

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

<i>Type</i>	<i>Smallest Value</i>	<i>Largest Value</i>
<code>short int</code>	-32,768	32,767
<code>unsigned short int</code>	0	65,535
<code>int</code>	-32,768	32,767
<code>unsigned int</code>	0	65,535
<code>long int</code>	-2,147,483,648	2,147,483,647
<code>unsigned long int</code>	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

<i>Type</i>	<i>Smallest Value</i>	<i>Largest Value</i>
<code>short int</code>	-32,768	32,767
<code>unsigned short int</code>	0	65,535
<code>int</code>	-2,147,483,648	2,147,483,647
<code>unsigned int</code>	0	4,294,967,295
<code>long int</code>	-2,147,483,648	2,147,483,647
<code>unsigned long int</code>	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

<i>Type</i>	<i>Smallest Value</i>	<i>Largest Value</i>
<code>short int</code>	-32,768	32,767
<code>unsigned short int</code>	0	65,535
<code>int</code>	-2,147,483,648	2,147,483,647
<code>unsigned int</code>	0	4,294,967,295
<code>long int</code>	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
<code>unsigned long int</code>	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.

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*.

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:

<code>float</code>	Single-precision floating-point
<code>double</code>	Double-precision floating-point
<code>long double</code>	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).

subnormal numbers ▶ 23.4

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.

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

C99

- 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

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`.

enumerated types ► 16.5

C99

`_Bool` type ► 5.2

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.

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.

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 sub-categories:

- 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 (non-printing) 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

<i>Name</i>	<i>Escape Sequence</i>
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

Q&A

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

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.

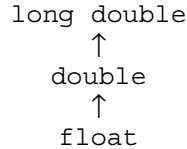
The Usual Arithmetic Conversions

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).

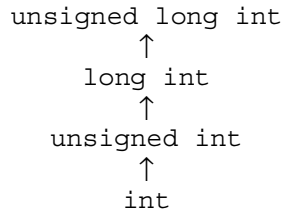
We can divide the rules for performing the usual arithmetic conversions into two cases:

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



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:



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]**

variable-length argument lists
► 26.1

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`.

`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=WXYZ.) 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<br>$y$ 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

- 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  $n$ ?
- Repeat part (a), using an `int` variable instead.
- Repeat part (a), using a `long` variable instead.
- Repeat part (a), using a `long long` variable instead (if your compiler supports the `long long` type).
- Repeat part (a), using a `float` variable instead.
- Repeat part (a), using a `double` variable instead.
- 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.

