4 Expressions

One does not learn computing by using a hand calculator, but one can forget arithmetic.

One of C's distinguishing characteristics is its emphasis on expressions—formulas that show how to compute a value—rather than statements. The simplest expressions are variables and constants. A variable represents a value to be computed as the program runs; a constant represents a value that doesn't change. More complicated expressions apply operators to operands (which are themselves expressions). In the expression `a + (b * c)`, the `+` operator is applied to the operands `a` and `(b * c)`, both of which are expressions in their own right.

Operators are the basic tools for building expressions, and C has an unusually rich collection of them. To start off, C provides the rudimentary operators that are found in most programming languages:

- Arithmetic operators, including addition, subtraction, multiplication, and division.
- Relational operators to perform comparisons such as "`i` is *greater than* 0."
- Logical operators to build conditions such as "`i` is greater than 0 *and* `i` is less than 10."

But C doesn't stop here; it goes on to provide dozens of other operators. There are so many operators, in fact, that we'll need to introduce them gradually over the first twenty chapters of this book. Mastering so many operators can be a chore, but it's essential to becoming proficient at C.

In this chapter, we'll cover some of C's most fundamental operators: the arithmetic operators (Section 4.1), the assignment operators (Section 4.2), and the increment and decrement operators (Section 4.3). Section 4.1 also explains operator precedence and associativity, which are important for expressions that contain more than one operator. Section 4.4 describes how C expressions are evaluated. Finally, Section 4.5 introduces the expression statement, an unusual feature that allows any expression to serve as a statement.

4.1 Arithmetic Operators

The ***arithmetic operators***—operators that perform addition, subtraction, multiplication, and division—are the workhorses of many programming languages, including C. Table 4.1 shows C's arithmetic operators.

Table 4.1
Arithmetic Operators

Unary	*Binary*	
	Additive	*Multiplicative*
+ unary plus	+ addition	* multiplication
- unary minus	- subtraction	/ division
		% remainder

The additive and multiplicative operators are said to be ***binary*** because they require *two* operands. The ***unary*** operators require *one* operand:

```
i = +1;    /* + used as a unary operator */
j = -i;    /* - used as a unary operator */
```

The unary + operator does nothing; in fact, it didn't even exist in K&R C. It's used primarily to emphasize that a numeric constant is positive.

The binary operators probably look familiar. The only one that might not is `%`, the remainder operator. The value of `i % j` is the remainder when `i` is divided by `j`. For example, the value of `10 % 3` is 1, and the value of `12 % 4` is 0.

Q&A The binary operators in Table 4.1—with the exception of `%`—allow either integer or floating-point operands, with mixing allowed. When `int` and `float` operands are mixed, the result has type `float`. Thus, `9 + 2.5f` has the value 11.5, and `6.7f / 2` has the value 3.35.

The `/` and `%` operators require special care:

- The `/` operator can produce surprising results. When both of its operands are integers, the `/` operator "truncates" the result by dropping the fractional part. Thus, the value of `1 / 2` is 0, not 0.5.
- The `%` operator requires integer operands; if either operand is not an integer, the program won't compile.

undefined behavior ➤4.4

- Using zero as the right operand of either `/` or `%` causes undefined behavior.

Q&A

- Describing the result when `/` and `%` are used with negative operands is tricky. The C89 standard states that if either operand is negative, the result of a division can be rounded either up or down. (For example, the value of `-9 / 7` could be either –1 or –2). If `i` or `j` is negative, the sign of `i % j` in C89 depends on the implementation. (For example, the value of `-9 % 7` could be either –2 or 5). In C99, on the other hand, the result of a division is always truncated toward zero (so `-9 / 7` has the value –1) and the value of `i % j` has the same sign as `i` (hence the value of `-9 % 7` is –2).

C99

Implementation-Defined Behavior

The term ***implementation-defined*** will arise often enough that it's worth taking a moment to discuss it. The C standard deliberately leaves parts of the language unspecified, with the understanding that an "implementation"—the software needed to compile, link, and execute programs on a particular platform—will fill in the details. As a result, the behavior of the program may vary somewhat from one implementation to another. The behavior of the `/` and `%` operators for negative operands in C89 is an example of implementation-defined behavior.

Leaving parts of the language unspecified may seem odd or even dangerous, but it reflects C's philosophy. One of the language's goals is efficiency, which often means matching the way that hardware behaves. Some CPUs yield –1 when –9 is divided by 7, while others produce –2; the C89 standard simply reflects this fact of life.

It's best to avoid writing programs that depend on implementation-defined behavior. If that's not possible, at least check the manual carefully—the C standard requires that implementation-defined behavior be documented.

