Chapter 5. Operators, Expressions, and Statementsa

Chapter Outline

- Keyword:

  while, typedef

- Operators:

  = - * /

  % ++ -- *(type)*

Chapter Outline

- C's multitudinous operators, including those used for common arithmetic operations
- Operator precedence and the meanings of the terms statement and expression
- The handy while loop
- Compound statements, automatic type conversions, and type casts
- How to write functions that use arguments

Chapter Outline

- Now that you've looked at ways to represent data, let's **explore ways to process data**.
- C offers **a wealth of operations for that purpose**.
- You can do arithmetic, compare values, modify variables, combine relationships logically, and more.
- Let's start with basic arithmetic—addition, subtraction, multiplication, and division.

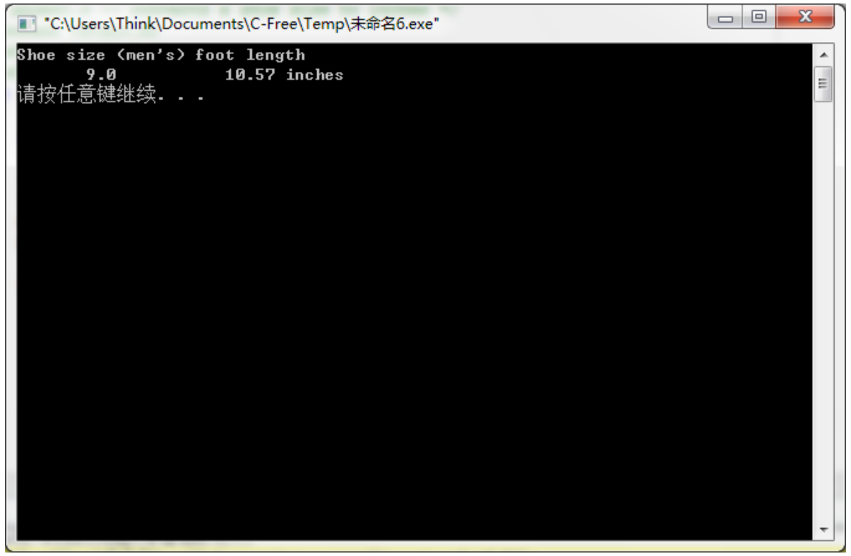
Chapter Outline

- Another aspect of processing data is **organizing your programs** so that they take the right steps in the right order.
- C has several language features to help you with that task.
- One of these features is the loop, and in this chapter you get a first look at it.
- A loop enables you to repeat actions and makes your programs more interesting and powerful.

Introducing Loops

Listing 5.1. The shoes1.c Program

```
/* shoes1.c -- converts a shoe size to inches */
#include <stdio.h>
#define ADJUST 7.64
#define SCALE 0.325
int main(void)
{
        double shoe, foot;

        shoe = 9.0;
        foot = SCALE * shoe + ADJUST;
        printf("Shoe size (men's) foot length\n");
        printf("%10.1f %15.2f inches\n", shoe, foot);

        return 0;
}
```



"C:\Users\Think\Documents\C-Free\Temp\未命名6.exe"

```
Shoe size (men's) foot length
       9.0           10.57 inches
请按任意键继续. . .
```

The Limitations of This Program

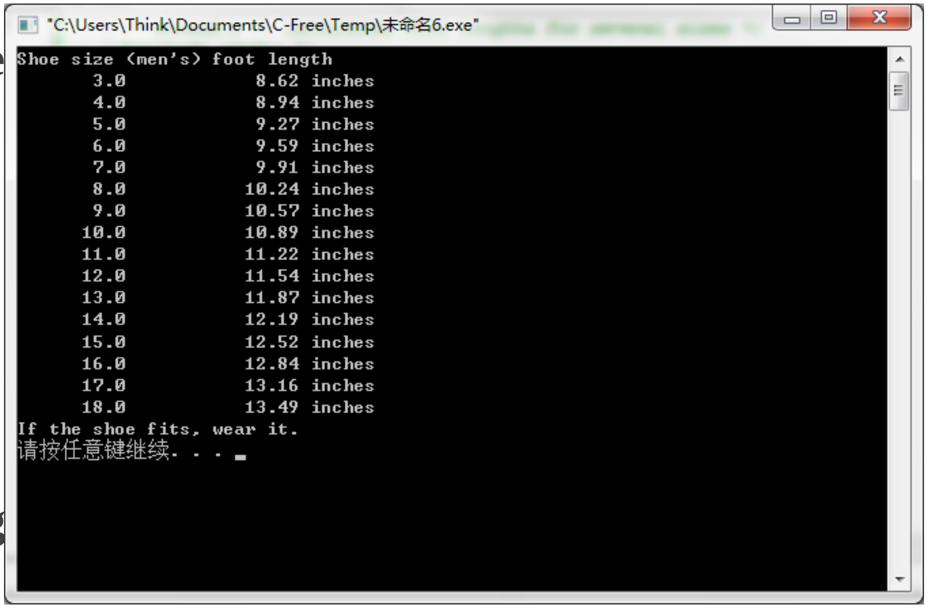
- Here is a program with multiplication and addition.
- It takes your shoe size (if you wear a size 9) and tells how long your foot is in inches.
- "But," you say, "I could solve this problem by hand more quickly than you could type the program."
- That's a good point.
- A one-shot program that does just one shoe size is a waste of time and effort.
- You could make the program more useful by writing it as an interactive program, but that still barely taps the potential of a computer.

The Limitations of This Program

- What you need is some way to have a computer do repetitive calculations for a succession of shoe sizes.
- After all, that's one of the main reasons for using a computer to **do arithmetic**.
- C offers several methods for **doing repetitive calculations**, and we will outline one here.
- This method, called **a while loop**, will enable you to make a more interesting exploration of operations.

Listing 5.2. The shoes2.c Program

```
/* shoes2.c -- calculates foot lengths for seve
#include <stdio.h>
#define ADJUST 7.64
#define SCALE 0.325
int main(void)
{
        double shoe, foot;

        printf("Shoe size (men's) foot leng
        shoe = 3.0;
        while (shoe < 18.5)    /* starting the while loop */
        {                      /* start of block */
                foot = SCALE*shoe + ADJUST;
                printf("%10.1f %15.2f inches\n", shoe, foot);
                shoe = shoe + 1.0;
        }                      /* end of block */
        printf("If the shoe fits, wear it. \n");

        return 0;
}
```



```
Shoe size (men's) foot length
       3.0            8.62 inches
       4.0            8.94 inches
       5.0            9.27 inches
       6.0            9.59 inches
       7.0            9.91 inches
       8.0           10.24 inches
       9.0           10.57 inches
      10.0           10.89 inches
      11.0           11.22 inches
      12.0           11.54 inches
      13.0           11.87 inches
      14.0           12.19 inches
      15.0           12.52 inches
      16.0           12.84 inches
      17.0           13.16 inches
      18.0           13.49 inches
If the shoe fits, wear it.
请按任意键继续. . .
```

The Main New Features of This Program

- Here is how the while loop works.
- When the program first reaches the while statement, it checks to **see whether the condition within parentheses is true**.
- In this case, the expression is as follows:

  shoe < 18.5
- The < symbol means "is less than." The variable shoe was initialized to 3.0, which certainly is less than 18.5.
- Therefore, the condition is true and the program proceeds to the next statement, which converts the size to inches. Then it prints the results.

The Main New Features of This Program

- The next statement increases shoe by 1.0, making it 4.0: *shoe = shoe + 1.0;*
- At this point, the program returns to the while portion to check the condition. **Why at this point?**
- Because the next line is **a closing brace (})**, and the code uses a set of braces ({}) to mark the extent of the while loop. The statements between the two braces are the ones that are repeated.

The Main New Features of This Program

- The section of program between and including the braces is called **a block**.
- Now back to the program. The value 4 is less than 18.5, so **the whole cycle of embraced commands (the block) following the while is repeated**.
- In computerese, the program is said to "**loop**" through these statements.

The Main New Features of This Program

- This continues until shoe reaches a value of 19.0.
- Now the condition

  shoe < 18.5

  becomes false because 19.0 is not less than 18.5.
- When this happens, control **passes to the first statement following the while loop**.
- In this case, that is the final printf() statement.
- You can easily modify this program to do other conversions.

Fundamental Operators

- C uses operators to represent arithmetic operations.
- Now take a look at the operators used for basic arithmetic: =, +, -, *, and /.
- C does not have an exponentiating operator.
- The standard C math library, however, provides the **pow() function** for that purpose.
- For example, pow(3.5, 2.2) returns 3.5 raised to the power of 2.2.

Assignment Operator: =

- In C, **the equal sign does not mean "equals."**
- Rather, it is **a value-assigning operator**.
- The statement *bmw = 2002;*

  assigns the value 2002 to the variable named bmw.
- That is, the item to the left of the = sign is the name of a variable, and the item on the right is the value assigned to the variable.
- The = symbol is called **the assignment operator**.
- Again, don't think of the line as saying, "bmw equals 2002." Instead, read it as "assign the value 2002 to the

The action goes from right to left for this operator.

Assignment Operator: =

- Perhaps this distinction between the name of a variable and the value of a variable seems like hair-splitting, but consider the following common type of computer statement:

  i = i + 1;
- As mathematics, this statement makes no sense.
- If you add 1 to a finite number, the result isn't "equal to" the number you started with, but **as a computer assignment statement**, **it is perfectly reasonable**.
- It means "**Find the value of the variable named i**, **add 1 to that value**, and **then assign this new value to the variable i**"

Assignment Operator: =

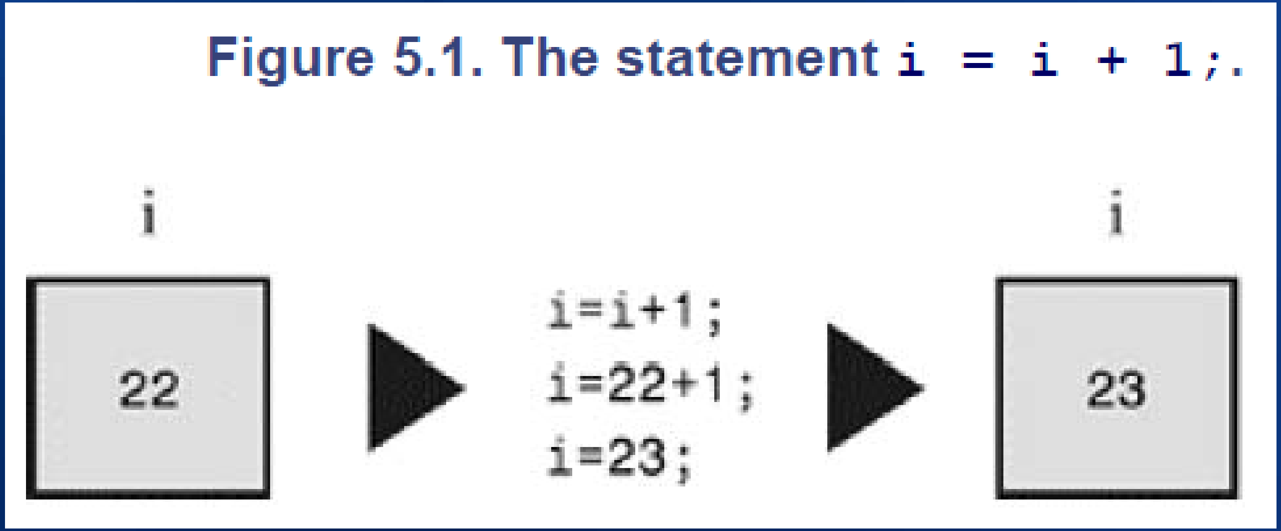
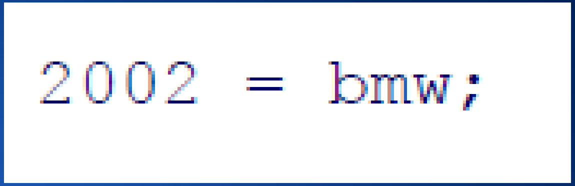


Figure 5.1. The statement `i = i + 1;`.

Assignment Operator: =

- When you sit down at the keyboard remember that **the item to the left of the = sign must be the name of a variable**.
- Actually, **the left side** must refer to **a storage location**. The simplest way is to **use the name of a variable**, but, as you will see later, a "**pointer**" can be used to point to a location.
- More generally, C uses the term **modifiable lvalue** to label those entities to which you can assign values.



```
2002 = bmw;
```

Assignment Operator: =

- "Modifiable lvalue" is not, perhaps, the most intuitive phrase you've encountered, so let's look at some definitions.
- Some Terminology: **Data Objects**, **Lvalues**, **Rvalues**, and **Operands**
- **Data object** is a general term for a region of data storage that can be used to hold values. **The data storage** used to hold **a variable** or **an array** is a data object, for instance.

Assignment Operator: =

- C uses the term **lvalue** to mean **a name** or **expression** that identifies a **particular** data object.
- The name of a variable, for instance, is an lvalue, so **object refers to the actual data storage**, but **lvalue is a label used to identify, or locate, that storage**.

Assignment Operator: =

- **Not all objects can have their values changed**, so C uses the term **modifiable lvalue** to **identify objects whose value can be changed**.
- Therefore, the left side of an assignment operator should be a modifiable lvalue.
- Indeed, the **l** in lvalue comes from **left** because modifiable lvalues can be used on the left side of assignment operators.

Assignment Operator: =

- The term **rvalue** refers to **quantities that can be assigned to modifiable lvalues**.
- For instance, consider the following statement:

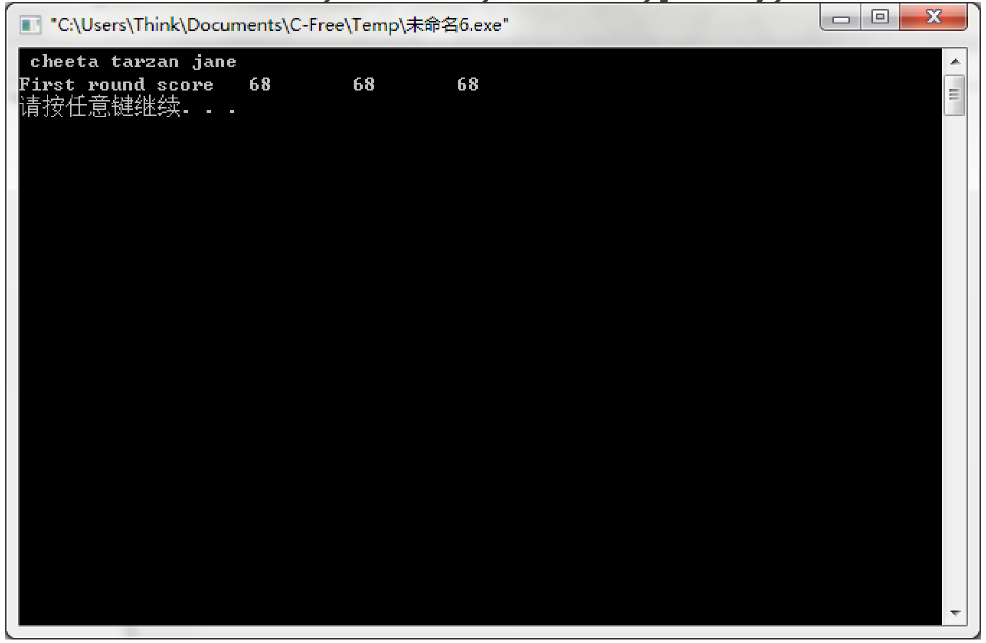
  bmw = 2002;
- Here, bmw is a modifiable lvalue, and 2002 is an rvalue.
- As you probably guessed, the **r** in rvalue comes from **right**.
- Rvalues can be **constants**, **variables**, or **any other expression that yields a value**.

