for example, then p + j typically adds 4 × j to p, assuming that int values are stored using 4 bytes. But if p has type double *, then p + j will probably add 8 × j to p, since double values are usually 8 bytes long.

Q: When writing a loop to process an array, is it better to use array subscripting or pointer arithmetic? [p. 261]

A: There's no easy answer to this question, since it depends on the machine you're using and the compiler itself. In the early days of C on the PDP-11, pointer arithmetic yielded a faster program. On today's machines, using today's compilers, array subscripting is often just as good, and sometimes even better. The bottom line: Learn both ways and then use whichever is more natural for the kind of program you're writing.

***Q: I read somewhere that i[a] is the same as a[i]. Is this true?**

A: Yes, it is, oddly enough. The compiler treats i[a] as *(i + a), which is the same as *(a + i). (Pointer addition, like ordinary addition, is commutative.) But *(a + i) is equivalent to a[i]. Q.E.D. But please don't use i[a] in programs unless you're planning to enter the next Obfuscated C contest.

Q: Why is `*a` the same as `a[]` in a parameter declaration? [p. 266]

A: Both indicate that the argument is expected to be a pointer. The same operations on `a` are possible in both cases (pointer arithmetic and array subscripting, in particular). And, in both cases, `a` itself can be assigned a new value within the function. (Although C allows us to use the name of an array *variable* only as a "constant pointer," there's no such restriction on the name of an array *parameter*.)

Q: Is it better style to declare an array parameter as `*a` or `a[]`?

A: That's a tough one. From one standpoint, `a[]` is the obvious choice, since `*a` is ambiguous (does the function want an array of objects or a pointer to a single object?). On the other hand, many programmers argue that declaring the parameter as `*a` is more accurate, since it reminds us that only a pointer is passed, not a copy of the array. Others switch between `*a` and `a[]`, depending on whether the function uses pointer arithmetic or subscripting to access the elements of the array. (That's the approach I'll use.) In practice, `*a` is more common than `a[]`, so you'd better get used to it. For what it's worth, Dennis Ritchie now refers to the `a[]` notation as "a living fossil" that "serves as much to confuse the learner as to alert the reader."

Q: We've seen that arrays and pointers are closely related in C. Would it be accurate to say that they're interchangeable?

A: No. It's true that array *parameters* are interchangeable with pointer parameters, but array *variables* aren't the same as pointer variables. Technically, the name of an array isn't a pointer; rather, the C compiler *converts* it to a pointer when necessary. To see this difference more clearly, consider what happens when we apply the `sizeof` operator to an array `a`. The value of `sizeof(a)` is the total number of bytes in the array—the size of each element multiplied by the number of elements. But if `p` is a pointer variable, `sizeof(p)` is the number of bytes required to store a pointer value.

Q: You said that treating a two-dimensional array as one-dimensional works with "most" C compilers. Doesn't it work with all compilers? [p. 268]

A: No. Some modern "bounds-checking" compilers track not only the type of a pointer, but—when it points to an array—also the length of the array. For example, suppose that `p` is assigned a pointer to `a[0][0]`. Technically, `p` points to the first element of `a[0]`, a one-dimensional array. If we increment `p` repeatedly in an effort to visit all the elements of `a`, we'll go out of bounds once `p` goes past the last element of `a[0]`. A compiler that performs bounds-checking may insert code to check that `p` is used only to access elements in the array pointed to by `a[0]`; an attempt to increment `p` past the end of this array would be detected as an error.

Q: If `a` is a two-dimensional array, why can we pass `a[0]`—but not `a` itself—to `find_largest`? Don't both `a` and `a[0]` point to the same place (the beginning of the array)? [p. 270]

A: They do, as a matter of fact—both point to element `a[0][0]`. The problem is that

a has the wrong type. When used as an argument, it's a pointer to an array, but `find_largest` is expecting a pointer to an integer. However, `a[0]` has type `int *`, so it's an acceptable argument for `find_largest`. This concern about types is actually good; if C weren't so picky, we could make all kinds of horrible pointer mistakes without the compiler noticing.

Exercises

Section 12.1

1. Suppose that the following declarations are in effect:

```
int a[] = {5, 15, 34, 54, 14, 2, 52, 72};
int *p = &a[1], *q = &a[5];
```

(a) What is the value of `*(p+3)`?
(b) What is the value of `*(q-3)`?
(c) What is the value of `q - p`?
(d) Is the condition `p < q` true or false?
(e) Is the condition `*p < *q` true or false?