Operator Precedence and Associativity

When an expression contains more than one operator, its interpretation may not be immediately clear. For example, does `i + j * k` mean "add `i` and `j`, then multiply the result by `k`," or does it mean "multiply `j` and `k`, then add `i`"? One solution to this problem is to add parentheses, writing either `(i + j) * k` or `i + (j * k)`. As a general rule, C allows the use of parentheses for grouping in all expressions.

What if we don't use parentheses, though? Will the compiler interpret `i + j * k` as `(i + j) * k` or `i + (j * k)`? Like many other languages, C uses ***operator precedence*** rules to resolve this potential ambiguity. The arithmetic operators have the following relative precedence:

```
Highest:  +   -  (unary)
          *   /   %
Lowest:   +   -  (binary)
```

Operators listed on the same line (such as `+` and `-`) have equal precedence.

When two or more operators appear in the same expression, we can determine how the compiler will interpret the expression by repeatedly putting parentheses around subexpressions, starting with high-precedence operators and working down to low-precedence operators. The following examples illustrate the result:

```
i + j * k     is equivalent to   i + (j * k)
-i * -j       is equivalent to   (-i) * (-j)
+i + j / k    is equivalent to   (+i) + (j / k)
```

Operator precedence rules alone aren't enough when an expression contains two or more operators at the same level of precedence. In this situation, the ***associativity***

of the operators comes into play. An operator is said to be ***left associative*** if it groups from left to right. The binary arithmetic operators (`*`, `/`, `%`, `+`, and `-`) are all left associative, so

```
i - j - k     is equivalent to   (i - j) - k
i * j / k     is equivalent to   (i * j) / k
```

An operator is ***right associative*** if it groups from right to left. The unary arithmetic operators (`+` and `-`) are both right associative, so

```
- + i          is equivalent to   -(+i)
```

Precedence and associativity rules are important in many languages, but especially so in C. However, C has so many operators (almost fifty!) that few programmers bother to memorize the precedence and associativity rules. Instead, they consult a table of operators when in doubt or just use plenty of parentheses.

table of operators ➤*Appendix A*

PROGRAM Computing a UPC Check Digit

For a number of years, manufacturers of goods sold in U.S. and Canadian stores have put a bar code on each product. This code, known as a Universal Product Code (UPC), identifies both the manufacturer and the product. Each bar code represents a twelve-digit number, which is usually printed underneath the bars. For example, the following bar code comes from a package of Stouffer's French Bread Pepperoni Pizza:

The digits

```
0  13800 15173  5
```

appear underneath the bar code. The first digit identifies the type of item (`0` or `7` for most items, `2` for items that must be weighed, `3` for drugs and health-related merchandise, and `5` for coupons). The first group of five digits identifies the manufacturer (13800 is the code for Nestlé USA's Frozen Food Division). The second group of five digits identifies the product (including package size). The final digit is a "check digit," whose only purpose is to help identify an error in the preceding digits. If the UPC is scanned incorrectly, the first 11 digits probably won't be consistent with the last digit, and the store's scanner will reject the entire code.

Here's one method of computing the check digit:

Add the first, third, fifth, seventh, ninth, and eleventh digits.
Add the second, fourth, sixth, eighth, and tenth digits.

Multiply the first sum by 3 and add it to the second sum.
Subtract 1 from the total.
Compute the remainder when the adjusted total is divided by 10.
Subtract the remainder from 9.

Using the Stouffer's example, we get 0 + 3 + 0 + 1 + 1 + 3 = 8 for the first sum and 1 + 8 + 0 + 5 + 7 = 21 for the second sum. Multiplying the first sum by 3 and adding the second yields 45. Subtracting 1 gives 44. The remainder upon dividing by 10 is 4. When the remainder is subtracted from 9, the result is 5. Here are a couple of other UPCs, in case you want to try your hand at computing the check digit (raiding the kitchen cabinet for the answer is *not* allowed):

Jif Creamy Peanut Butter (18 oz.): 0 51500 24128 ?
Ocean Spray Jellied Cranberry Sauce (8 oz.): 0 31200 01005 ?

The answers appear at the bottom of the page.

Let's write a program that calculates the check digit for an arbitrary UPC. We'll ask the user to enter the first 11 digits of the UPC, then we'll display the corresponding check digit. To avoid confusion, we'll ask the user to enter the number in three parts: the single digit at the left, the first group of five digits, and the second group of five digits. Here's what a session with the program will look like:

```
Enter the first (single) digit: 0
Enter first group of five digits: 13800
Enter second group of five digits: 15173
Check digit: 5
```

Instead of reading each digit group as a *five*-digit number, we'll read it as five *one*-digit numbers. Reading the numbers as single digits is more convenient; also, we won't have to worry that one of the five-digit numbers is too large to store in an `int` variable. (Some older compilers limit the maximum value of an `int` variable to 32,767.) To read single digits, we'll use `scanf` with the `%1d` conversion specification, which matches a one-digit integer.