Assignment Operator: =

- As long as you are learning the names of things, the proper term for what we have called an "**item**" (as in "the item to the left of the =") is **operand**.
- Operands are **what operators operate on**.
- For example, you can describe eating a hamburger as applying the "eat" operator to the "hamburger" operand; similarly, you can say that **the left operand of the = operator shall be a modifiable lvalue**.

Listing 5.3. The golf.c Program

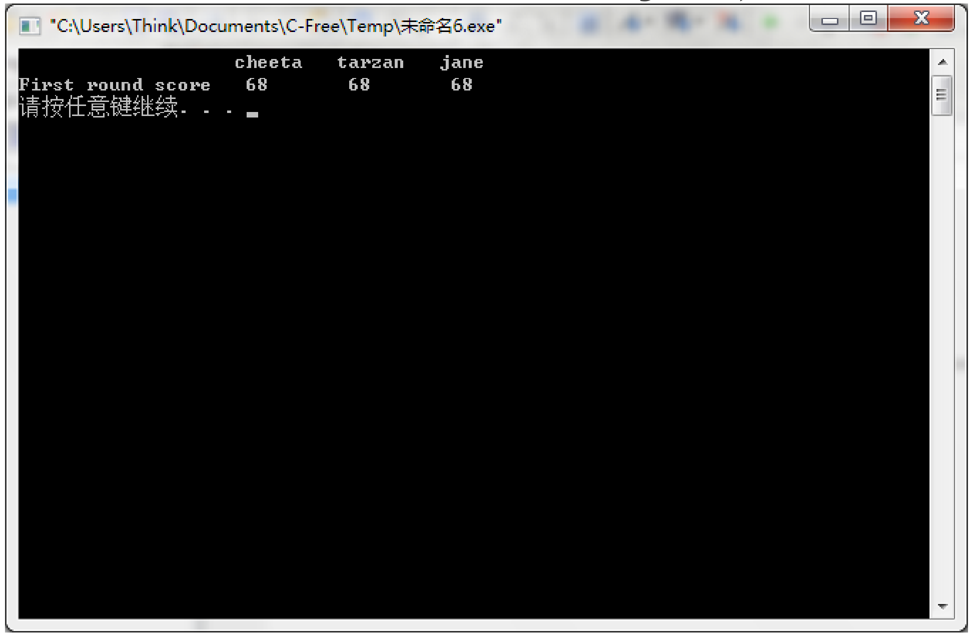
```
/* golf.c -- golf tournament scorecard */
#include <stdio.h>
int main(void)
{
        int jane, tarzan, cheeta;

        cheeta = tarzan = jane = 68;
        printf(" cheeta tarzan jane\n");
        printf("First round score %4d %8d %8d\n",cheeta,tarzan,jane);

        return 0;
}
```



```
 cheeta tarzan jane
First round score   68       68       68
请按任意键继续. . .
```

Listing 5.3. The golf.c Program

```
/* golf.c -- golf tournament scorecard */
#include <stdio.h>
int main(void)
{
        int jane, tarzan, cheeta;

        cheeta = tarzan = jane = 68;
        printf("                  cheeta   tarzan   jane\n");
        printf("First round score %4d %8d %8d\n",cheeta,tarzan,jane);

        return 0;
}
```



"C:\Users\Think\Documents\C-Free\Temp\未命名6.exe"

```
                  cheeta   tarzan   jane
First round score   68       68       68
请按任意键继续. . .
```

Addition Operator: +

- The addition operator causes **the two values on either side of it to be added together**.
- For example, the statement

  printf("%d", 4 + 20);

  causes the number 24 to be printed, not the expression

  4 + 20.
- The values (operands) to be added can be variables as well as constants. Therefore, the statement

  income = salary + bribes;

  causes the computer to look up the values of the two variables on the right, add them, and then assign this total to the variable income.

Subtraction Operator: –

- The subtraction operator causes **the number after the – sign to be subtracted from the number before the sign**.
- The statement

  takehome = 224.00 – 24.00;

  assigns the value 200.0 to takehome.
- The + and - operators are termed **binary**, or **dyadic**, **operators**, meaning that **they require two operands**.

Sign Operators: - and +

- The minus sign can also **be used to indicate or to change the algebraic sign of a value**.
- For instance, the sequence

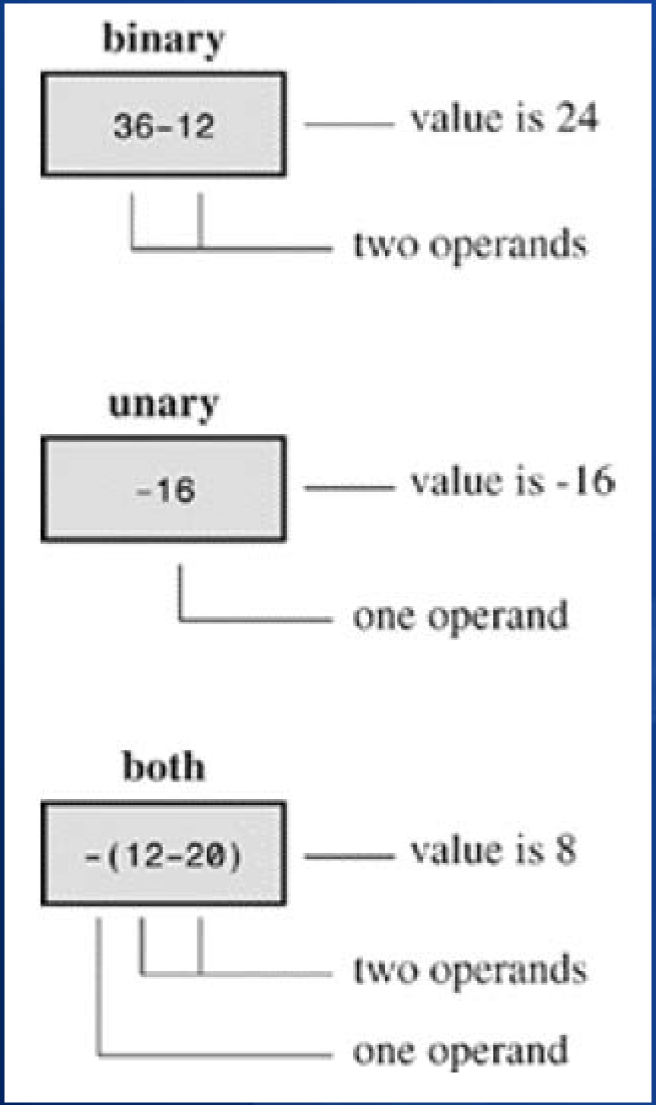
  rocky = –12;

  smokey = –rocky;

  gives smokey the value 12.
- When the minus sign is used in this way, it is called a **unary operator**, meaning that **it takes just one operand**

Sign Operators: - and +

- **Figure 5.2. Unary and binary operators.**



Multiplication Operator: *

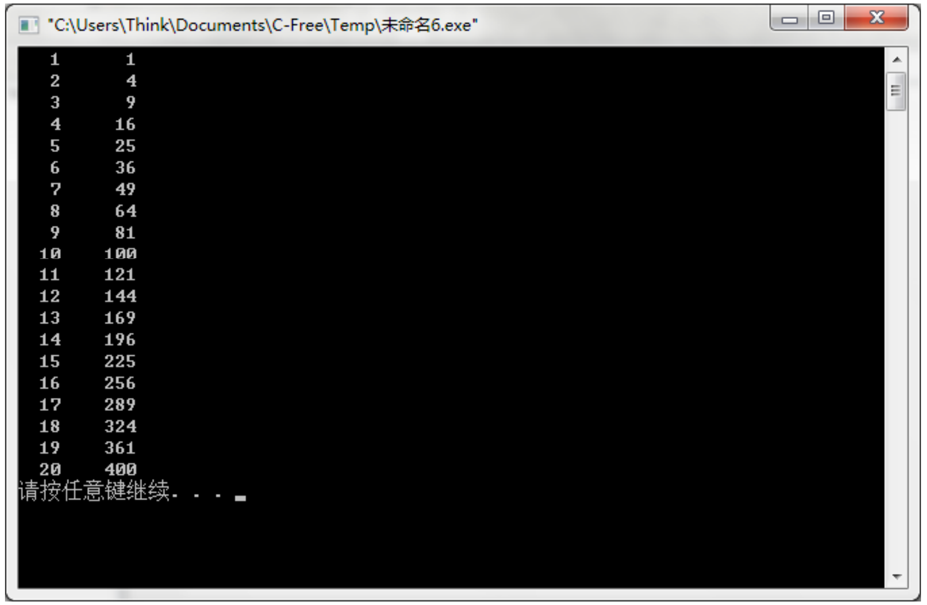
- Multiplication is indicated by the * symbol. The statement

  *cm = 2.54 * inch;*

  multiplies the variable inch by 2.54 and assigns the answer to cm.
- By any chance, do you want a table of squares? C doesn't have a squaring function

Listing 5.4. The squares.c Program

```
/* squares.c -- produces a table of first 20 squares */
#include <stdio.h>
int main(void)
{
        int num = 1;

        while (num < 21)
        {
                printf("%4d %6d\n", num, num * num);
                num = num + 1;
        }

        return 0;
}
```



"C:\Users\Think\Documents\C-Free\Temp\未命名6.exe"

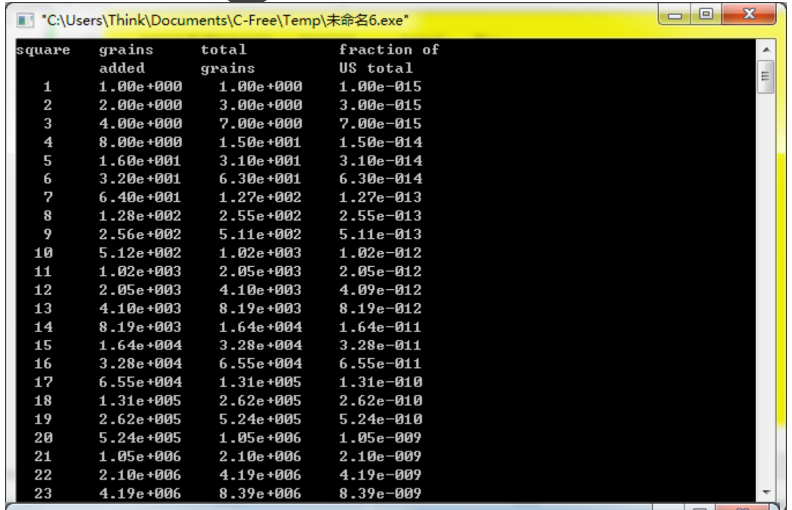
```
   1      1
   2      4
   3      9
   4     16
   5     25
   6     36
   7     49
   8     64
   9     81
  10    100
  11    121
  12    144
  13    169
  14    196
  15    225
  16    256
  17    289
  18    324
  19    361
  20    400
请按任意键继续. . .
```

Exponential Growth

- You have probably heard the story of the powerful ruler who seeks to reward a scholar who has done him great service.
- When the scholar is asked what he would like, he points to a chessboard and says, just one grain of wheat on the first square, two on the second, four on the third, eight on the next, and so on.
- The ruler, lacking mathematical erudition, is astounded at the modesty of this request, for he had been prepared to offer great riches.
- The joke, of course, is on the ruler, as the program in Listing 5.5 shows.

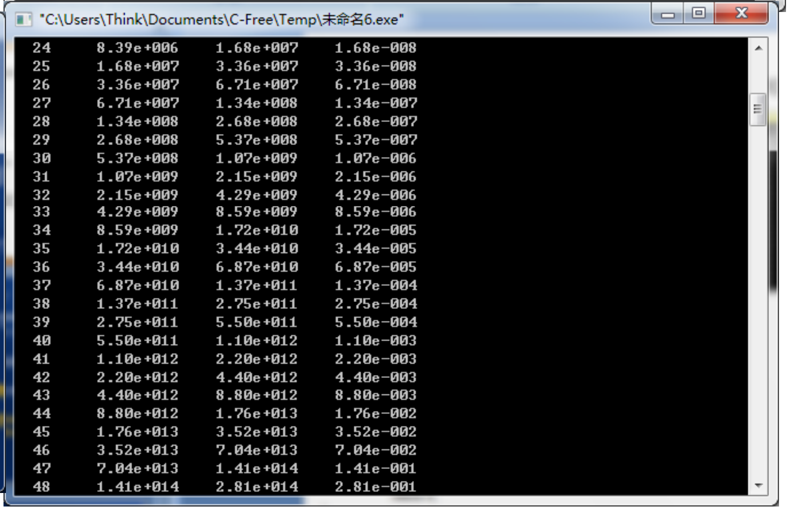
Listing 5.5. The wheat.c Program

```
/* wheat.c -- exponential growth */
#include <stdio.h>
#define SQUARES 64          /* squares on a checkerboard */
#define CROP 1E15           /* US wheat crop in grains    */
int main(void)
{
        double current, total;
        int count = 1;

        printf("square              grains    total              ");
        printf("   fraction of \n");
        printf("         added               grains       ");
        printf("   US total\n");
        total = cu
        printf("%

        while (cou
        {






        }
        printf("That's all.\n");

        return 0;
}
```



