Agenda

Structured Program Development in C

Introduction

- Before writing a program to solve a particular problem, we must have a thorough understanding of the problem and a carefully planned solution approach. The next two chapters discuss techniques that facilitate the development of structured computer programs.

Structured Program Development in C

Algorithms

- The solution to any computing problem involves executing a series of actions in a specific order. A procedure for solving a problem in terms of the actions to be executed and the order in which these actions are to be executed is called an algorithm.

- Specifying the order in which the statements are to be executed in a computer program is called program control.

Structured Program Development in C

Pseudocode

- Pseudocode is an artificial and informal language that helps you develop algorithms.
- Pseudocode consists purely of characters, so you may conveniently type pseudocode programs into a computer using an editor program. A carefully prepared pseudocode program may be easily converted to a corresponding C program.

Normally, statements in a program are executed one after the other in the order in which they're written. This is called sequential execution. Variaous C statements we'll soon discuss enable you to specify that the next statement to be executed may be other than the next one in sequence. This is called transfer of control.

All programs could be written in terms of only three control structures, namely the sequence structure, the selection structure and the repetition structure. The sequence structure is simple- unless directed otherwise, the computer executes C statements one after the other in the order in which they're written. The flowchart segment of Fig. 1.1 illustrated C's sequence structure.

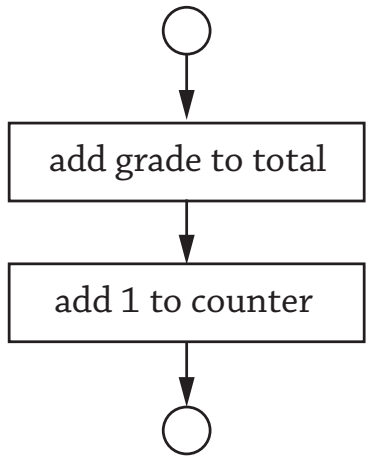

Fig. 1.1 | Flowcharting C's sequence structure

Control Structures--Flowcharts

- A flowchart is a graphical representation of an algorithm or of a portion of an algorithm. Flowcharts are drawn using certain special-purpose symbols such as rectangles, diamonds, rounded rectangles, and small circles, these symbols are connected by arrows called flowlines.

- Consider the flowchart for the sequence structure in Figure 1.1. We use rectangle symbol, also called the action symbol, to indicate any type of action including calculation, or an input/output operation. The flowlines in the figure indicate the order in which the actions are performed.

- When drawing a flowchart that represent a complete algorithm, a rouned rectangle symbol containing the word “Begin” is the first symbol used in the flowchart; a rouned rectangle symbol containing the word “End” is the last symbol used. When drawing only a portion of an algorithm as in Fig. 1.1, the rouned rectangle symbols are omitted in favor of using small circle symbols, also called connector symbols.

- Perhaps the most important flowcharting symbol is the diamond symbol, also called the decision symbol, which indicates that a decision is to be made.

- C provides three types of selection structures in the form of statements.

- The if selection statement either selects an action if a condition is true or skips the action if the condition is false. The if...else statement performs an action if a condition is true and performs a different action if the condition is false. The switch selection statement peforms one of many different actions, depending on the value of an expression.

- The if statement is called a single-selection statement because it selects or ignore a single action. The if...else statement is called a double-selection statement because it selects between two different actions. The switch statement is called a multiple-selection statement because it selects among many different actions.

- C provides three types of repetition structures in the form of statements, namely while, do...while, and for.

- C has only seven control statements: sequence, three types of selection and three types of repetition. Each C program is formed by combining as many of each type of control statement as is appropriate for the algorithm the program implements.

- As with the sequence structure of Fig. 1.1, we'll see that the flowchart representation of each control statement has two small circle symbols, one at the entry point to the control statement and one at the exit point.

- The control statement flowchart segments can be attached to one another by connecting the exit point of one control statement to the entry point of the next. This is much like the way in which a child stacks building blocks, so we call this control-statement stacking. We'll learn that there's only one other way control statements may be connected-a method called control-statement nesting.