upc.c

```
/* Computes a Universal Product Code check digit */

#include <stdio.h>

int main(void)
{
  int d, i1, i2, i3, i4, i5, j1, j2, j3, j4, j5,
      first_sum, second_sum, total;

  printf("Enter the first (single) digit: ");
  scanf("%1d", &d);
  printf("Enter first group of five digits: ");
  scanf("%1d%1d%1d%1d%1d", &i1, &i2, &i3, &i4, &i5);
  printf("Enter second group of five digits: ");
  scanf("%1d%1d%1d%1d%1d", &j1, &j2, &j3, &j4, &j5);
```

The missing check digits are 8 (Jif) and 6 (Ocean Spray).

```
  first_sum = d + i2 + i4 + j1 + j3 + j5;
  second_sum = i1 + i3 + i5 + j2 + j4;
  total = 3 * first_sum + second_sum;

  printf("Check digit: %d\n", 9 - ((total - 1) % 10));

  return 0;
}
```

Note that the expression `9 - ((total - 1) % 10)` could have been written as `9 - (total - 1) % 10`, but the extra set of parentheses makes it easier to understand.

4.2 Assignment Operators

Once the value of an expression has been computed, we'll often need to store it in a variable for later use. C's = (***simple assignment***) operator is used for that purpose. For updating a value already stored in a variable, C provides an assortment of compound assignment operators.

Simple Assignment

The effect of the assignment *v* = *e* is to evaluate the expression *e* and copy its value into *v*. As the following examples show, *e* can be a constant, a variable, or a more complicated expression:

```
i = 5;              /* i is now 5  */
j = i;              /* j is now 5  */
k = 10 * i + j;     /* k is now 55 */
```

If *v* and *e* don't have the same type, then the value of *e* is converted to the type of *v* as the assignment takes place:

```
int i;
float f;

i = 72.99f;    /* i is now 72 */
f = 136;       /* f is now 136.0 */
```

conversion during assignment ➤7.4

We'll return to the topic of type conversion later.

In many programming languages, assignment is a *statement*; in C, however, assignment is an *operator*, just like +. In other words, the act of assignment produces a result, just as adding two numbers produces a result. The value of an assignment *v* = *e* is the value of *v after* the assignment. Thus, the value of `i = 72.99f` is 72 (not 72.99).

Side Effects

We don't normally expect operators to modify their operands, since operators in mathematics don't. Writing `i + j` doesn't modify either `i` or `j`; it simply computes the result of adding `i` and `j`.

Most C operators don't modify their operands, but some do. We say that these operators have ***side effects***, since they do more than just compute a value. The simple assignment operator is the first operator we've seen that has side effects; it modifies its left operand. Evaluating the expression `i = 0` produces the result 0 and—as a side effect—assigns 0 to `i`.

Since assignment is an operator, several assignments can be chained together:

```
i = j = k = 0;
```

The = operator is right associative, so this assignment is equivalent to

```
i = (j = (k = 0));
```

The effect is to assign 0 first to `k`, then to `j`, and finally to `i`.

Watch out for unexpected results in chained assignments as a result of type conversion:

```
int i;
float f;

f = i = 33.3f;
```

`i` is assigned the value 33, then `f` is assigned 33.0 (not 33.3, as you might think).

In general, an assignment of the form *v* = *e* is allowed wherever a value of type *v* would be permitted. In the following example, the expression `j = i` copies `i` to `j`; the new value of `j` is then added to `1`, producing the new value of `k`:

```
i = 1;
k = 1 + (j = i);
printf("%d %d %d\n", i, j, k);   /* prints "1 1 2" */
```

Using the assignment operator in this fashion usually isn't a good idea. For one thing, "embedded assignments" can make programs hard to read. They can also be a source of subtle bugs, as we'll see in Section 4.4.

Lvalues

Q&A

Most C operators allow their operands to be variables, constants, or expressions containing other operators. The assignment operator, however, requires an ***lvalue***

as its left operand. An lvalue (pronounced "L-value") represents an object stored in computer memory, not a constant or the result of a computation. Variables are lvalues; expressions such as `10` or `2 * i` are not. At this point, variables are the only lvalues that we know about; other kinds of lvalues will appear in later chapters.

Since the assignment operator requires an lvalue as its left operand, it's illegal to put any other kind of expression on the left side of an assignment expression:

```
12 = i;       /*** WRONG ***/
i + j = 0;    /*** WRONG ***/
-i = j;       /*** WRONG ***/
```

The compiler will detect errors of this nature, and you'll get an error message such as *"invalid lvalue in assignment."*

Compound Assignment

Assignments that use the old value of a variable to compute its new value are common in C programs. The following statement, for example, adds 2 to the value stored in `i`:

```
i = i + 2;
```

C's ***compound assignment*** operators allow us to shorten this statement and others like it. Using the += operator, we simply write:

```
i += 2;   /* same as i = i + 2; */
```

The += operator adds the value of the right operand to the variable on the left.