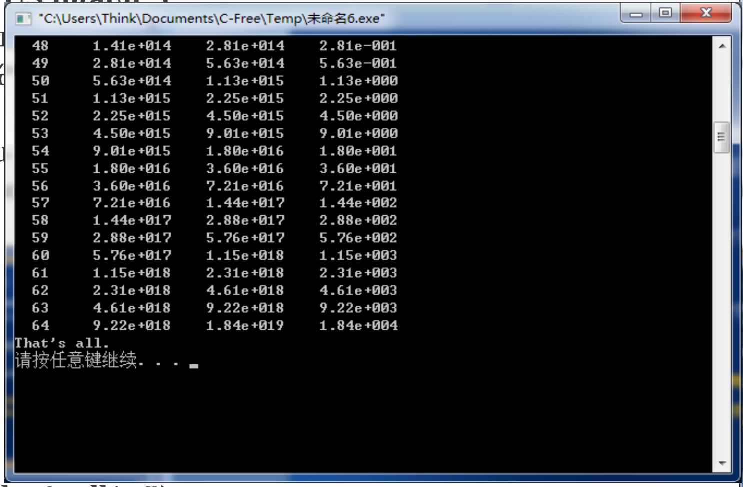
"C:\Users\Think\Documents\C-Free\Temp\未命名6.exe"

```
square    grains      total           fraction of
          added       grains          US total
   1      1.00e+000    1.00e+000      1.00e-015
   2      2.00e+000    3.00e+000      3.00e-015
   3      4.00e+000    7.00e+000      7.00e-015
   4      8.00e+000    1.50e+001      1.50e-014
   5      1.60e+001    3.10e+001      3.10e-014
   6      3.20e+001    6.30e+001      6.30e-014
   7      6.40e+001    1.27e+002      1.27e-013
   8      1.28e+002    2.55e+002      2.55e-013
   9      2.56e+002    5.11e+002      5.11e-013
  10      5.12e+002    1.02e+003      1.02e-012
  11      1.02e+003    2.05e+003      2.05e-012
  12      2.05e+003    4.10e+003      4.09e-012
  13      4.10e+003    8.19e+003      8.19e-012
  14      8.19e+003    1.64e+004      1.64e-011
  15      1.64e+004    3.28e+004      3.28e-011
  16      3.28e+004    6.55e+004      6.55e-011
  17      6.55e+004    1.31e+005      1.31e-010
  18      1.31e+005    2.62e+005      2.62e-010
  19      2.62e+005    5.24e+005      5.24e-010
  20      5.24e+005    1.05e+006      1.05e-009
  21      1.05e+006    2.10e+006      2.10e-009
  22      2.10e+006    4.19e+006      4.19e-009
  23      4.19e+006    8.39e+006      8.39e-009
```



"C:\Users\Think\Documents\C-Free\Temp\未命名6.exe"

```
  24      8.39e+006    1.68e+007      1.68e-008
  25      1.68e+007    3.36e+007      3.36e-008
  26      3.36e+007    6.71e+007      6.71e-008
  27      6.71e+007    1.34e+008      1.34e-007
  28      1.34e+008    2.68e+008      2.68e-007
  29      2.68e+008    5.37e+008      5.37e-007
  30      5.37e+008    1.07e+009      1.07e-006
  31      1.07e+009    2.15e+009      2.15e-006
  32      2.15e+009    4.29e+009      4.29e-006
  33      4.29e+009    8.59e+009      8.59e-006
  34      8.59e+009    1.72e+010      1.72e-005
  35      1.72e+010    3.44e+010      3.44e-005
  36      3.44e+010    6.87e+010      6.87e-005
  37      6.87e+010    1.37e+011      1.37e-004
  38      1.37e+011    2.75e+011      2.75e-004
  39      2.75e+011    5.50e+011      5.50e-004
  40      5.50e+011    1.10e+012      1.10e-003
  41      1.10e+012    2.20e+012      2.20e-003
  42      2.20e+012    4.40e+012      4.40e-003
  43      4.40e+012    8.80e+012      8.80e-003
  44      8.80e+012    1.76e+013      1.76e-002
  45      1.76e+013    3.52e+013      3.52e-002
  46      3.52e+013    7.04e+013      7.04e-002
  47      7.04e+013    1.41e+014      1.41e-001
  48      1.41e+014    2.81e+014      2.81e-001
```



"C:\Users\Think\Documents\C-Free\Temp\未命名6.exe"

```
  48      1.41e+014    2.81e+014      2.81e-001
  49      2.81e+014    5.63e+014      5.63e-001
  50      5.63e+014    1.13e+015      1.13e+000
  51      1.13e+015    2.25e+015      2.25e+000
  52      2.25e+015    4.50e+015      4.50e+000
  53      4.50e+015    9.01e+015      9.01e+000
  54      9.01e+015    1.80e+016      1.80e+001
  55      1.80e+016    3.60e+016      3.60e+001
  56      3.60e+016    7.21e+016      7.21e+001
  57      7.21e+016    1.44e+017      1.44e+002
  58      1.44e+017    2.88e+017      2.88e+002
  59      2.88e+017    5.76e+017      5.76e+002
  60      5.76e+017    1.15e+018      1.15e+003
  61      1.15e+018    2.31e+018      2.31e+003
  62      2.31e+018    4.61e+018      4.61e+003
  63      4.61e+018    9.22e+018      9.22e+003
  64      9.22e+018    1.84e+019      1.84e+004
That's all.
请按任意键继续. . .
```

Division Operator: /

- C uses the / symbol to represent division.
- The value to the left of the / is divided by the value to the right.
- For example, the following gives four the value of 4.0:

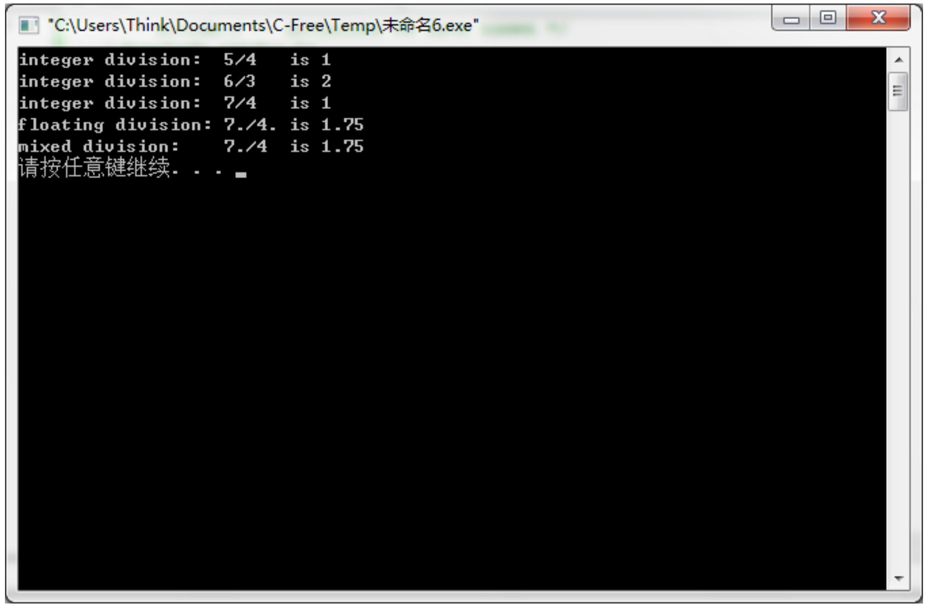
  four = 12.0/3.0;
- Division works differently for integer types than it does for floating types.
- **Floating-type division gives a floating-point answer**, **but integer division yields an integer answer**.

Division Operator: /

- An integer can't have a fractional part, which makes dividing 5 by 3 awkward, because the answer does have a fractional part.
- In C, any fraction resulting from integer division is discarded.
- This process is called **truncation**.

Listing 5.6. The divide.c Program

```
/* divide.c -- divisions we have known */
#include <stdio.h>
int main(void)
{
        printf("integer division:  5/4   is %d \n", 5/4);
        printf("integer division:  6/3   is %d \n", 6/3);
        printf("integer division:  7/4   is %d \n", 7/4);
        printf("floating division: 7./4. is %1.2f \n", 7./4.);
        printf("mixed division:    7./4  is %1.2f \n", 7./4);

        return 0;
}
```



"C:\Users\Think\Documents\C-Free\Temp\未命名6.exe"

```
integer division:  5/4   is 1
integer division:  6/3   is 2
integer division:  7/4   is 1
floating division: 7./4. is 1.75
mixed division:    7./4  is 1.75
请按任意键继续. . .
```

Division Operator: /

- Notice how integer division **does not round to the nearest integer**, but **always truncates** (that is, discards the entire fractional part).
- When you **mixed integers with floating point**, **the answer came out the same as floating point**.
- Actually, the computer is not really capable of dividing a floating-point type by an integer type, so the compiler converts both operands to a single type.
- In this case, the integer is converted to floating point before division.

Operator Precedence

- What happens when you combine more than one operation into one statement?
- Consider the following line of code:

  *butter = 25.0 + 60.0 * n / SCALE;*

  This statement has an addition, a multiplication, and a division operation.
- Which operation takes place first?
- Clearly, the order of executing the various operations can make a difference, so C needs unambiguous rules for choosing what to do first.
- C does this by setting up **an operator pecking order**. **Each operator is assigned a precedence level**.

Operator Precedence

- As in ordinary arithmetic, **multiplication and division have a higher precedence than addition and subtraction**, so they are performed first.
- What if two operators have the same precedence?
- **If they share an operand, they are executed according to the order in which they occur in the statement**.
- For most operators, the order is **from left to right**.(**The = operator was an exception to this**.)

Operator Precedence

*butter = 25.0 + 60.0 * n / SCALE;*

- the order of operations is as follows:

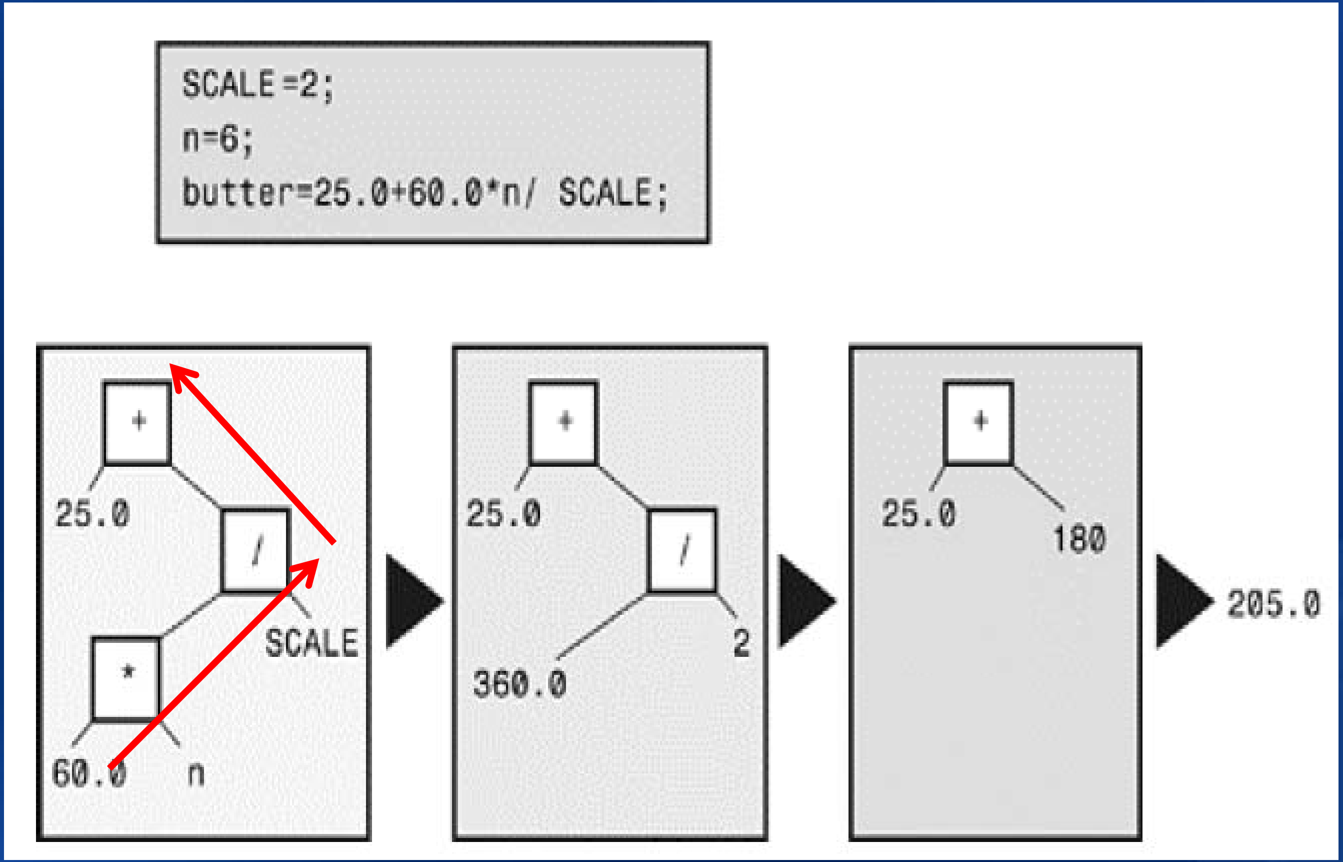
60.0 * n The first * or / in the expression (assuming n is 6 so that 60.0 * n is 360.0)

360.0 / SCALE Then the second * or / in the expression

25.0 + 180 Finally (because SCALE is 2.0), the first + or - in the expression, to yield 205.0

Operator Precedence

- Many people like to represent the order of evaluation with a type of diagram called **an expression tree**.



```
SCALE=2;
n=6;
butter=25.0+60.0*n/ SCALE;
```

Figure 5.3. Expression trees showing operators, operands, and order of evaluation.

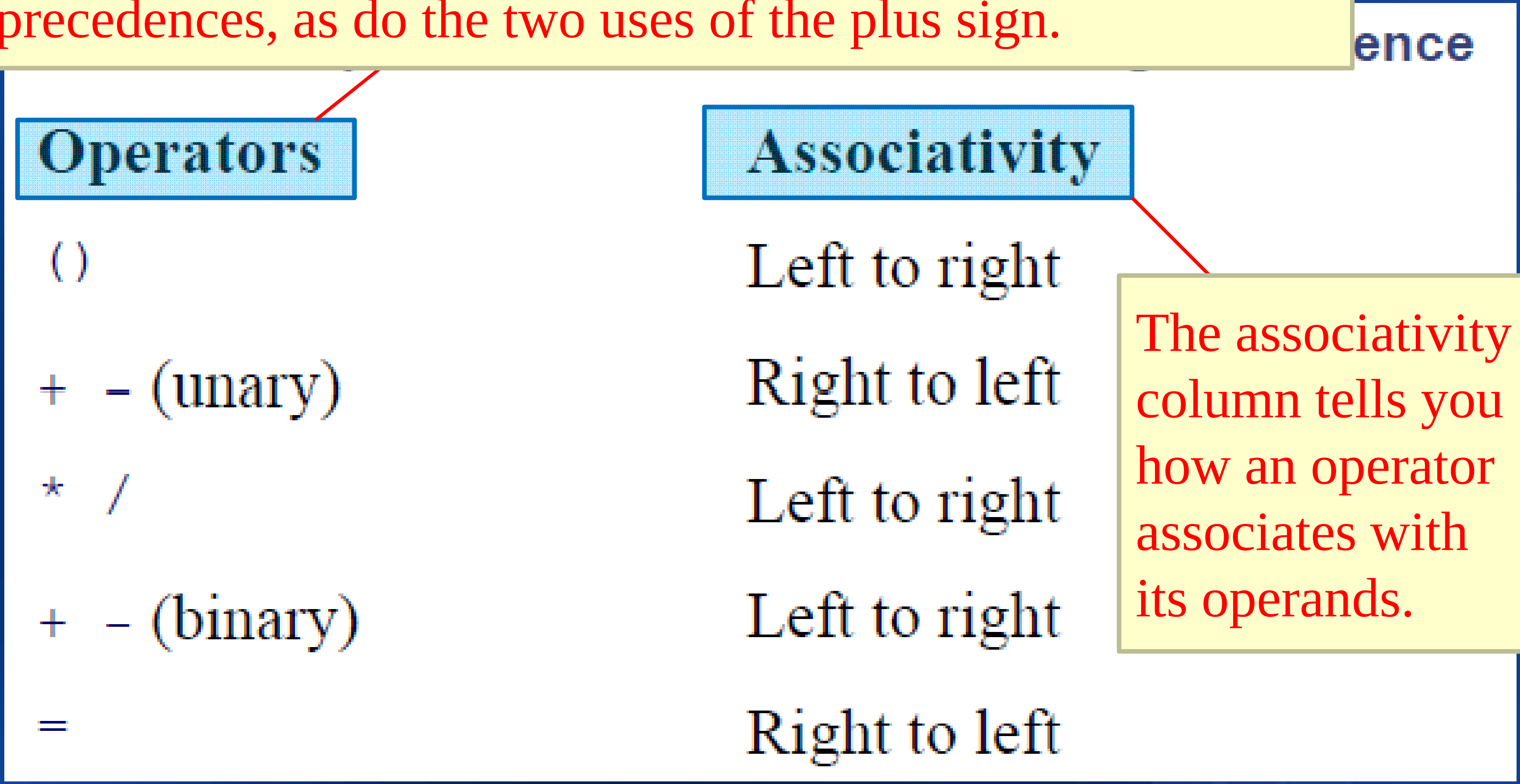
Operator Precedence

- What if you want an addition operation to take place before division?
- Then you can do as we have done in the following line:

  *flour = (25.0 + 60.0 * n) / SCALE;*
- Whatever is enclosed in parentheses **is executed first**.
- Within the parentheses, the usual rules hold.
- For this example, first the multiplication takes place and then the addition.
- That completes the expression in the parentheses.
- Now the result can be divided by SCALE.