- Thus, any C program we'll ever need to build can be constructed from only seven different types of control statements combined in only two ways.

The if Selection Statement

- Selection statements are used to choose among alternative courses of action. For example, suppose the passing grade on an exam is 60. The pseudocode statement

If student's grade is greater than or equal to 60
Print "Passed"

determines whether the condition "student's grade is greater than or equal to 60" is true or false. If the condition is true, then "Passed" is printed, and the next pseudecode statement in order is "performed". If the condition is false, the printing is ignored, and the next pseudocode statement in order is performed.

- The preceding pseudocode If statement may be written in C as

```
if( grade >= 60 ){
  printf( "Passed\n" );
}
```

The flowchart of Fig. 1.2 illustrates the single-selection if statement. This flowchart contains what is perhaps the most important flowcharting symbol-the diamond symbol, also called the decision symbol, which indicates that a decision is to be made.

The decision symbol contains an expression, such as a condition, that can be either true of false. The decision symbol has two flowlines emerging from it. One indicates the direction to take when the expression in the symbol is true and the other the direction to take when the expression is false.

In fact, a decision can be based on any expression-if the expression evaluates to zero, it's treated as false, and if it evaluates to nonzero, it's treated as true.

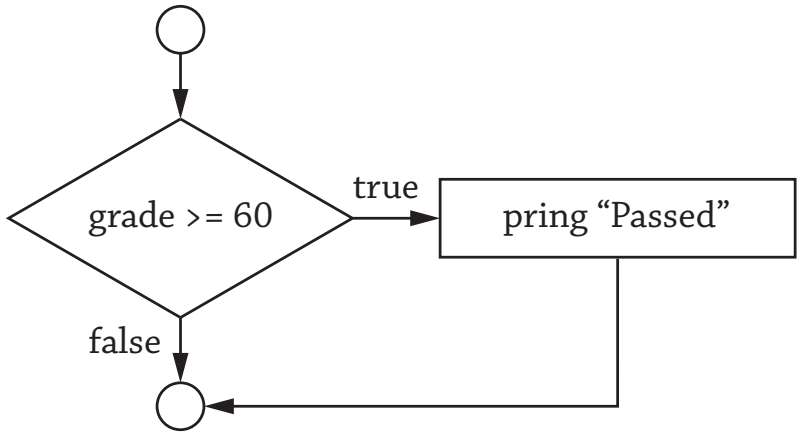

Fig. 1.2 | Flowcharting the single-selection if statement

The if…else Selection Statement

- The if…else selection statement allows you to specify that different actions are to be performed when the condition is true and when it's false. For example, the pseudocode statement

```
If student's grade is greater than or equal to 60
  Print "Passed"
else
  Print "Failed"
```

prints Passed if the student's grade is greater than or equal to 60 and Failed if the student's grade is less than 60. In either case, after printing occurs, the next pseudocode statement in sequence is "performed".

- The preceding pseudocode If…else statement may be written in C as

```
if( grade >= 60 ){
  printf( "Passed\n" );
}
else{
  printf( "Failed\n" );
}
```

The flowchart of Fig. 1.3 illustrates the flow of control in the if...else statement. Once again, besides small circles and arrows, the only symbols in the flowchart are rectangles (for actions) and a diamond (for decision).