There are nine other compound assignment operators, including the following:

```
-=  *=  /=  %=
```

other assignment operators ➤20.1

(We'll cover the remaining compound assignment operators in a later chapter.) All compound assignment operators work in much the same way:

v += *e* adds *v* to *e*, storing the result in *v*
v -= *e* subtracts *e* from *v*, storing the result in *v*
v *= *e* multiplies *v* by *e*, storing the result in *v*
v /= *e* divides *v* by *e*, storing the result in *v*
v %= *e* computes the remainder when *v* is divided by *e*, storing the result in *v*

Note that I've been careful not to say that *v* += *e* is "equivalent" to *v* = *v* + *e*. One problem is operator precedence: `i *= j + k` isn't the same as `i = i * j + k`. There are also rare cases in which *v* += *e* differs from *v* = *v* + *e* because *v* itself has a side effect. Similar remarks apply to the other compound assignment operators.

Q&A

When using the compound assignment operators, be careful not to switch the two characters that make up the operator. Switching the characters may yield an expression that is acceptable to the compiler but that doesn't have the intended meaning. For example, if you meant to write `i += j` but typed `i =+ j` instead, the

program will still compile. Unfortunately, the latter expression is equivalent to `i = (+j)`, which merely copies the value of `j` into `i`.

The compound assignment operators have the same properties as the = operator. In particular, they're right associative, so the statement

```
i += j += k;
```

means

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

4.3 Increment and Decrement Operators

Two of the most common operations on a variable are "incrementing" (adding 1) and "decrementing" (subtracting 1). We can, of course, accomplish these tasks by writing

```
i = i + 1;
j = j - 1;
```

The compound assignment operators allow us to condense these statements a bit:

```
i += 1;
j -= 1;
```

But C allows increments and decrements to be shortened even further, using the ++ (***increment***) and -- (***decrement***) operators.

Q&A

At first glance, the increment and decrement operators are simplicity itself: ++ adds 1 to its operand, whereas -- subtracts 1. Unfortunately, this simplicity is misleading—the increment and decrement operators can be tricky to use. One complication is that ++ and -- can be used as ***prefix*** operators (`++i` and `--i`, for example) or ***postfix*** operators (`i++` and `i--`). The correctness of a program may hinge on picking the proper version.

Another complication is that, like the assignment operators, ++ and -- have side effects: they modify the values of their operands. Evaluating the expression `++i` (a "pre-increment") yields `i + 1` and—as a side effect—increments `i`:

```
i = 1;
printf("i is %d\n", ++i);   /* prints "i is 2" */
printf("i is %d\n", i);     /* prints "i is 2" */
```

Evaluating the expression `i++` (a "post-increment") produces the result `i`, but causes `i` to be incremented afterwards:

```
i = 1;
printf("i is %d\n", i++);   /* prints "i is 1" */
printf("i is %d\n", i);     /* prints "i is 2" */
```

The first `printf` shows the original value of `i`, before it is incremented. The second `printf` shows the new value. As these examples illustrate, `++i` means "increment `i` immediately," while `i++` means "use the old value of `i` for now, but increment `i` later." How much later? The C standard doesn't specify a precise time, but it's safe to assume that `i` will be incremented before the next statement is executed.

Q&A

The `--` operator has similar properties:

```
i = 1;
printf("i is %d\n", --i);   /* prints "i is 0" */
printf("i is %d\n", i);     /* prints "i is 0" */

i = 1;
printf("i is %d\n", i--);   /* prints "i is 1" */
printf("i is %d\n", i);     /* prints "i is 0" */
```

When `++` or `--` is used more than once in the same expression, the result can often be hard to understand. Consider the following statements:

```
i = 1;
j = 2;
k = ++i + j++;
```

What are the values of `i`, `j`, and `k` after these statements are executed? Since `i` is incremented *before* its value is used, but `j` is incremented *after* it is used, the last statement is equivalent to

```
i = i + 1;
k = i + j;
j = j + 1;
```

so the final values of `i`, `j`, and `k` are 2, 3, and 4, respectively. In contrast, executing the statements

```
i = 1;
j = 2;
k = i++ + j++;
```

will give `i`, `j`, and `k` the values 2, 3, and 3, respectively.

For the record, the postfix versions of `++` and `--` have higher precedence than unary plus and minus and are left associative. The prefix versions have the same precedence as unary plus and minus and are right associative.