Operator Precedence

Notice that the two uses of the minus sign have different precedences, as do the two uses of the plus sign.



ence

Operators	Associativity
( )	Left to right
+ - (unary)	Right to left
* /	Left to right
+ - (binary)	Left to right
=	Right to left

The associativity column tells you how an operator associates with its operands.

Precedence and the Order of Evaluation

- Operator precedence provides vital rules for determining the order of evaluation in an expression, but it doesn't necessarily determine the complete order.
- C leaves some choices up to the implementation.
- Consider the following statement:

  *y = 6 * 12 + 5 * 20;*
- Precedence dictates the order of evaluation when two operators share an operand.
- For example, the 12 is an operand for both the * and the + operators, and precedence says that multiplication comes first.

Precedence and the Order of Evaluation

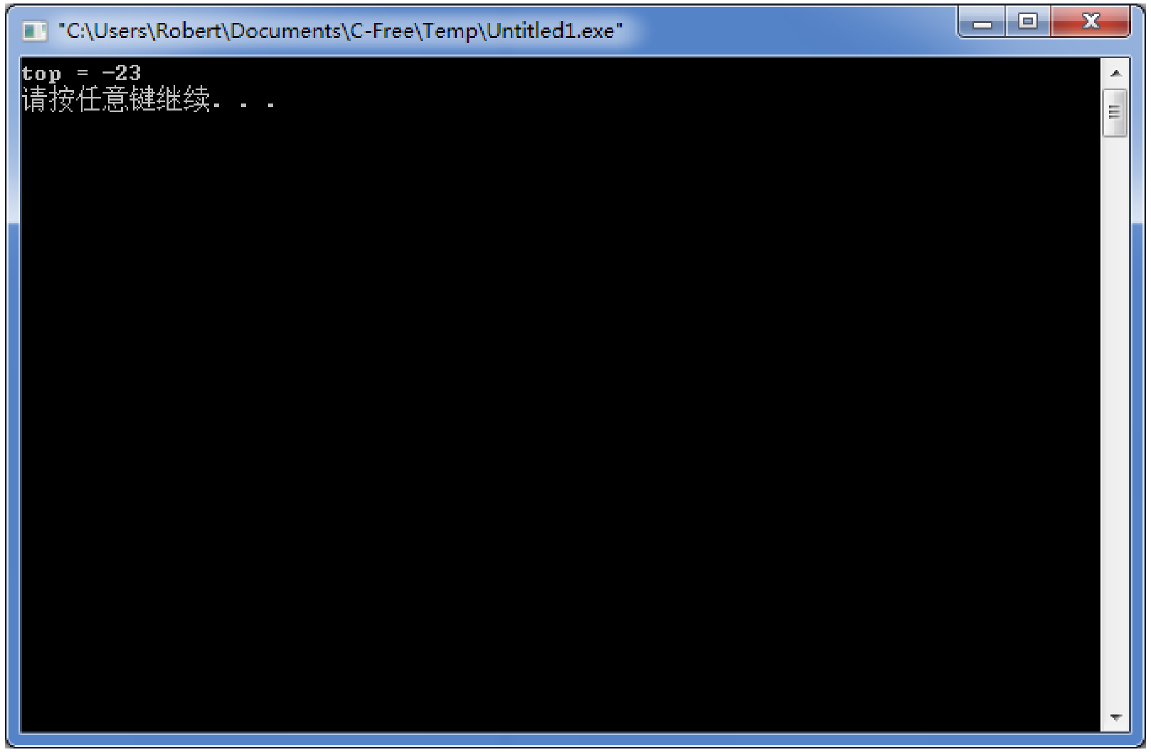
- In short, the multiplications 6 * 12 and 5 * 20 take place before any addition.
- **What precedence does not establish is which of these two multiplications occurs first**.
- C leaves that choice to the implementation because one choice might be more efficient for one kind of hardware, but the other choice might work better on another kind of hardware.
- In either case, the expression reduces to 72 + 100, so the choice doesn't affect the final value for this particular example.

Precedence and the Order of Evaluation

- You may say, "multiplication associates from left to right. Doesn't that mean the leftmost multiplication is performed first?“ **The association rule applies for operators that share an operand**.
- For instance, in the expression 12 / 3 * 2, the / and * operators, which have the same precedence, share the operand 3.
- Therefore, the left-to-right rule applies in this case, and the expression reduces to 4 * 2, or 8. (Going from right to left would give 12 / 6, or 2.)
- In the previous example, the two * operators did not share a common operand, so the left-to-right rule did not apply.

Listing 5.7. The rules.c Program

```
/* rules.c -- precedence test */
#include <stdio.h>
int main(void)
{
        int top, score;
        top = score = -(2 + 5) * 6 + (4 + 3 * (2 + 3));
        printf("top = %d \n", top);
        return 0;
}
```



```
top = -23
请按任意键继续. . .
```

Some Additional Operators - Modulus Operator: %

- C has **about 40 operators**, but some are used much more than others.
- The ones just covered are the most common, but let's add four more useful operators to the list.

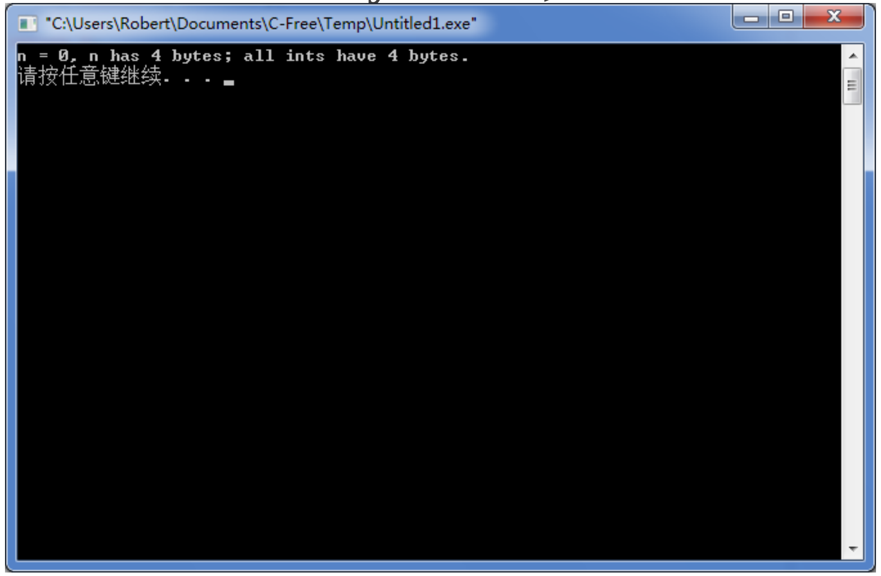
The sizeof Operator and the size_t Type

- To review, the sizeof operator returns the size, in bytes, of its operand. (Recall that a C byte is defined as the size used by the char type. In the past, this has most often been 8 bits, but some character sets may use larger bytes.)
- The operand can be a specific data object, such as the name of a variable, or it can be a type. If it is a type, such as float, the operand must be enclosed in parentheses.

Listing 5.8. The sizeof.c Program

```
// sizeof.c -- uses sizeof operator
// uses C99 %z mod
#include <stdio.h>
int main(void)
{
        int n = 0;
        size_t intsize;

        intsize = sizeof (int);
        printf("n = %d, n has %u bytes; all ints have %u bytes.\n",
                n, sizeof n, intsize );

        return 0;
}
```

C says that sizeof returns a value of type size_t. This is an unsigned integer type, but not a brand-new type. Instead, like the portable types (int32_t and so on), it is defined in terms of the standard types.



```
n = 0, n has 4 bytes; all ints have 4 bytes.
请按任意键继续. . .
```

The sizeof Operator and the size_t Type

- C has a typedef mechanism (discussed further in Chapter 14, "Structures and Other Data Forms") that lets you create **an alias** for an existing type.
- For example,

  typedef double real;

- makes **real** another name for **double**. Now you can declare a variable of type real:

  real deal; // using a typedef

- The compiler will see the word real, recall that the typedef statement made real an alias for double, and **create deal as a type double variable**.

The sizeof Operator and the size_t Type

- Similarly, the C header files system can use **typedef** to make **size_t** a synonym for **unsigned int** on one system or for **unsigned long** on another. Thus, when you use the size_t type, the compiler will **substitute** the standard type that works for your system.
- C99 goes a step further and supplies %zd as a printf() specifier for displaying a size_t value. If your system doesn't implement %zd, you can try using %u or %lu instead.

Modulus Operator: %

- The **modulus operator** is used in integer arithmetic.
- It gives **the remainder** that results when the integer to its left is divided by the integer to its right.
- For example, 13 % 5 (read as "13 modulo 5") has the value 3, because 5 goes into 13 twice, with a remainder of 3.
- Don't bother trying to use this operator with floating-point numbers. It just won't work.

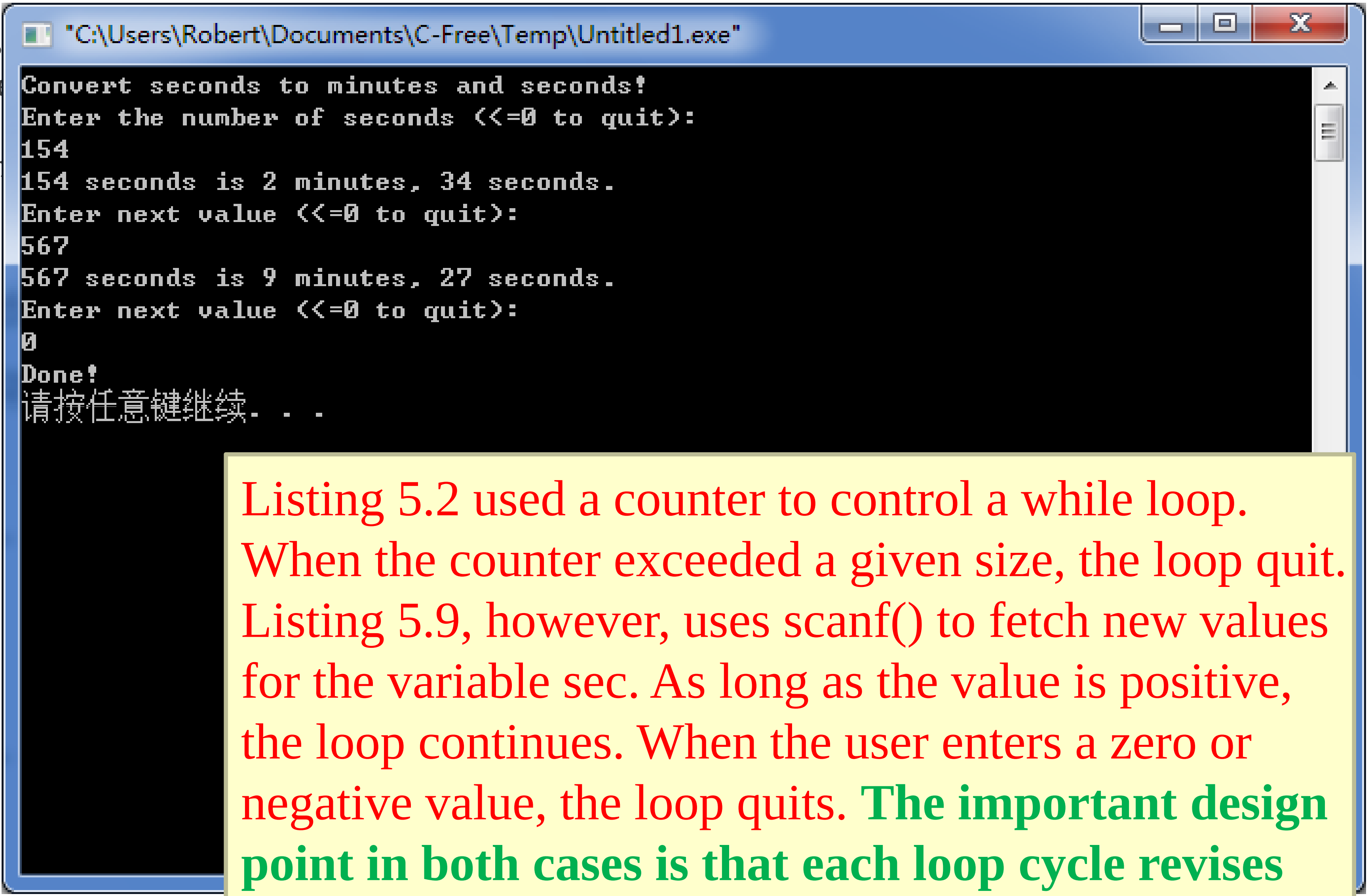
Modulus Operator: %

- One common use is to **help you control the flow of a program**.
- Suppose, for example, you are working on a bill-preparing program designed to add in an extra charge every third month.
- Just have the program evaluate the month number modulo 3 (that is, month % 3) and check to see whether the result is 0. If it is, the program adds in the extra charge.