W *2. Suppose that `high`, `low`, and `middle` are all pointer variables of the same type, and that `low` and `high` point to elements of an array. Why is the following statement illegal, and how could it be fixed?

```
middle = (low + high) / 2;
```

Section 12.2

3. What will be the contents of the a array after the following statements are executed?

```
#define N 10

int a[N] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int *p = &a[0], *q = &a[N-1], temp;

while (p < q) {
  temp = *p;
  *p++ = *q;
  *q-- = temp;
}
```

W 4. Rewrite the `make_empty`, `is_empty`, and `is_full` functions of Section 10.2 to use the pointer variable `top_ptr` instead of the integer variable `top`.

Section 12.3

5. Suppose that a is a one-dimensional array and p is a pointer variable. Assuming that the assignment `p = a` has just been performed, which of the following expressions are illegal because of mismatched types? Of the remaining expressions, which are true (have a nonzero value)?

(a) `p == a[0]`
(b) `p == &a[0]`
(c) `*p == a[0]`
(d) `p[0] == a[0]`

W 6. Rewrite the following function to use pointer arithmetic instead of array subscripting. (In other words, eliminate the variable `i` and all uses of the `[]` operator.) Make as few changes as possible.

```
int sum_array(const int a[], int n)
{
  int i, sum;

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

7. Write the following function:

```
bool search(const int a[], int n, int key);
```

`a` is an array to be searched, `n` is the number of elements in the array, and `key` is the search key. `search` should return `true` if `key` matches some element of `a`, and `false` if it doesn't. Use pointer arithmetic—not subscripting—to visit array elements.

8. Rewrite the following function to use pointer arithmetic instead of array subscripting. (In other words, eliminate the variable `i` and all uses of the `[]` operator.) Make as few changes as possible.

```
void store_zeros(int a[], int n)
{
  int i;

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

9. Write the following function:

```
double inner_product(const double *a, const double *b,
                     int n);
```

`a` and `b` both point to arrays of length `n`. The function should return `a[0] * b[0] + a[1] * b[1] + ... + a[n-1] * b[n-1]`. Use pointer arithmetic—not subscripting—to visit array elements.

10. Modify the `find_middle` function of Section 11.5 so that it uses pointer arithmetic to calculate the return value.

11. Modify the `find_largest` function so that it uses pointer arithmetic—not subscripting—to visit array elements.

12. Write the following function:

```
void find_two_largest(const int *a, int n, int *largest,
                      int *second_largest);
```

`a` points to an array of length `n`. The function searches the array for its largest and second-largest elements, storing them in the variables pointed to by `largest` and `second_largest`, respectively. Use pointer arithmetic—not subscripting—to visit array elements.

Section 12.4

W 13. Section 8.2 had a program fragment in which two nested `for` loops initialized the array `ident` for use as an identity matrix. Rewrite this code, using a single pointer to step through the array one element at a time. *Hint:* Since we won't be using `row` and `col` index variables, it won't be easy to tell where to store 1. Instead, we can use the fact that the first element of the array should be 1, the next `N` elements should be 0, the next element should

be 1, and so forth. Use a variable to keep track of how many consecutive 0s have been stored; when the count reaches `N`, it's time to store 1.

14. Assume that the following array contains a week's worth of hourly temperature readings, with each row containing the readings for one day:

```
int temperatures[7][24];
```

Write a statement that uses the `search` function (see Exercise 7) to search the entire `temperatures` array for the value 32.

W 15. Write a loop that prints all temperature readings stored in row `i` of the `temperatures` array (see Exercise 14). Use a pointer to visit each element of the row.

16. Write a loop that prints the highest temperature in the `temperatures` array (see Exercise 14) for each day of the week. The loop body should call the `find_largest` function, passing it one row of the array at a time.

17. Rewrite the following function to use pointer arithmetic instead of array subscripting. (In other words, eliminate the variables `i` and `j` and all uses of the `[]` operator.) Use a single loop instead of nested loops.

```
int sum_two_dimensional_array(const int a[][LEN], int n)
{
  int i, j, sum = 0;

  for (i = 0; i < n; i++)
    for (j = 0; j < LEN; j++)
      sum += a[i][j];
  return sum;
}
```

18. Write the `evaluate_position` function described in Exercise 13 of Chapter 9. Use pointer arithmetic—not subscripting—to visit array elements. Use a single loop instead of nested loops.

Programming Projects

W 1. (a) Write a program that reads a message, then prints the reversal of the message:

```
Enter a message: Don't get mad, get even.
Reversal is: .neve teg ,dam teg t'noD
```

Hint: Read the message one character at a time (using `getchar`) and store the characters in an array. Stop reading when the array is full or the character read is `'\n'`.

(b) Revise the program to use a pointer instead of an integer to keep track of the current position in the array.

2. (a) Write a program that reads a message, then checks whether it's a palindrome (the letters in the message are the same from left to right as from right to left):

```
Enter a message: He lived as a devil, eh?
Palindrome

Enter a message: Madam, I am Adam.
Not a palindrome
```

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.