4.4 Expression Evaluation

Table 4.2 summarizes the operators we've seen so far. (Appendix A has a similar table that shows *all* operators.) The first column shows the precedence of each

Table 4.2
A Partial List of C Operators

Precedence	*Name*	*Symbol(s)*	*Associativity*
1	increment (postfix) decrement (postfix)	`++` `--`	left

2	increment (prefix)
	decrement (prefix)
	unary plus
	unary minus
		`++`
		`--`
		`+`
		`-`
			right

3	multiplicative	`* / %`	left
4	additive	`+ -`	left
5	assignment	`= *= /= %= += -=`	right

operator relative to the other operators in the table (the highest precedence is 1; the lowest is 5). The last column shows the associativity of each operator.

Table 4.2 (or its larger cousin in Appendix A) has a variety of uses. Let's look at one of these. Suppose that we run across a complicated expression such as

```
a = b += c++ - d + --e / -f
```

as we're reading someone's program. This expression would be easier to understand if there were parentheses to show how the expression is constructed from subexpressions. With the help of Table 4.2, adding parentheses to an expression is easy: after examining the expression to find the operator with highest precedence, we put parentheses around the operator and its operands, indicating that it should be treated as a single operand from that point onwards. We then repeat the process until the expression is fully parenthesized.

In our example, the operator with highest precedence is `++`, used here as a postfix operator, so we put parentheses around `++` and its operand:

```
a = b += (c++) - d + --e / -f
```

We now spot a prefix `--` operator and a unary minus operator (both precedence 2) in the expression:

```
a = b += (c++) - d + (--e) / (-f)
```

Note that the other minus sign has an operand to its immediate left, so it must be a subtraction operator, not a unary minus operator.

Next, we notice the `/` operator (precedence 3):

```
a = b += (c++) - d + ((--e) / (-f))
```

The expression contains two operators with precedence 4, subtraction and addition. Whenever two operators with the same precedence are adjacent to an operand, we've got to be careful about associativity. In our example, `-` and `+` are both adjacent to `d`, so associativity rules apply. The `-` and `+` operators group from left to right, so parentheses go around the subtraction first, then the addition:

```
a = b += (((c++) - d) + ((--e) / (-f)))
```

The only remaining operators are = and +=. Both operators are adjacent to b, so we must take associativity into account. Assignment operators group from right to left, so parentheses go around the += expression first, then the = expression:

```
(a = (b += (((c++) - d) + ((--e) / (-f)))))
```

The expression is now fully parenthesized.

Order of Subexpression Evaluation

The rules of operator precedence and associativity allow us to break any C expression into subexpressions—to determine uniquely where the parentheses would go if the expression were fully parenthesized. Paradoxically, these rules don't always allow us to determine the value of the expression, which may depend on the order in which its subexpressions are evaluated.

logical *and* and *or* operators ➤*5.1*
conditional operator ➤*5.2*
comma operator ➤*6.3*

C doesn't define the order in which subexpressions are evaluated (with the exception of subexpressions involving the logical *and*, logical *or*, conditional, and comma operators). Thus, in the expression `(a + b) * (c - d)` we don't know whether `(a + b)` will be evaluated before `(c - d)`.

Most expressions have the same value regardless of the order in which their subexpressions are evaluated. However, this may not be true when a subexpression modifies one of its operands. Consider the following example:

```
a = 5;
c = (b = a + 2) - (a = 1);
```

The effect of executing the second statement is undefined; the C standard doesn't say what will happen. With most compilers, the value of c will be either 6 or 2. If the subexpression `(b = a + 2)` is evaluated first, b is assigned the value 7 and c is assigned 6. But if `(a = 1)` is evaluated first, b is assigned 3 and c is assigned 2.

Avoid writing expressions that access the value of a variable and also modify the variable elsewhere in the expression. The expression `(b = a + 2) - (a = 1)` accesses the value of a (in order to compute `a + 2`) and also modifies the value of a (by assigning it 1). Some compilers may produce a warning message such as "*operation on 'a' may be undefined*" when they encounter such an expression.

To prevent problems, it's a good idea to avoid using the assignment operators in subexpressions; instead, use a series of separate assignments. For example, the statements above could be rewritten as

```
a = 5;
b = a + 2;
a = 1;
c = b - a;
```

The value of c will always be 6 after these statements are executed.