Listing 5.9. The min_sec.c Program

// min_s
#include
#define
int mair
{

}


"C:\Users\Robert\Documents\C-Free\Temp\Untitled1.exe"

```
Convert seconds to minutes and seconds!
Enter the number of seconds (<=0 to quit):
154
154 seconds is 2 minutes, 34 seconds.
Enter next value (<=0 to quit):
567
567 seconds is 9 minutes, 27 seconds.
Enter next value (<=0 to quit):
0
Done!
请按任意键继续. . .
```

Listing 5.2 used a counter to control a while loop. When the counter exceeded a given size, the loop quit. Listing 5.9, however, uses scanf() to fetch new values for the variable sec. As long as the value is positive, the loop continues. When the user enters a zero or negative value, the loop quits. **The important design point in both cases is that each loop cycle revises the value of the variable being tested**.

Modulus Operator: %

- What about negative numbers?
- Before C99 settled on the "truncate toward zero" rule for integer division, there were a couple of possibilities.
- But with the rule in place, you get a negative modulus value if the first operand is negative, and you get a positive modulus otherwise:

  11 / 5 is 2, and 11 % 5 is 1

  11 / -5 is -2, and 11 % -2 is 1

  -11 / -5 is 2, and -11 % -5 is -1

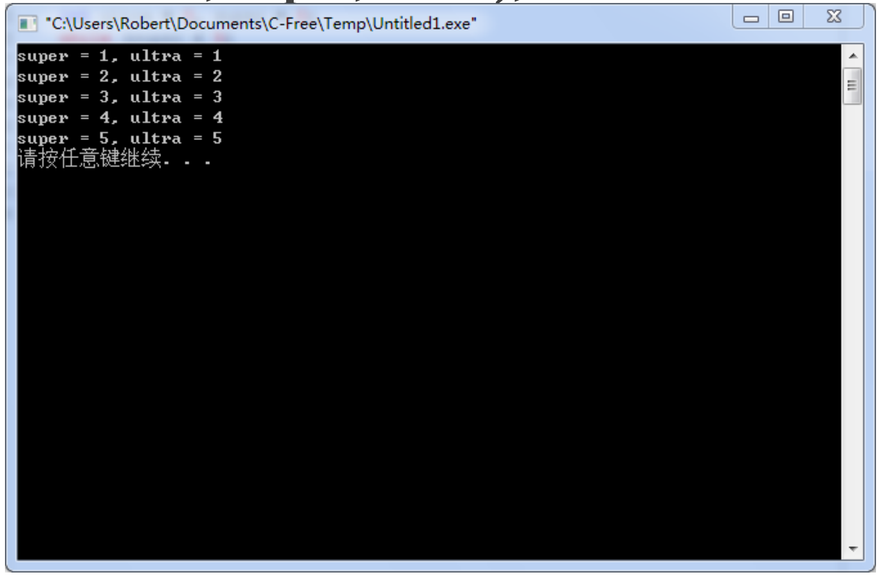
  -11 / 5 is -2, and -11 % 5 is -1

Increment and Decrement Operators: ++ and --

- The increment operator performs a simple task; **it increments (increases) the value of its operand by 1**.
- This operator comes **in two varieties**.
- The first variety has the ++ come before the affected variable; this is **the prefix mode**.
- The second variety has the ++ after the affected variable; this is **the postfix mode**.
- The two modes differ with regard to **the precise time that the incrementing takes place**.
- We'll explain **the similarities first** and then return to that difference.

Listing 5.10. The add_one.c Program

```
/* add_one.c -- incrementing: prefix and postfix */
#include <stdio.h>
int main(void)
{
        int ultra = 0, super = 0;
        while (super < 5)
        {
                super++;
                ++ultra;
                printf("super = %d, ultra = %d \n", super, ultra);
        }
        return 0;
}
```



"C:\Users\Robert\Documents\C-Free\Temp\Untitled1.exe"

```
super = 1, ultra = 1
super = 2, ultra = 2
super = 3, ultra = 3
super = 4, ultra = 4
super = 5, ultra = 5
请按任意键继续. . .
```

Increment and Decrement Operators: ++ and --

- The program counted to five twice and simultaneously. You could get the same results by replacing the two increment statements with this:

  super = super + 1;

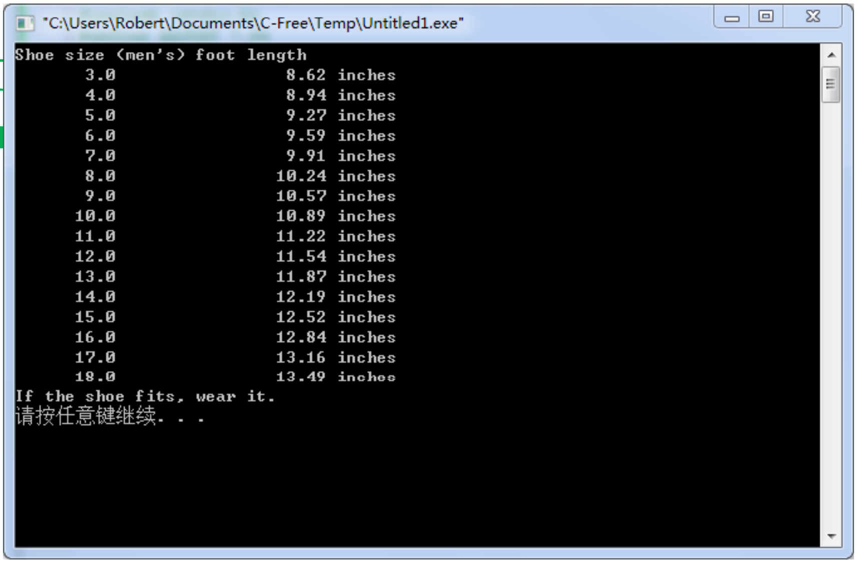
  ultra = ultra + 1;

- These are simple enough statements. Why bother creating one, let alone two, abbreviations?
- One reason is that **the compact form makes your programs neater and easier to follow**. These operators give your programs an elegant gloss that cannot fail to please the eye.

Listing 5.2. The shoes2.c Program

```c
/* shoes2.c -- calculates foot lengths for several sizes */
#include <stdio.h>
#define ADJUST 7.64
#define SCALE 0.325
int main(void)
{
        double shoe, foot;

        printf("Shoe size (men's) foot length\n");
        shoe = 3.0;
        while (shoe < 18.5)    /* starting the while loop */
        {                      /* start of block */
                foot = SCALE*shoe + ADJUST
                printf("%10.1f %15.2f inches\
                shoe = shoe + 1.0;
        }                      /* end of block */
        printf("If the shoe fits, wear it. \n");

        return 0;
}
```

```c
shoe = 3.0;
while (shoe < 18.5)
{
        foot = SCALE * shoe + ADJUST;
        printf("%10.1f %20.2f inches\n", shoe, foot);
        ++shoe;
}
```

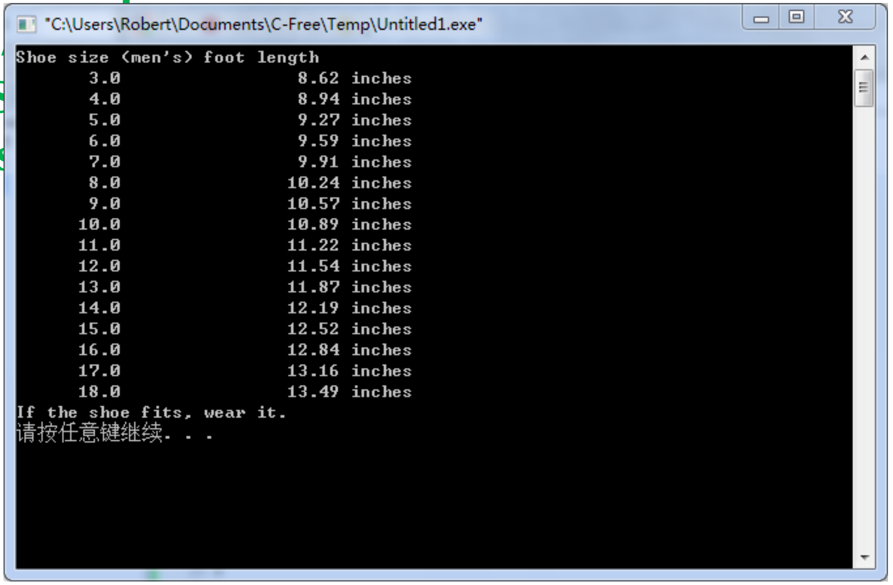

"C:\Users\Robert\Documents\C-Free\Temp\Untitled1.exe"

```
Shoe size (men's) foot length
       3.0                 8.62 inches
       4.0                 8.94 inches
       5.0                 9.27 inches
       6.0                 9.59 inches
       7.0                 9.91 inches
       8.0                10.24 inches
       9.0                10.57 inches
      10.0                10.89 inches
      11.0                11.22 inches
      12.0                11.54 inches
      13.0                11.87 inches
      14.0                12.19 inches
      15.0                12.52 inches
      16.0                12.84 inches
      17.0                13.16 inches
      18.0                13.49 inches
If the shoe fits, wear it.
请按任意键继续. . .
```

Listing 5.2. The shoes2.c Program

```
/* shoes2.c -- calculates foot lengths for several sizes */
#include <stdio.h>
#define ADJUST 7.64
#define SCALE 0.325
int main(void)
{
        double shoe, foot;

        printf("Shoe size (men's) foot length\n");
        shoe = 3.0;
        while (shoe < 18.5)   /* starting the while loop */
        {                     /* start of block *
                foot = SCALE*shoe + ADJUS
                printf("%10.1f %15.2f inche
                shoe = shoe + 1.0;
        }                     /* end of block */
        printf("If the shoe fits, wear it. \n");

        return 0;
}
```

```
shoe = 2.0;
while (++shoe < 18.5)
{
        foot = SCALE*shoe + ADJUST;
        printf("%10.1f %20.2f inches\n", shoe, foot);
}
```



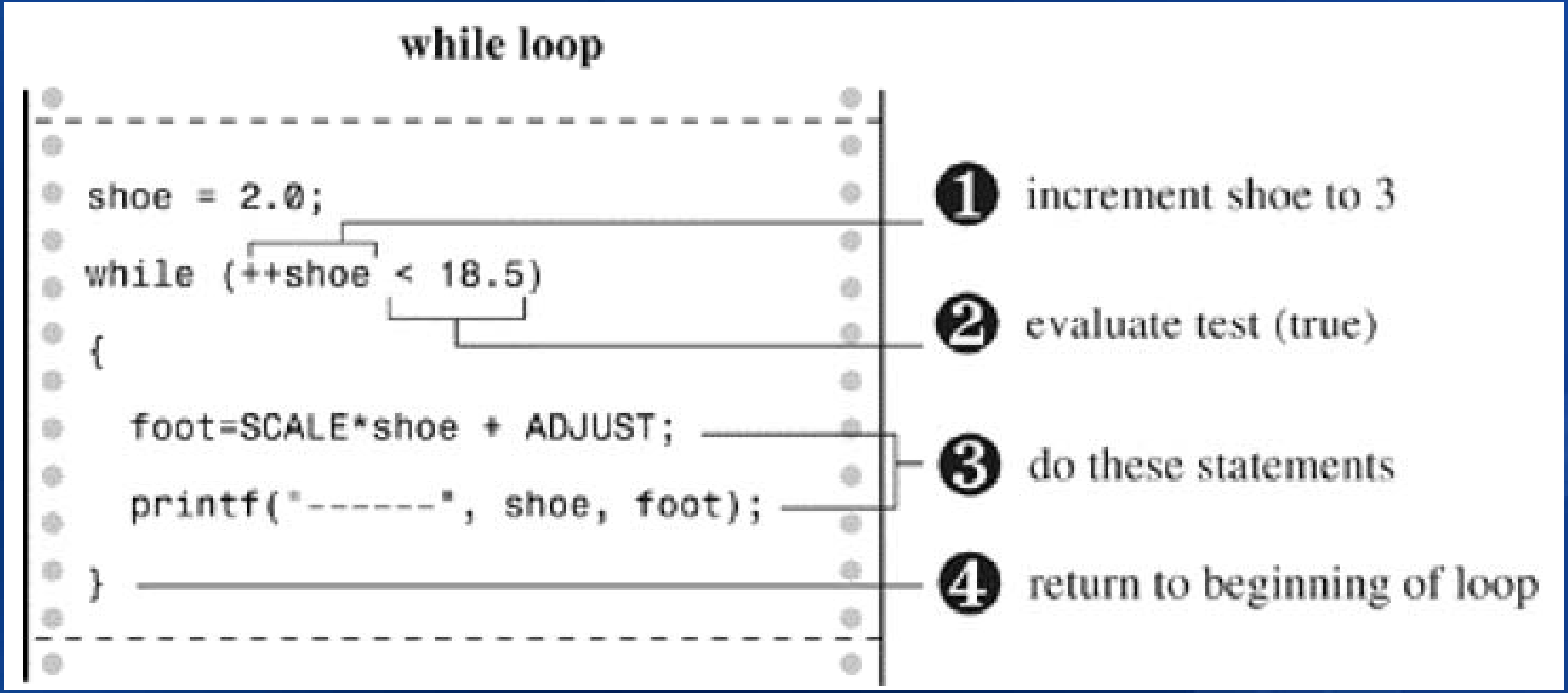
"C:\Users\Robert\Documents\C-Free\Temp\Untitled1.exe"

```
Shoe size (men's) foot length
       3.0                 8.62 inches
       4.0                 8.94 inches
       5.0                 9.27 inches
       6.0                 9.59 inches
       7.0                 9.91 inches
       8.0                10.24 inches
       9.0                10.57 inches
      10.0                10.89 inches
      11.0                11.22 inches
      12.0                11.54 inches
      13.0                11.87 inches
      14.0                12.19 inches
      15.0                12.52 inches
      16.0                12.84 inches
      17.0                13.16 inches
      18.0                13.49 inches
If the shoe fits, wear it.
请按任意键继续. . .
```

Increment and Decrement Operators: ++ and --

- Here you have combined the incrementing process and the while comparison into one expression. This type of construction is so common in C that it merits a closer look.
- First, how does this construction work?
- Simply. The value of shoe is increased by 1 and then compared to 18.5. If it is less than 18.5, the statements between the braces are executed once. Then shoe is increased by 1 again, and the cycle is repeated until shoe gets too big.
- You **changed the initial value of shoe from 3.0 to 2.0** to compensate for shoe being incremented before the first evaluation of foot

Increment and Decrement Operators: ++ and --



```
shoe = 2.0;
while (++shoe < 18.5)
{
   foot=SCALE*shoe + ADJUST;
   printf("------", shoe, foot);
}
```

Increment and Decrement Operators: ++ and --

- Second, what's so **good** about this approach?
- It is more compact.
- More important, it gathers in one place the two processes that control the loop.
- The primary process is the test: Do you continue or not? In this case, the test is checking to see whether the shoe size is less than 18.5.
- The secondary process changes an element of the test; in this case, the shoe size is increased.