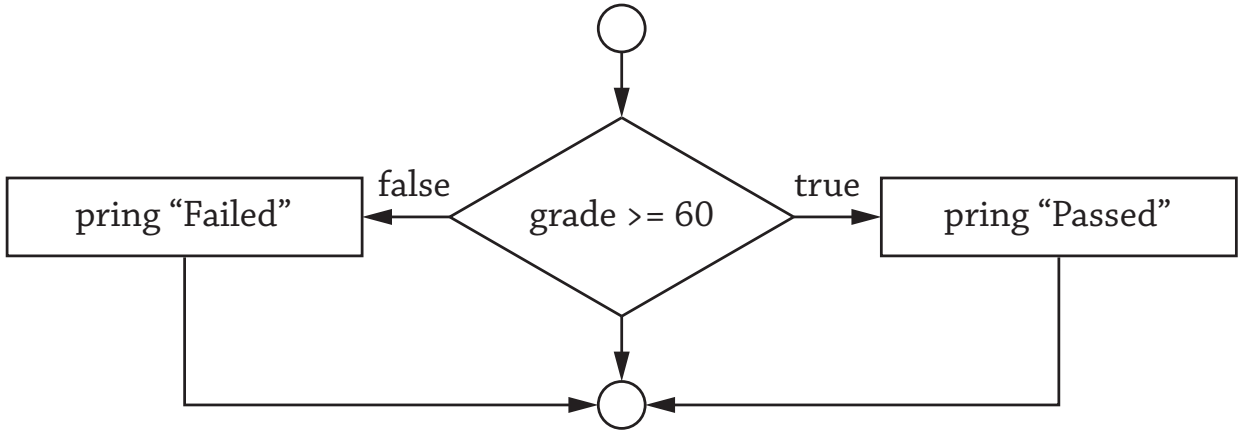

Fig. 1.3 | Flowcharting the double-selection if...else statement

The if...else Selection Statement

C provides the conditional operator (?:), which is closely related to the if...else statement. The conditional operator is C's only ternary operator-it takes three operands. The first operand is condition. The second operand is the value for the entrie conditional expression if the condition true and the operand is the value for the entire conditional expression if the condition is false. For example, the puts statement

```
puts( grade >= 60 ? “Passed” : “Failed” );
```

contains as its second argument a conditional expression that evaluates to the string “Passed” if the conditional grade >= 60 is true and to the string “Failed” if the condition is false.

The second and third operands in a conditional expression can also be actions to be executed. For example, conditional expression

```
grade >= 60 ? puts( “Passed” ) : puts( “Failed” );
```

is read, “If the grade is greater than or equal to 60, then puts(”Passed”), otherwise puts(“Failed”).

The if…else Selection Statement--Nested if…else Statements

Nested if…else statements test for multipe cases by placing if…else statements inside if…else statements.

- For example, the following pseudocode statement will print A for exam grades greater than or equal to 90, B for grades greaters than or equal to 80 (but less than 90), …, and F for all other grades.

```
If student's grade is greater than or equal to 90
      Print "A"
else
   If student's grade is greater than or equal to 80
         Print "B"
   else
      If student's grade is greater than or equal to 70
            Print "B"
      else
         If student's grade is greater than or equal to 60
               Print "D"
         else
               Print "D"
```

- This pseudocode may be written in C as

```
if ( grade >= 90 ){
  puts( “A” );
}
else{
  if ( grade >= 80 ){
    puts( “B” );
  }
  else{
    if ( grade >= 70 ){
      puts( “C” );
    }
    else{
      if ( grade >= 60 ){
        puts( “D” );
      }
      else{
        puts( “D” );
      }
    }
  }
}
```

- You may prefer to write the preceding if statemet as

```
if ( grade >= 90 ){
  puts( “A” );
}
else if ( grade <= 80 ) {
  puts( “B” );
}
else if ( grade >= 70 ){
  puts( “C” );
}
else if ( grade >= 60 ){
  puts( “D” );
}
else{
  puts( “D” );
}
```

- The if selection statement expect only one statement in its body--if you have only one statement in the if's body, you don't need the enclose it in braces. To include several statements in the body of an if, you must enclose the set of statements in braces. A set of statements contained within a pair of braces is called a compound statement or a block.

- The following example includes a compound statement in the else part of an if...else statement.

```
if ( grade >= 60 ){
  puts( "Passed" );
}
else{
  puts( "Failed" );
  puts( "You must take this course again." );
}
```

- A syntax error is caught by the compiler. A logic error has its effect at execution time. A fatal logic error causes a program fail or terminate prematurely. A nonfatal logic error allows a program to continue executing but to produce incorrect result.