Besides the assignment operators, the only operators that modify their operands are increment and decrement. When using these operators, be careful that your expressions don't depend on a particular order of evaluation. In the following example, `j` may be assigned one of two values:

```
i = 2;
j = i * i++;
```

It's natural to assume that `j` is assigned the value 4. However, the effect of executing the statement is undefined, and `j` could just as well be assigned 6 instead. Here's the scenario: (1) The second operand (the original value of `i`) is fetched, then `i` is incremented. (2) The first operand (the new value of `i`) is fetched. (3) The new and old values of `i` are multiplied, yielding 6. "Fetching" a variable means to retrieve the value of the variable from memory. A later change to the variable won't affect the fetched value, which is typically stored in a special location (known as a ***register***) inside the CPU.

registers ➤ 18.2

Undefined Behavior

According to the C standard, statements such as `c = (b = a + 2) - (a = 1);` and `j = i * i++;` cause ***undefined behavior,*** which is different from implementation-defined behavior (see Section 4.1). When a program ventures into the realm of undefined behavior, all bets are off. The program may behave differently when compiled with different compilers. But that's not the only thing that can happen. The program may not compile in the first place, if it compiles it may not run, and if it does run, it may crash, behave erratically, or produce meaningless results. In other words, undefined behavior should be avoided like the plague.

4.5 Expression Statements

C has the unusual rule that *any* expression can be used as a statement. That is, any expression—regardless of its type or what it computes—can be turned into a statement by appending a semicolon. For example, we could turn the expression `++i` into a statement:

```
++i;
```

When this statement is executed, `i` is first incremented, then the new value of `i` is fetched (as though it were to be used in an enclosing expression). However, since `++i` isn't part of a larger expression, its value is discarded and the next statement executed. (The change to `i` is permanent, of course.)

Q&A

Since its value is discarded, there's little point in using an expression as a statement unless the expression has a side effect. Let's look at three examples. In

the first example, 1 is stored into `i`, then the new value of `i` is fetched but not used:

```
i = 1;
```

In the second example, the value of `i` is fetched but not used; however, `i` is decremented afterwards:

```
i--;
```

In the third example, the value of the expression `i * j - 1` is computed and then discarded:

```
i * j - 1;
```

Since `i` and `j` aren't changed, this statement has no effect and therefore serves no purpose.

A slip of the finger can easily create a "do-nothing" expression statement. For example, instead of entering

```
i = j;
```

we might accidentally type

```
i + j;
```

(This kind of error is more common than you might expect, since the = and + characters usually occupy the same key.) Some compilers can detect meaningless expression statements; you'll get a warning such as *"statement with no effect."*

Q & A

Q: I notice that C has no exponentiation operator. How can I raise a number to a power?

A: Raising an integer to a small positive integer power is best done by repeated multiplication (`i * i * i` is `i` cubed). To raise a number to a noninteger power, call the `pow` function.

pow function ➤23.3

Q: I want to apply the `%` operator to a floating-point operand, but my program won't compile. What can I do? [p. 54]

fmod function ➤23.3

A: The `%` operator requires integer operands. Try the `fmod` function instead.

Q: Why are the rules for using the `/` and `%` operators with negative operands so complicated? [p. 54]

A: The rules aren't as complicated as they may first appear. In both C89 and C99, the goal is to ensure that the value of `(a / b) * b + a % b` will always be equal to `a`

(and indeed, both standards guarantee that this is the case, provided that the value of `a / b` is "representable"). The problem is that there are two ways for `a / b` and `a % b` to satisfy this equality if either `a` or `b` is negative, as seen in C89, where either `-9 / 7` is –1 and `-9 % 7` is –2, or `-9 / 7` is –2 and `-9 % 7` is 5. In the first case, `(-9 / 7) * 7 + -9 % 7` has the value $-1 \times 7 + -2 = -9$, and in the second case, `(-9 / 7) * 7 + -9 % 7` has the value $-2 \times 7 + 5 = -9$. By the time C99 rolled around, most CPUs were designed to truncate the result of division toward zero, so this was written into the standard as the only allowable outcome.

C99

Q: If C has lvalues, does it also have rvalues? [p. 59]

A: Yes, indeed. An *l*value is an expression that can appear on the *left* side of an assignment; an *r*value is an expression that can appear on the *right* side. Thus, an rvalue could be a variable, constant, or more complex expression. In this book, as in the C standard, we'll use the term "expression" instead of "rvalue."

***Q: You said that *v* += *e* isn't equivalent to *v* = *v* + *e* if *v* has a side effect. Can you explain? [p. 60]**

A: Evaluating *v* += *e* causes *v* to be evaluated only once; evaluating *v* = *v* + *e* causes *v* to be evaluated twice. Any side effect caused by evaluating *v* will occur twice in the latter case. In the following example, `i` is incremented once:

```
a[i++] += 2;
```

If we use = instead of +=, here's what the statement will look like:

```
a[i++] = a[i++] + 2;
```

The value of `i` is modified as well as used elsewhere in the statement, so the effect of executing the statement is undefined. It's likely that `i` will be incremented twice, but we can't say with certainty what will happen.