Increment and Decrement Operators: ++ and --

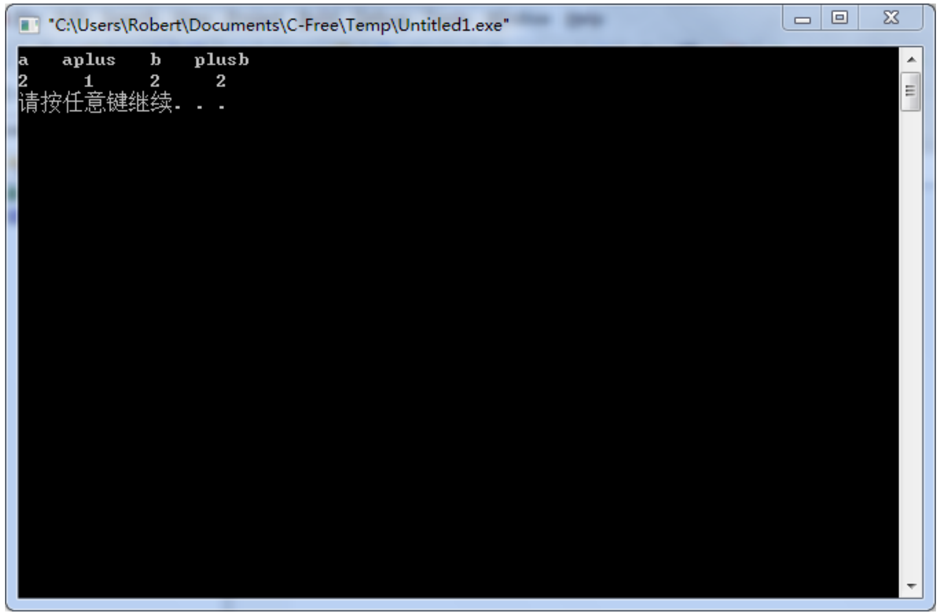
- Suppose you forgot to change the shoe size. Then shoe would always be less than 18.5, and the loop would never end.
- The computer would churn out line after identical line, caught in **a dreaded infinite loop**.
- Eventually, you would lose interest in the output and have to kill the program somehow.
- Having the loop test and the loop change at one place, instead of at separate locations, **helps you to remember to update the loop**.

Increment and Decrement Operators: ++ and --

- A **disadvantage** is that combining two operations in a single expression can make the code harder to follow and can make it easier to make counting errors.
- **Another advantage** of the increment operator is that it usually produces slightly more efficient machine language code because it is similar to actual machine language instructions.
- However, as vendors produce better C compilers, this advantage may disappear. A smart compiler can recognize that x = x + 1 can be treated the same as ++x.
- Finally, these operators have an additional feature that can be useful in certain delicate situations.

Listing 5.11. The post_pre.c Program

```
/* post_pre.c -- postfix vs prefix */
#include <stdio.h>
int main(void)
{
        int a = 1, b = 1;
        int aplus, plusb;
        aplus = a++; /* postfix */
        plusb = ++b; /* prefix */
        printf("a   aplus   b   plusb \n");
        printf("%1d %5d %5d %5d\n", a, aplus, b, plusb);
        return 0;
}
```

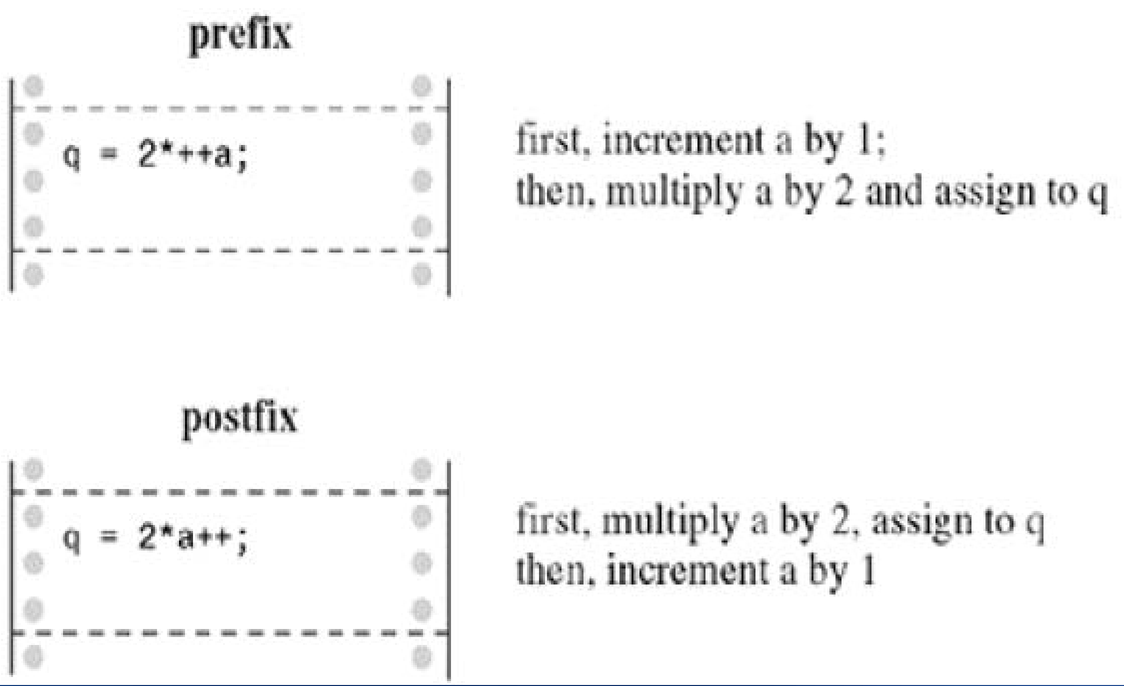


```
"C:\Users\Robert\Documents\C-Free\Temp\Untitled1.exe"
a    aplus    b    plusb
2     1       2     2
请按任意键继续. . .
```

Increment and Decrement Operators: ++ and --

- Both a and b were increased by 1, as promised.
- However, aplus **has the value of a before a changed**, but plusb **has the value of b after b changed**.
- This is the difference between the **prefix** form and the **postfix** form

Figure 5.5. Prefix and postfix.



```
q = 2*++a;
```

first, increment a by 1;
then, multiply a by 2 and assign to q

```
q = 2*a++;
```

first, multiply a by 2, assign to q
then, increment a by 1

Precedence

- The increment and decrement operators have a very high precedence of association; only parentheses are higher.
- Therefore, x*y++ means (x)*(y++), not (x*y)++, which is fortunate because the latter is invalid.
- The increment and decrement operators affect a variable (or, more generally, a modifiable lvalue), and the combination x*y is not itself a variable, although its parts are.
- **Don't confuse precedence of these two operators with the order of evaluation**.

Precedence

- Suppose you have the following:

  y = 2;

  n = 3;

  nextnum = (y + n++)*6;
- What value does nextnum get? Substituting in values yields

  nextnum = (2 + 3)*6 = 5*6 = 30
- Only after n is used is it increased to 4. Precedence tells us that the ++ is attached only to the n, not to y + n.
- It also tells us when the value of n is used for evaluating the expression, but the nature of the increment operator determines when the value of n is changed.

Precedence

- When n++ is part of an expression, you can think of it as meaning "use n; then increment it."
- On the other hand, ++n means "increment n; then use it."

Don't Be Too Clever

- In C, the compiler can choose which arguments in a function to evaluate first.
- This freedom increases compiler efficiency, but can cause trouble if you use an increment operator on a function argument.
- Another possible source of trouble is a statement like this one:

  ans = num/2 + 5*(1 + num++);
- Again, the problem is that the compiler may not do things in the same order you have in mind. You would think that it would find num/2 first and then move on, but it might do the last term first, increase num, and use the new value in num/2. There is no guarantee.

Don't Be Too Clever

- You can easily avoid these problems:
- Don't use increment or decrement operators on a variable that is part of more than one argument of a function.
- Don't use increment or decrement operators on a variable that appears more than once in an expression.

Expressions and Statements

- **Statements** form **the basic program steps of C**, and **most statements are constructed from expressions**.
- An **expression consists of a combination of operators and operands**. (An operand, recall, is what an operator operates on.) The simplest expression is a lone operand, and you can build in complexity from there.

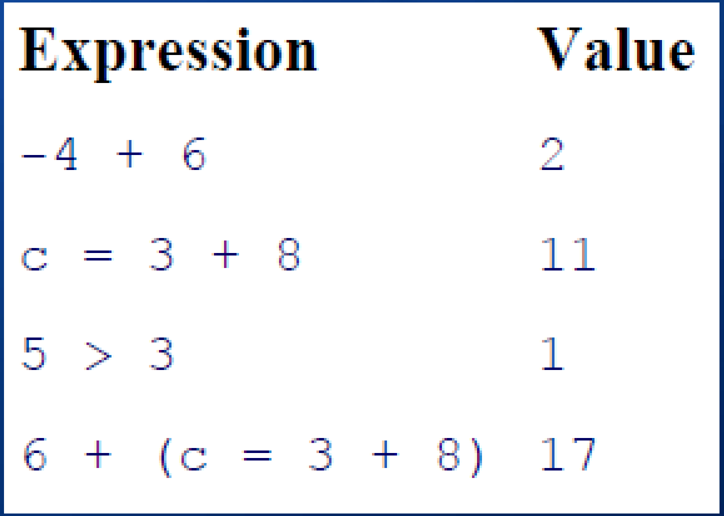
```
4
-6
4+21
a*(b + c/d)/20
q = 5*2
x = ++q % 3
q > 3
```

•As you can see, the operands can be constants, variables, or combinations of the two.
•Some expressions are combinations of smaller expressions, called subexpressions.
•For example, c/d is a subexpression of the fourth example.

Every Expression Has a Value

- An important property of C is that **every C expression has a value**.
- To find the value, you perform the operations in the order dictated by operator precedence. The value of the first few expressions we just listed is clear, but what about the ones with = signs?
- Those expressions simply have the same value that the variable to the left of the = sign receives. Therefore, the expression q=5*2 as a whole has the value 10.
- What about the expression q > 3? Such relational expressions have the value 1 if true and 0 if false.

Every Expression Has a Value



Expression	Value
`-4 + 6`	2
`c = 3 + 8`	11
`5 > 3`	1
`6 + (c = 3 + 8)`	17

- The last expression looks strange!
- However, it is perfectly legal (but ill-advised) in C because it is the sum of two subexpressions, each of which has a value.

Statements

- Statements are the primary building blocks of a program.
- A program is a series of statements with some necessary punctuation.
- A statement is a complete instruction to the computer.
- In C, statements are indicated by a semicolon at the end.
- Therefore,

  legs = 4

  is just an expression (which could be part of a larger expression), but

  legs = 4;

  is a statement.

Statements

- What makes a complete instruction?
- First, C considers any expression to be a statement if you append a semicolon. (These are called expression statements.) Therefore, C won't object to lines such as the following:

```
8;
3 + 4;
```

- However, these statements do nothing for your program and can't really be considered sensible statements.

Statements

- More typically, statements change values and call functions:

```
x = 25;
++x;
y = sqrt(x);
```

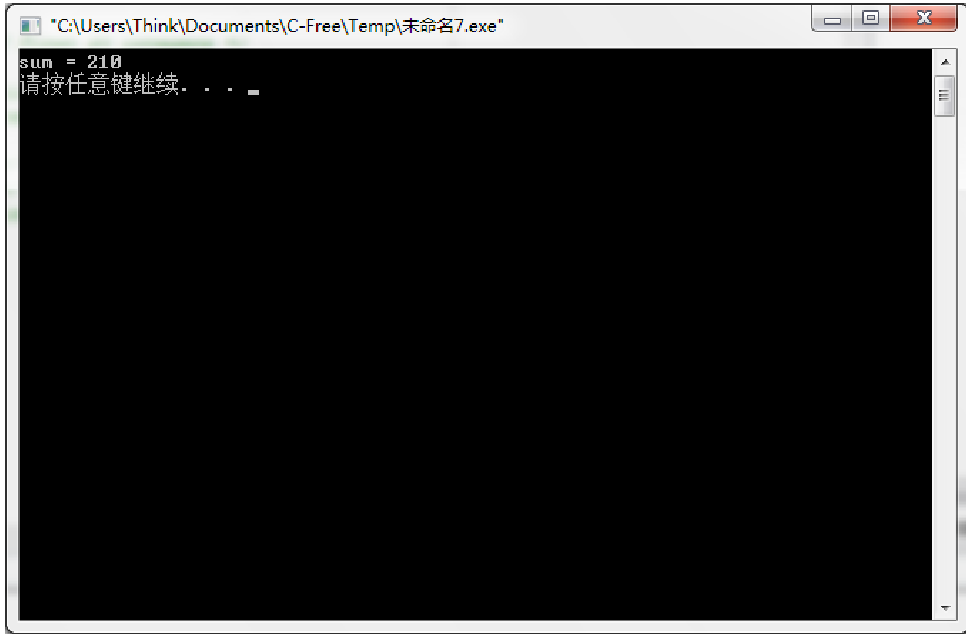
- Although a statement (or, at least, a sensible statement) is a complete instruction, not all complete instructions are statements. Consider the following statement:

```
x = 6 + (y = 5);
```

Because a complete instruction is not necessarily a statement, a semicolon is needed to identify instructions that truly are statements.

Listing 5.13. The addemup.c Program

```
/* addemup.c -- four kinds of statements */
#include <stdio.h>
int main(void)                              /* finds sum of first 20 integers */
{
        int count, sum;                     /* declaration statement */
        count = 0;                          /* assignment statement */
        sum = 0;                            /* ditto */
        while (count++ < 20)                /* while */
                sum = sum + count;          /* statement */
        printf("sum = %d\n", sum);          /* function statement */
        return 0;
}
```



```
sum = 210
请按任意键继续. . .
```

Statements - Declaration Statement

- It **establishes the names and type of variables** and **causes memory locations to be set aside for them**.
- Note that a declaration statement is not an expression statement.
- That is, if you remove the semicolon from a declaration, you get something that is not an expression and that does not have a value:

```
int port                /* not an expression, has no value */
```

Statements - Assignment Statement

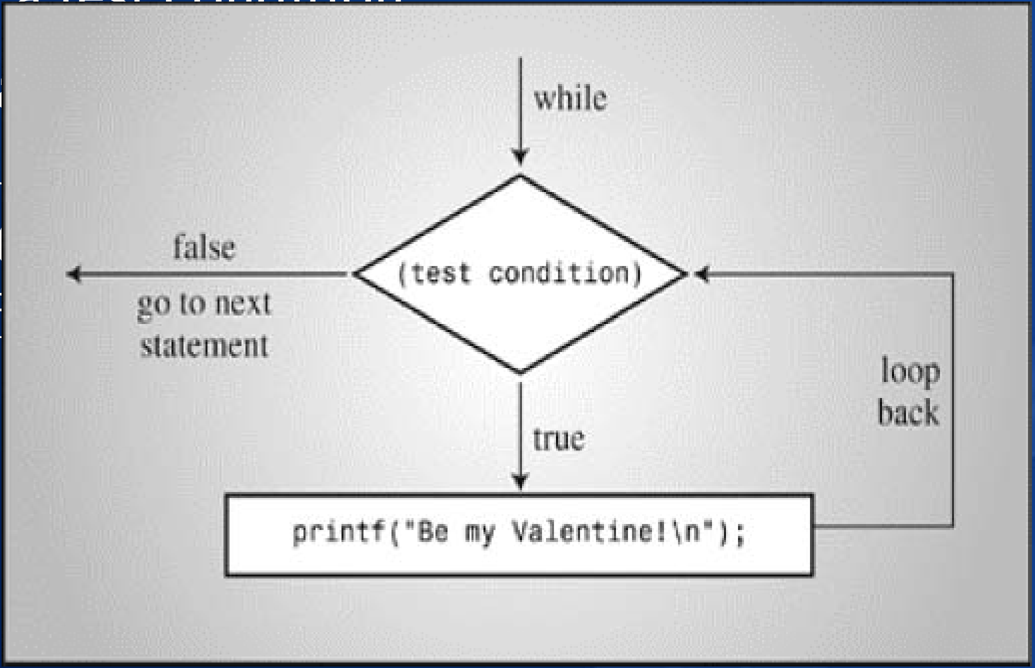
- The assignment statement is the workhorse of many programs; it assigns a value to a variable.
- It consists of a variable name followed by the assignment operator (=) followed by an expression followed by a semicolon.
- Note that this particular while statement includes an assignment statement within it.
- An assignment statement is an example of an expression statement.