Structured Program Development in C

The while Repetition Statement

- A repetition statement (also called an iteration statement) allows you to specify that an action is to be repeated while some condition remains true.
- The statement(s) contained in the while repetition statement constitute the body of the while. The while statement only may be a single statement or a compound statement.
- Eventually, the condition will become false. At this point, the repetition terminates, and the first pseudo-code statement after the repetition structure is executed.

Common Programming Error

Not providing in the body of a while statement an action that eventually causes condition in the while to become false. Normally, such a repetition structure will never terminate--an error called an “infinite loop”.

As an example of a while statement, consider a program segment designed to find the first power of 3 larger than 100. Suppose the integer variable product has been initialized to 3. When the following while repetition statement finishes executing, product will contain the desired answer:

```
product = 3;
while( product <= 100 ) {
  product = 3 * product;
}
```

The flowchart of Fig. 1.4 illustrates the flow of control in the while repetition statement. Once again, note that the flowchart contains only a rectangle symbol and a diamond symbol. The flowchart clearly shows the repetition. The flowline emerging from the rectangle wraps back to the decision, which is tested each time through the loop until the decision eventually becomes false. At this point, the while statement is exited and control passes to the next statement in the program.

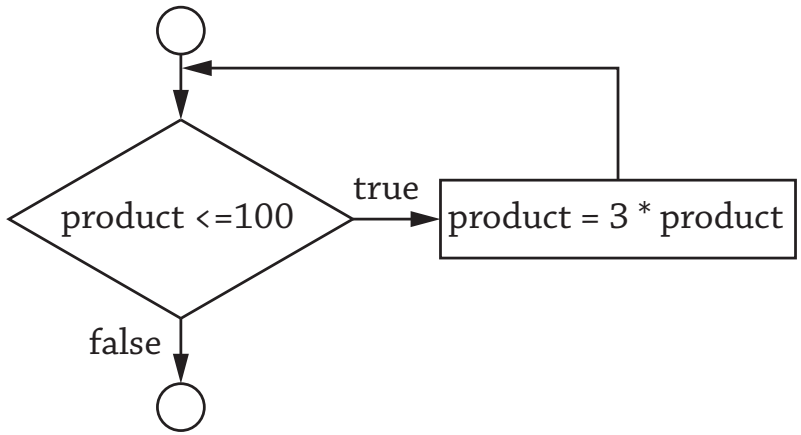

Fig. 1.4 | Flowcharting the while repetition if statement

Formulating Algorithms Case Study I: Counter-Controlled Repetition

- To illustrate how algorithms are developed, we solve several variations of a class-averaging problem. Consider the following problem statement:

A class of ten students took a quiz. The grades (integers in the range 0 to 100) for this quiz are available to you. Determine the class average on the quiz.

- Let's use pseudocode to list the actions to execute and specify the order in which these actions should execute. We use counter-controlled repetition to input the grades one at a time. This technique uses a variable called a counter to specify the number of times a set of statement should execute. In this example, repetition terminates when the counter exceeds 10.

- In this section we simply present the pseudocode algorithm (Fig. 1.5) and the corresponding C program (Fig. 1.6). Counter-controlled repetition is often called definite repetition because the number of repetitions is known before the loop begins executing.

```
1  Set total to zero
2  Set grade counter to one
3
4  While grade counter is less than or equal to ten
5     Input the next grade
6     Add the grade into the total
7     Add one to the grade counter
8
9     Set the class average to the total divided by ten
10    Print the class average
```

Fig. 1.5 Pseudocode algorithm that uses counter-controlled repetition to solve the class average problem

```
1   // Class average program with counter-controlled repetition.
2   #include <stdio.h>
3
4   // function main begins program execution
5   int main( void )
6   {
7      unsinged int counter; // number of grade to be entered next
8      int grade; // grade value
9      int total; // sum of grades entered by user
10     int average; // average of grades
11
12     // initialization phase
13     total = 0; // initialize total
14     counter = 1; // initialize loop counter
15
16     // processing phase
17     while ( counter <= 10 ) { // loop 10 times
18        printf( “%s”, “Enter grade: ” ); // prompt for input
19        scanf( “%d”, &grade ); // read grade from user
20        total = total + grade; // add grade to total
21        counter = counter + 1; // increment counter
22     } //end while
23
24     // termination phase
25     average = total / counter; // integer division
26
27     printf( “Class average is %d\n”, average ); // prompt for input
28  } //end function main
```

- Variables used to store totals should normally be initialized to zero before being used in a program; otherwise the sum would include the previous value stored in the total's memory location. Counter variables are normally initialized to zero or one, depending on their use.