Q: Why does C provide the `++` and `--` operators? Are they faster than other ways of incrementing and decrementing, or they are just more convenient? [p. 61]

A: C inherited `++` and `--` from Ken Thompson's earlier B language. Thompson apparently created these operators because his B compiler could generate a more compact translation for `++i` than for `i = i + 1`. These operators have become a deeply ingrained part of C (in fact, many of C's most famous idioms rely on them). With modern compilers, using `++` and `--` won't make a compiled program any smaller or faster; the continued popularity of these operators stems mostly from their brevity and convenience.

Q: Do `++` and `--` work with `float` variables?

A: Yes; the increment and decrement operations can be applied to floating-point numbers as well as integers. In practice, however, it's fairly rare to increment or decrement a `float` variable.

***Q: When I use the postfix version of ++ or --, just when is the increment or decrement performed? [p. 62]**

A: That's an excellent question. Unfortunately, it's also a difficult one to answer. The C standard introduces the concept of "sequence point" and says that "updating the stored value of the operand shall occur between the previous and the next sequence point." There are various kinds of sequence points in C; the end of an expression statement is one example. By the end of an expression statement, all increments and decrements within the statement must have been performed; the next statement can't begin to execute until this condition has been met.

Certain operators that we'll encounter in later chapters (logical *and*, logical *or*, conditional, and comma) also impose sequence points. So do function calls: the arguments in a function call must be fully evaluated before the call can be performed. If an argument happens to be an expression containing a ++ or -- operator, the increment or decrement must occur before the call can take place.

Q: What do you mean when you say that the value of an expression statement is discarded? [p. 65]

A: By definition, an expression represents a value. If `i` has the value 5, for example, then evaluating `i + 1` produces the value 6. Let's turn `i + 1` into a statement by putting a semicolon after it:

```
i + 1;
```

When this statement is executed, the value of `i + 1` is computed. Since we have failed to save this value—or at least use it in some way—it is lost.

Q: But what about statements like `i = 1;`? I don't see what is being discarded.

A: Don't forget that = is an operator in C and produces a value just like any other operator. The assignment

```
i = 1;
```

assigns 1 to `i`. The value of the entire expression is 1, which is discarded. Discarding the expression's value is no great loss, since the reason for writing the statement in the first place was to modify `i`.

Exercises

Section 4.1

1. Show the output produced by each of the following program fragments. Assume that `i`, `j`, and `k` are `int` variables.

(a)
```
i = 5; j = 3;
printf("%d %d", i / j, i % j);
```

(b)
```
i = 2; j = 3;
printf("%d", (i + 10) % j);
```

(c)
```
i = 7; j = 8; k = 9;
printf("%d", (i + 10) % k / j);
```

(d)
```
i = 1; j = 2; k = 3;
printf("%d", (i + 5) % (j + 2) / k);
```

W *2. If `i` and `j` are positive integers, does `(-i)/j` always have the same value as `-(i/j)`? Justify your answer.

3. What is the value of each of the following expressions in C89? (Give all possible values if an expression may have more than one value.)

   (a) `8 / 5`
   (b) `-8 / 5`
   (c) `8 / -5`
   (d) `-8 / -5`

4. Repeat Exercise 3 for C99.

5. What is the value of each of the following expressions in C89? (Give all possible values if an expression may have more than one value.)

   (a) `8 % 5`
   (b) `-8 % 5`
   (c) `8 % -5`
   (d) `-8 % -5`

6. Repeat Exercise 5 for C99.

7. The algorithm for computing the UPC check digit ends with the following steps:

   Subtract 1 from the total.
   Compute the remainder when the adjusted total is divided by 10.
   Subtract the remainder from 9.

   It's tempting to try to simplify the algorithm by using these steps instead:

   Compute the remainder when the total is divided by 10.
   Subtract the remainder from 10.

   Why doesn't this technique work?

8. Would the `upc.c` program still work if the expression `9 - ((total - 1) % 10)` were replaced by `(10 - (total % 10)) % 10`?

Section 4.2

W 9. Show the output produced by each of the following program fragments. Assume that `i`, `j`, and `k` are `int` variables.

(a)
```
i = 7; j = 8;
i *= j + 1;
printf("%d %d", i, j);
```
(b)
```
i = j = k = 1;
i += j += k;
printf("%d %d %d", i, j, k);
```
(c)
```
i = 1; j = 2; k = 3;
i -= j -= k;
printf("%d %d %d", i, j, k);
```
(d)
```
i = 2; j = 1; k = 0;
i *= j *= k;
printf("%d %d %d", i, j, k);
```

10. Show the output produced by each of the following program fragments. Assume that `i` and `j` are `int` variables.

(a)
```
i = 6;
j = i += i;
printf("%d %d", i, j);
```

(b)
```
i = 5;
j = (i -= 2) + 1;
printf("%d %d", i, j);
```

(c)
```
i = 7;
j = 6 + (i = 2.5);
printf("%d %d", i, j);
```

(d)
```
i = 2; j = 8;
j = (i = 6) + (j = 3);
printf("%d %d", i, j);
```

Section 4.3

*11. Show the output produced by each of the following program fragments. Assume that `i`, `j`, and `k` are `int` variables.