Statements - Function Statement

- A function statement causes the function to do whatever it does.
- In this example, the printf() function is invoked to print some results.

Statements - While Statement

- A while statement has three distinct parts
  - First is the keyword while.
  - Then, in parentheses, is a test condition
  - Finally, you have the st
    met. Only one statemen
    simple statement, as in t
    are needed to mark it of
    statement, like some of
    braces are required.



```
printf("Be my Valentine!\n");
```

The while statement belongs to a class of statements sometimes called structured statements because they possess a structure more complex than that of a simple assignment statement.

Statements - Side Effects and Sequence Points

- A side effect is the modification of a data object or file.
- For instance, the side effect of the statement

  states = 50;

  is to set the states variable to 50. Side effect? This looks more like the main intent!
- From the standpoint of C, however, the main intent is evaluating expressions.
- Show C the expression 4 + 6, and C evaluates it to 10. Show it the expression states = 50, and C evaluates it to 50. Evaluating that expression has the side effect of changing the states variable to 50.

Statements - Side Effects and Sequence Points

- A sequence point is a point in program execution at which all side effects are evaluated before going on to the next step.
- In C, the semicolon in a statement marks a sequence point.
- That means all changes made by assignment operators, increment operators, and decrement operators in a statement must take place before a program proceeds to the next statement.

Statements - Side Effects and Sequence Points

- What's a full expression?
- A full expression is **one that's not a subexpression of a larger expression**.
- Examples of full expressions include the expression in an expression statement and the expression serving as a test condition for a while loop.
- Sequence points help clarify when postfix incrementation takes place.
- Consider, for instance, the following code:

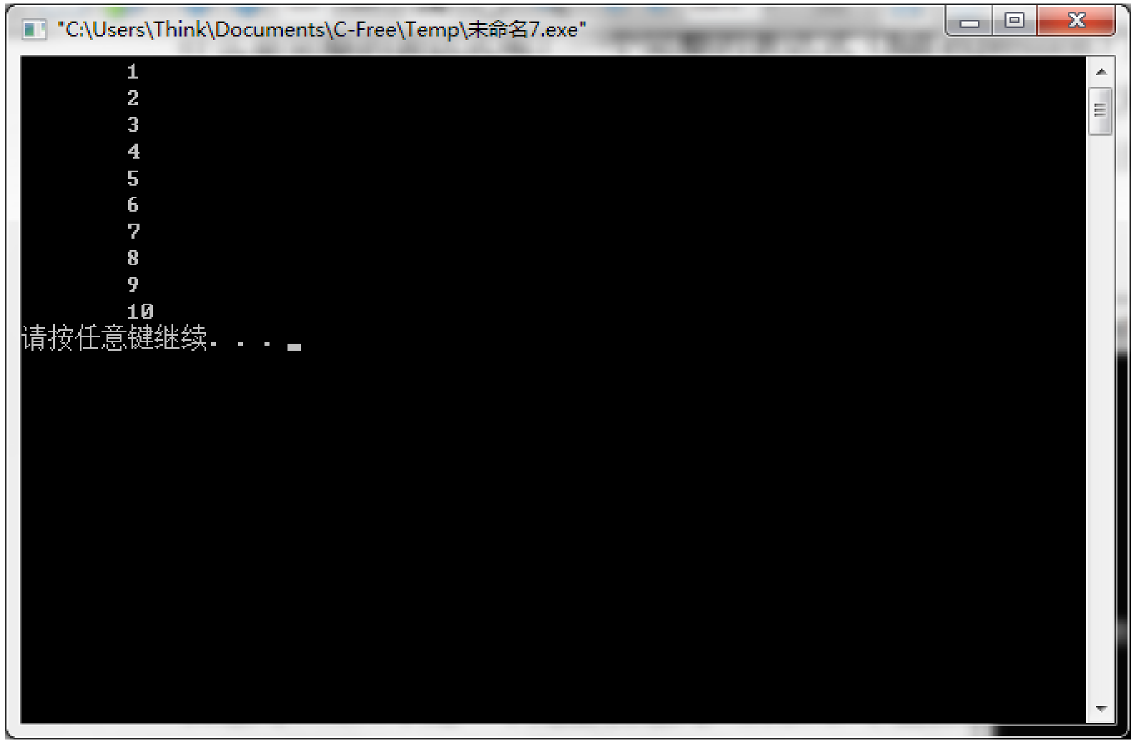
```
while (guests++ < 10)
        printf("%d \n", guests);
```

Statements - Side Effects and Sequence Points

- Sometimes C newcomers assume that "use the value and then increment it" means, in this context, to increment guests after it's used in the printf() statement.
- However, the guests++ < 10 expression is **a full expression** because it is a while loop test condition, so **the end of this expression is a sequence point**.
- Therefore, **C guarantees that the side effect (incrementing guests) takes place before the program moves on to printf()**.
- **Using the postfix form, however, guarantees that guests will be incremented after the comparison to 10 is made**.

Listing 5.13. The addemup.c Program

```
#include <stdio.h>
int main(void)
{
        int guests;
        guests = 0;

        while (guests++ < 10)           /* while */
                printf("   %d\n", guests);  /* function statement */

        return 0;
}
```



"C:\Users\Think\Documents\C-Free\Temp\未命名7.exe"

```
        1
        2
        3
        4
        5
        6
        7
        8
        9
        10
请按任意键继续. . .
```

Statements - Side Effects and Sequence Points

- Now consider this statement:

  y = (4 + x++) + (6 + x++);

- The expression *4 + x++* is not a full expression, so C does not guarantee that x will be incremented immediately after the subexpression *4 + x++* is evaluated.
- Here, the full expression is the entire assignment statement, and the semicolon marks the sequence point, so all that C guarantees is that x will have been

C does not specify whether x is incremented after each subexpression is evaluated or only after all the expressions have been evaluated, which is why you should avoid statements of this kind.

Compound Statements (Blocks)

- A compound statement is two or more statements grouped together by enclosing them in braces; it is also called **a block**.
- The shoes2.c program used a block to let the while statement encompass several statements. Compare the following program fragments:

Compound Statements (Blocks)

```
/* fragment 1 */
index = 0;
while (index++ < 10)
    sam = 10 * index + 2;
printf("sam = %d\n", sam);

/* fragment 2 */
index = 0;
while (index++ < 10)
{

}
```

•In fragment 1, only the assignment statement is included in the while loop. In the absence of braces, a while statement runs from the while to the next semicolon. The printf() function will be called just once, after the loop has been completed.

•In fragment 2, the braces ensure that both statements are part of the while loop, and printf() is called each time the loop is executed. The entire

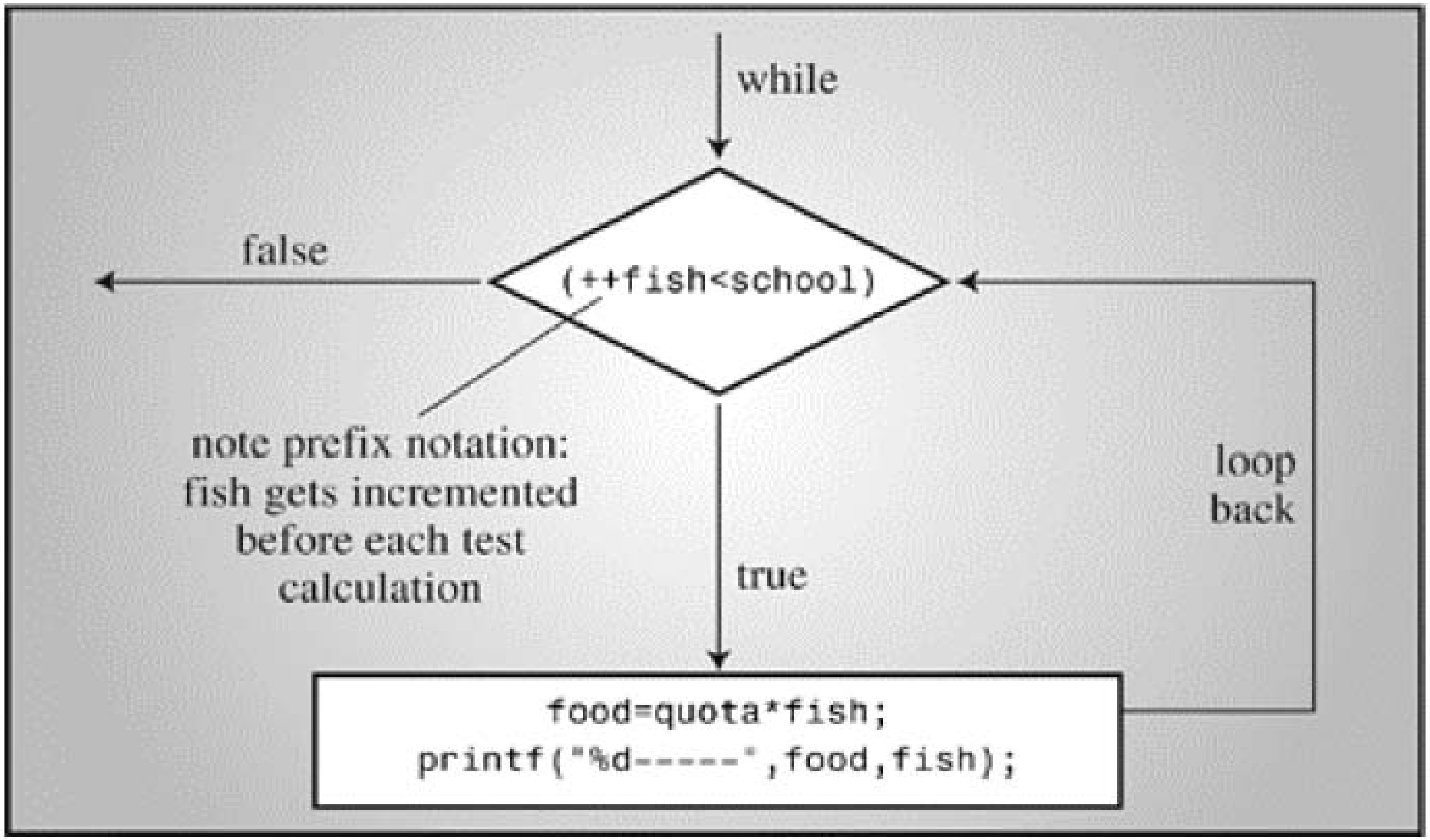
e

structure of a while statement

•To sum up, use indentation as a tool to point out the structure of a program to the reader.

Compound Statements (Blocks)

Figure 5.7. A `while` loop with a compound statement.


while
false
(++fish<school)
note prefix notation: fish gets incremented before each test calculation
true
loop back
food=quota*fish;
printf("%d-----",food,fish);

Summary: Expressions and Statements

- Expressions:
- An expression is a combination of operators and operands.
- The simplest expression is just a constant or a variable with no operator, such as 22 or beebop.
- More complex examples are 55 + 22 and vap = 2 * (vip + (vup = 4)).

Summary: Expressions and Statements

- Statements:
- A statement is a command to the computer. There are simple statements and compound statements. Simple statements terminate in a semicolon, as in these examples:
- Declaration statement: *int toes;*
- Assignment statement: *toes = 12;*
- Function call statement: *printf("%d\n", toes);*
- Structured statement: *while (toes < 20)*

  toes = toes + 2;
- NULL statement: *; /* does nothing */*

Summary: Expressions and Statements

- Compound statements, or blocks, consist of one or more statements (which themselves can be compound statements) enclosed in braces.
- The following while statement contains an example:

```
while (years < 100)
{
    wisdom = wisdom * 1.05;
    printf("%d %d\n", years, wisdom);
    years = years + 1;
}
```

Type Conversions

- Statements and expressions should normally use variables and constants of just one type.
- If, however, you mix types, C doesn't stop dead in its tracks the way, say, Pascal does.
- Instead, it **uses a set of rules to make type conversions automatically**.
- This **can be a convenience**, but **it can also be a danger**, especially if you are mixing types inadvertently.
- It is a good idea to have at least some knowledge of the type conversion rules.

The Basic Rules

- When appearing in an expression, **char** and **short**, both **signed** and **unsigned**, are automatically converted to **int** or, if necessary, to **unsigned int**. (If short is the same size as int, unsigned short is larger than int; in that case, unsigned short is converted to unsigned int.) Under K&R C, but not under current C, float is automatically converted to double. Because they are conversions to larger types, they are called **promotions**.
- In any operation involving two types, **both values are converted to the higher ranking of the two types**.