- An uninitialized variable contains a "garbage" value--the value last stored in the memory location reserved for that variable.

Common Programming Error

If a counter or total isn't initialized, the results of your program will probably will be incorrect. This is an example of a logic error.

Structured Program Development in C

Top-Down, Stepwise Refinment Case Study I: Sentinel-Controlled Repetition

- Let's generalize the class-average problem. Consider the following problem:

  Develop a class-averaging program that will process an arbitrary number of grades each time the program is run.

- The program must process an arbitrary number of grades. How can the program determine when to stop the input grades. How will it know when to calculate and print the class average ?

- One way to solve this problem is to use a special vlaue called a sentinel value (also called a signal value, a dummy value, or a flag value) to indicate end of data entry.

- The user types in grades until all legitimate grades have been entered. The user then types the sentinel value to indicate the last grade has been entered. Sentinel-controlled repetition is often called indefinite repetition because the number of repetitions isn't known before the loop begins executing.

- Clearly, the sentinel value must be chosen so that is cannot be confused with an acceptable input value. Because grades on a quiz are normally nonnegative integers, -1 is an acceptable sentinel value for this problem.

Top-Down, Stepwise Refinment Case Study I: Sentinel-Controlled Repetition

We approach the class-average program with a technique called top-down, stepwise refinement, a technique that's essential to the development of well-structured programs. We begin with a pseudocode representation of the top:

Determine the class average for the quiz

The top is a single statement that conveys the program's overall function. As such, the top is, in effect, a complete representation of a program. Unfortunately, the top rarely conveys a sufficient amount of detail for writing the C program. So we not begin the refinement process. We divide the top into a series of smaller tasks and list these in the order in which they need to be performed. This reults in the following first refinement.

Initialize variables

Input, sum, and count the quiz grades

Calculate and print the class average

To proceed to the next level of refinement, i.e., the second refinment, we commit to specific variables. We need a running total of the numbers, a count of how many numbers have been processed, a variable to receive the value of each grade as it's input and a variable to hold the calculated average. The pseudocode statement

Initialize variables

may be refined as follows:

Intialize total to zero
Initialize counter to zero

- The pseudocode statement

Input, sum and count the quiz grades

requires a repetition structure that successively inputs each grade. Because we do not know in advance how many grades are to be processed, we'll use sentinel-controlled repetition. The user will enter legitimate grades one at a time. After the legitimate grade is typed, the user will type the sentinel value. The program will test for this value after each grade is input and will terminate the loop when the sentine-lis entered. The refinement of the preceding pseudocode statement is then

Input the first grade
While the user has no as yet entered the sentinel
  Add this grade in to the running total
  Add one to the grade counter
  Input the next grade (possibly the sentinel)

- The pseudocode statement

Calculate and print the class average

may be refined as follows:

If the counter is not equal to zero
  Set the average to the total divided by the counter
  Print the average
else
  Print "No grades were entered"

- Notice that we're beging careful here to test for the possibility of division by zero--a fatal error that if undeteceted would cause the program to fail (often called crashing). The complete second refinement is shown in Fig. 1.7.