(a)
```
i = 1;
printf("%d ", i++ - 1);
printf("%d", i);
```

(b)
```
i = 10; j = 5;
printf("%d ", i++ - ++j);
printf("%d %d", i, j);
```

(c)
```
i = 7; j = 8;
printf("%d ", i++ - --j);
printf("%d %d", i, j);
```

(d)
```
i = 3; j = 4; k = 5;
printf("%d ", i++ - j++ + --k);
printf("%d %d %d", i, j, k);
```

12. Show the output produced by each of the following program fragments. Assume that `i` and `j` are `int` variables.

(a)
```
i = 5;
j = ++i * 3 - 2;
printf("%d %d", i, j);
```

(b)
```
i = 5;
j = 3 - 2 * i++;
printf("%d %d", i, j);
```

(c)
```
i = 7;
j = 3 * i-- + 2;
printf("%d %d", i, j);
```

(d)
```
i = 7;
j = 3 + --i * 2;
printf("%d %d", i, j);
```

W 13. Only one of the expressions `++i` and `i++` is exactly the same as `(i += 1)`; which is it? Justify your answer.

Section 4.4

14. Supply parentheses to show how a C compiler would interpret each of the following expressions.

(a) `a * b - c * d + e`
(b) `a / b % c / d`
(c) `- a - b + c - + d`
(d) `a * - b / c - d`

Section 4.5

15. Give the values of `i` and `j` after each of the following expression statements has been executed. (Assume that `i` has the value 1 initially and `j` has the value 2.)

(a) `i += j;`
(b) `i--;`
(c) `i * j / i;`
(d) `i % ++j;`

Programming Projects

1. Write a program that asks the user to enter a two-digit number, then prints the number with its digits reversed. A session with the program should have the following appearance:

```
Enter a two-digit number: 28
The reversal is: 82
```

Read the number using `%d`, then break it into two digits. *Hint:* If `n` is an integer, then `n % 10` is the last digit in `n` and `n / 10` is `n` with the last digit removed.

W 2. Extend the program in Programming Project 1 to handle *three*-digit numbers.

3. Rewrite the program in Programming Project 2 so that it prints the reversal of a three-digit number without using arithmetic to split the number into digits. *Hint:* See the `upc.c` program of Section 4.1.

4. Write a program that reads an integer entered by the user and displays it in octal (base 8):

```
Enter a number between 0 and 32767: 1953
In octal, your number is: 03641
```

The output should be displayed using five digits, even if fewer digits are sufficient. *Hint:* To convert the number to octal, first divide it by 8; the remainder is the last digit of the octal number (1, in this case). Then divide the original number by 8 and repeat the process to arrive at the next-to-last digit. (`printf` is capable of displaying numbers in base 8, as we'll see in Chapter 7, so there's actually an easier way to write this program.)

5. Rewrite the `upc.c` program of Section 4.1 so that the user enters 11 digits at one time, instead of entering one digit, then five digits, and then another five digits.

```
Enter the first 11 digits of a UPC: 01380015173
Check digit: 5
```

6. European countries use a 13-digit code, known as a European Article Number (EAN) instead of the 12-digit Universal Product Code (UPC) found in North America. Each EAN ends with a check digit, just as a UPC does. The technique for calculating the check digit is also similar:

Add the second, fourth, sixth, eighth, tenth, and twelfth digits.
Add the first, third, fifth, seventh, ninth, and eleventh digits.
Multiply the first sum by 3 and add it to the second sum.

Subtract 1 from the total.
Compute the remainder when the adjusted total is divided by 10.
Subtract the remainder from 9.

For example, consider Güllüoglu Turkish Delight Pistachio & Coconut, which has an EAN of 8691484260008. The first sum is 6 + 1 + 8 + 2 + 0 + 0 = 17, and the second sum is 8 + 9 + 4 + 4 + 6 + 0 = 31. Multiplying the first sum by 3 and adding the second yields 82. Subtracting 1 gives 81. The remainder upon dividing by 10 is 1. When the remainder is subtracted from 9, the result is 8, which matches the last digit of the original code. Your job is to modify the `upc.c` program of Section 4.1 so that it calculates the check digit for an EAN. The user will enter the first 12 digits of the EAN as a single number:

```
Enter the first 12 digits of an EAN: 869148426000
Check digit: 8
```