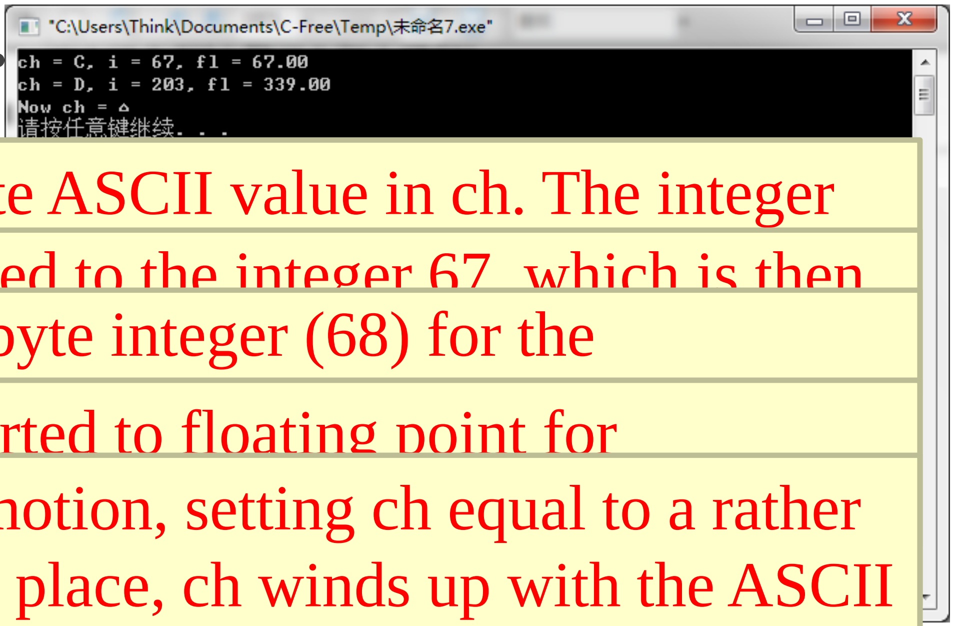
The Basic Rules

- The ranking of types, **from highest to lowest**, is long double, double, float, unsigned long long, long long, unsigned long, long, unsigned int, and int. One possible exception is when long and int are the same size, in which case unsigned int outranks long. The short and char types don't appear in this list because they would have been already promoted to int or perhaps unsigned int.
- In an assignment statement, the final result of the calculations is converted to the type of the variable being assigned a value. This process can result in promotion, as described in rule 1, or **demotion**, in which a value is converted to a lower-ranking type.

The Basic Rules

- When passed as function arguments, char and short are converted to int, and float is converted to double. This automatic promotion can be overridden by **function prototyping**, as discussed in Chapter 9, "Functions.“
- **Promotion is usually a smooth, uneventful process, but demotion can lead to real trouble**. The reason is simple: The lower-ranking type may not be big enough to hold the complete number. An 8-bit char variable can hold the integer 101 but not the integer 22334. When floating types are demoted to integer types, they are truncated, or rounded toward zero. That means 23.12 and 23.99 both are truncated to 23 and that -23.5 is truncated to -23.

Listing 5.14. The convert.


"C:\Users\Think\Documents\C-Free\Temp\未命名7.exe"

```
ch = C, i = 67, fl = 67.00
ch = D, i = 203, fl = 339.00
Now ch = ▵
请按任意键继续. . .
```

The character 'C' is stored as a 1-byte ASCII value in ch. The integer

The character variable 'C' is converted to the integer 67, which is then

The value of ch is converted to a 4-byte integer (68) for the

The value of ch ('D', or 68) is converted to floating point for

Here the example tries a case of demotion, setting ch equal to a rather large number. After truncation takes place, ch winds up with the ASCII code for the hyphen character.

```
        printf("ch = %c, i = %d, fl = %2.2f\n", ch, i, fl); /* line 10 */
        ch = ch + 1;                                         /* line 11 */
        i = fl + 2 * ch;                                     /* line 12 */
        fl = 2.0 * ch + i;                                   /* line 13 */
        printf("ch = %c, i = %d, fl = %2.2f\n", ch, i, fl); /* line 14 */
        ch = 5212205.17;                                     /* line 15 */
        printf("Now ch = %c\n", ch);

        return 0;
}
```

The Cast Operator

- You should usually **steer clear of automatic type conversions**, especially of demotions, but sometimes it is convenient to make conversions, provided you exercise care. The type conversions we've discussed so far are done automatically.
- However, it is possible for you to demand the precise type conversion that you want or else document that you know you're making a type conversion.
- The method for doing this is called **a cast** and consists of preceding the quantity with the name of the desired type in parentheses.

The Cast Operator

- This is the general form of a cast operator:

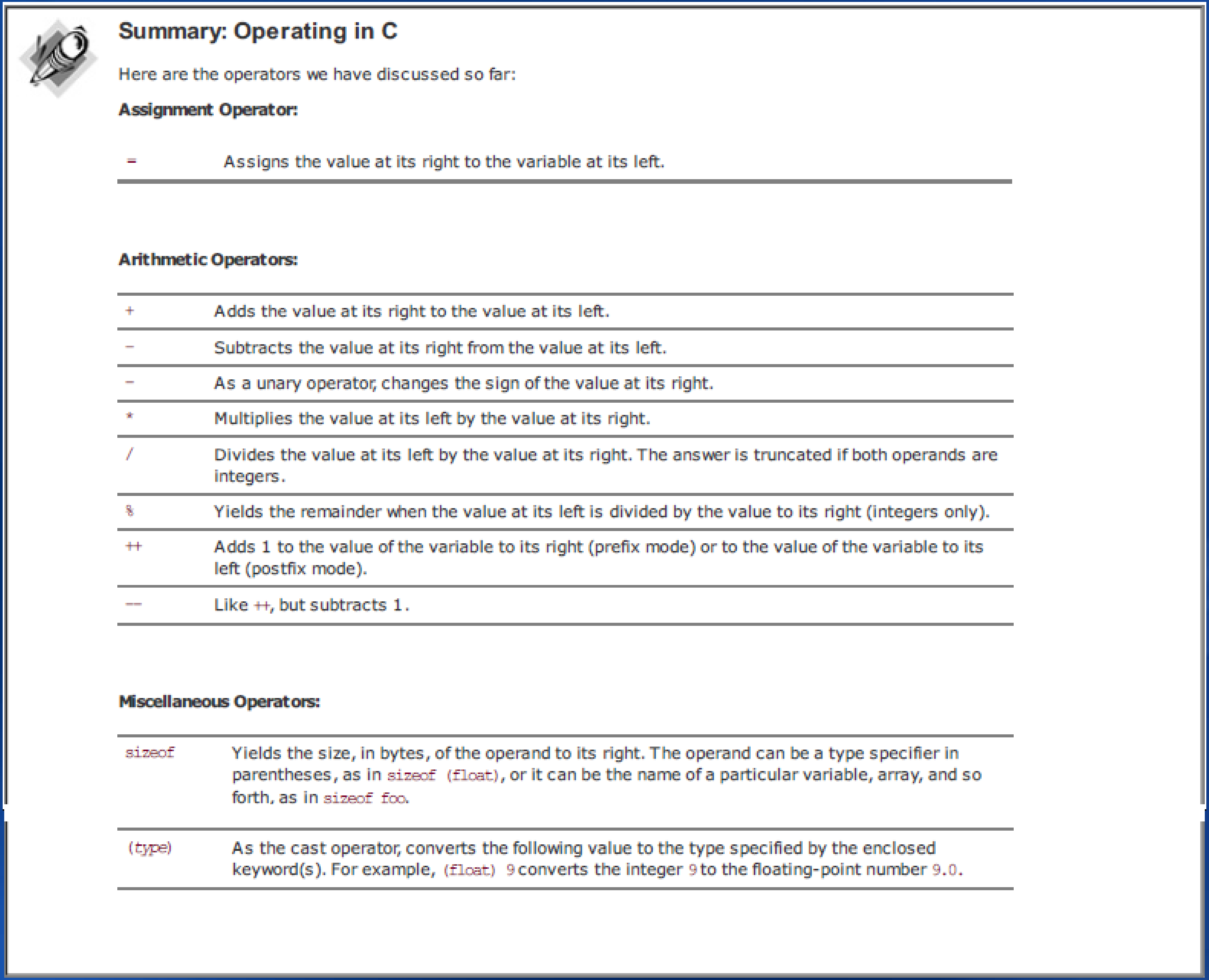
  (type)
- The actual type desired, such as long, is substituted for the word type.
- Consider the next two code lines, in which mice is an int variable. The second line contains two casts to type int.

  ```
  mice = 1.6 + 1.7;
  mice = (int) 1.6 + (int) 1.7;
  ```
- The first example uses automatic conversion.

The Cast Operator

- First, 1.6 and 1.7 are added to yield 3.3. This number is then converted through truncation to the integer 3 to match the int variable.
- In the second example, 1.6 is converted to an integer (1) before addition, as is 1.7, so that mice is assigned the value 1+1, or 2.
- Neither form is intrinsically more correct than the other; you have to consider the context of the programming problem to see which makes more sense.

The C philosophy is to avoid putting barriers in your way and to give you the responsibility of not abusing that freedom.



Summary: Operating in C

Here are the operators we have discussed so far:

Assignment Operator:

=	Assigns the value at its right to the variable at its left.

Arithmetic Operators:

+	Adds the value at its right to the value at its left.
-	Subtracts the value at its right from the value at its left.
-	As a unary operator, changes the sign of the value at its right.
*	Multiplies the value at its left by the value at its right.
/	Divides the value at its left by the value at its right. The answer is truncated if both operands are integers.
%	Yields the remainder when the value at its left is divided by the value to its right (integers only).
++	Adds 1 to the value of the variable to its right (prefix mode) or to the value of the variable to its left (postfix mode).
--	Like ++, but subtracts 1.

Miscellaneous Operators:

sizeof	Yields the size, in bytes, of the operand to its right. The operand can be a type specifier in parentheses, as in sizeof (float), or it can be the name of a particular variable, array, and so forth, as in sizeof foo.
(type)	As the cast operator, converts the following value to the type specified by the enclosed keyword(s). For example, (float) 9 converts the integer 9 to the floating-point number 9.0.

Function with Arguments

- The next step along the road to function mastery is learning how to write your own functions that use arguments.
- Let's preview that skill now. (At this point, you might want to review the butler() function example near the end of Chapter 2, "Introducing C"; it shows how to write a function without an argument.)

Listing 5.15. The pound.c Program

```
/* pound.c -- defines a function with an argument */
#include <stdio.h>
void pound(int n);              /* ANSI prototype */
int main(void)
{
        int times = 5;
        char ch = '!';          /* ASCII code is 33 */
        float f = 6.0;
        pound(times);           /* int argument */
        pound(ch);              /* char automatically -> int */
        pound((int) f);         /* cast forces f -> int */
        return 0;
}
void pound(int n)               /* ANSI-style function header */
{                               /* says takes one int argument */
        while (n-- > 0)
        printf("#");
        printf("\n");
}
```

Function with Arguments

- First, let's examine the function heading:

  void pound(int n)
- If the function took no arguments, the parentheses in the function heading would contain the keyword **void**.
- Because the function takes one type int argument, the parentheses contain a declaration of an int variable called n.
- You can use any name consistent with C's naming rules.

Function with Arguments

- Declaring an argument creates a variable called **the formal argument** or **the formal parameter**.
- In this case, the formal parameter is the int variable called n. Making a function call such as pound(10) acts to assign the value 10 to n.
- In this program, the call pound(times) assigns the value of times (5) to n. We say that **the function call passes a value**, and this value is called **the actual argument** or **the actual parameter**, so the function call pound(10) passes the actual argument 10 to the function, where 10 is assigned to the formal parameter (the variable n).
- That is, the value of the times variable in main() is copied to the new variable n in pound().

Arguments versus Parameters

- Although the terms argument and parameter often have been used interchangeably, the C99 documentation has decided to use the term argument for actual argument or actual parameter and the term parameter for formal parameter or formal argument.
- With this convention, we can say that parameters are variables and that arguments are values provided by a function call and assigned to the corresponding parameters.

Function with Arguments

- Variable names are **private** to the function.
- This means that a name defined in one function doesn't conflict with the same name defined elsewhere.
- If you used times instead of n in pound(), that would create a variable distinct from the times in main().
- That is, you would have two variables with the same name, but the program keeps track of which is which.

The Function Calls

- The first one is pound(times), and, as we said, it causes the times value of 5 to be assigned to n. This causes the function to print five pound signs and a newline.
- The second call is pound(ch). Here, ch is type char. It is initialized to the ! character, which, on ASCII systems, means that ch has the numerical value 33. The automatic promotion of char to int converts this, on this system, from 33 stored in 1 byte to 33 stored in 4 bytes, so the value 33 is now in the correct form to be used as an argument to this function.
- The last call, pound ((int) f), uses a type cast to convert the type float variable f to the proper type for this argument.

The Function Calls

- Suppose you omit the type cast. With modern C, the program will make the type cast automatically for you.
- That's because of the ANSI prototype near the top of the file:

  void pound(int n); / ANSI prototype */*
- A **prototype** is **a function declaration that describes a function's return value and its arguments**. This prototype says two things about the pound() function:
- The function has no return value.
- The function takes one argument, which is a type int value.

The Function Calls

- Because the compiler sees this prototype before pound() is used in main(), the compiler knows what sort of argument pound() should have, and it inserts a type cast if one is needed to make the actual argument agree in type with the prototype.
- For example, the call pound(3.859) will be converted to pound(3).

Key Concepts

- C uses operators to provide a variety of services.
- Each operator can be characterized by **the number of operands it requires**, **its precedence**, and its **associativity**.
- The last two qualities determine which operator is applied first when the two share an operand.
- Operators are combined with values to produce expressions, and every C expression has a value.

Key Concepts

- If you are not aware of operator precedence and associativity, you may construct expressions that are illegal or that have values different from what you intend; that would not enhance your reputation as a programmer.
- C allows you to write expressions combining different numerical types.
- But arithmetic operations require operands to be of the same type, so C makes automatic conversions.
- However, it's good programming practice not to rely upon automatic conversions.

Key Concepts

- Instead, make your choice of types explicit either by choosing variables of the correct type or by using typecasts.
- That way, you won't fall prey to automatic conversions that you did not expect.

Summary - Operators

- C has many operators, such as the assignment and arithmetic operators discussed in this chapter.
- In general, an operator operates on one or more operands to produce a value.
- Operators that take one operand, such as the minus sign and sizeof, are termed unary operators.
- Operators requiring two operands, such as the addition and the multiplication operators, are called binary operators.

Summary - Expressions

- Expressions are combinations of operators and operands.
- In C, every expression has a value, including assignment expressions and comparison expressions.
- Rules of operator precedence help determine how terms are grouped when expressions are evaluated.
- When two operators share an operand, the one of higher precedence is applied first.
- If the operators have equal precedence, the associativity (left-right or right-left) determines which operator is applied first.

Summary - Statements

- Statements are complete instructions to the computer and are indicated in C by a terminating semicolon.
- So far, you have worked with declaration statements, assignment statements, function call statements, and control statements.
- Statements included within a pair of braces constitute a compound statement, or block.
- One particular control statement is the while loop, which repeats statements as long as a test condition remains true.

Summary – Type Conversions

- In C, many type conversions take place automatically.
- The char and short types are promoted to type int whenever they appear in expressions or as function arguments.
- The float type is promoted to type double when used as a function argument.
- Under K&R C (but not ANSI C), float is also promoted to double when used in an expression. When a value of one type is assigned to a variable of a second type, the value is converted to the same type as the variable.

Summary – Type Conversions

- When larger types are converted to smaller types (long to short or double to float, for example), there might be a loss of data.
- In cases of mixed arithmetic, smaller types are converted to larger types following the rules outlined in this chapter.

Summary – Function with Argument

- When you define a function that takes an argument, you declare a variable, or formal argument, in the function definition.
- Then the value passed in a function call is assigned to this variable, which can now be used in the function.

Questions & Homeworks