Good Programming Practice
When performing division by an expression whose value could be zero, explicitly test for this case and handle it appropriately in your program (such as printing an error message) rather than allowing the fatal error to occur.

```
1    Intialize total to zero
2    Initialize counter to zero
3
4    Input the first grade
5    While the user has no as yet entered the sentinel
6       Add this grade in to the running total
7       Add one to the grade counter
8        Input the next grade (possibly the sentinel)
9
10   If the counter is not equal to zero
11      Set the average to the total divided by the counter
12      Print the average
13   else
14       Print "No grades were entered"
```

Fig. 1.7 Pseudocode algorithm that uses sentinel-controlled repetition to solve the class-average problem

Software Engineering Observation
You terminate the top-down, stepwise refinment process when the pseudocode algorithm is specified in sufficient detail for you to be able to convert the pseudocode to C. Implementing the C program is then normally straightforward.

- The C program is shown in Fig. 1.8. Although only integer grades are entered, the averaging calculation is likely to produce a number with a decimal point. The type int cannot represent such a number.

- The program introduces the data type float to handle numbers with decimal points (called floating-point numbers) and introduces a special operator called a cast operator to handle the averaging calculation.

```
1   // Class average program with sentinel-controlled repetition.
2   #include <stdio.h>
3
4   // function main begins program execution
5   int main( void )
6   {
7      unsinged int counter; // number of grade to be entered next
8      int grade; // grade value
9      int total; // sum of grades entered by user
10     float average; // number with decimal point for average
11
12     // initialization phase
13     total = 0; // initialize total
14     counter = 0; // initialize loop counter
15
16     // processing phase
17     printf( “%s”, “Enter grade, -1 to end: ” ); // prompt for input
18     scanf( “%d”, &grade ); // read grade from user
19
```

Fig. 1.8. Class-average program with sentinel-controlled repetition

```
20      // loop while sentinel valu not yet read from user
21      while ( grade != -1 ) {
22        total = total + grade; // add grade to total
23        counter = counter + 1; // increment counter
24
25        // get next grade from user
26        printf( “%s”, “Enter grade, -1 to end: ” ); // prompt for input
27        scanf( “%d”, &grade ); // read grade from user
28      } //end while
29
30      // termination phase
31      // if user entered at least one grade
32      if ( counter != 0 ) {
33        // calculate average of all grades entered
34        average = ( float ) total / counter // avoid truncation
35
36        // display average with two digits of precision
37        printf( “Class average is %.2f\n”, average );
38      } // end if
39      else{
40        puts( no grades were entered” ); // prompt for input
41      } //end else
42    } //end function main
```

Fig. 1.8. Class-average program with sentinel-controlled repetition

- The variable average is defined to be of type float to capture the fractional result of our calculation. However, the reuslt of the calculation total / counter is an integer because total and counter are both integer variables. Dividing two integer results in integer division in which any fractional part of the calculation is truncated.

- To produce a floating-point calculation with integer values, we must create temporary values that are floating-point numbers. C povides the unary cast operator to accomplish this task.

- The line average = (float) total / counter; includes the cast operator (float), which creates a temporary floating-point copy of its operand, total. The value stored in total is still an integer. Using a cast operator in this manner is called explicit conversion.

- C evaluates arithmetic expressions only in which the data types of the operands are identical. To ensure that the operands are of the same type, the compiler performs an operation called implicit conversion on the selected operands. For example, in an expression containing the data types unsigned int and float, copies of unsigned int operands are made converted to float.

- Each cast operator is a unary operator, i.e., an operator that takes only one operand.

- C also supports unary version of the plus (+) and minus (-) operatos, so you can write expression such as -7 or +5. Cast operations associate from right to left and have the same precedence as other unary operators such as unary + and unary -. This precedence is one level higher than that of the multiplicative operators *, / and %.

- Figure 1.8 uses the printf conversion specifier %.2f to print the value of average. The f specifies that a floating-point value will be printed. The .2 is the precision with which the value will be displayed--with 2 digits to the right of the decimal point.

- If the %f conversion specifier is used (without specifiying the precision), the default precision of 6 is used--exactly as if the conversion specifier %.6f had been used. When floating-point values are printed with precision, the printed value is rounded to the indicated number of the decimal positions. The value in memory is unaltered. When the following statement are executed, the values 3.45 and 3.4 are printed.

```
printf( “%.2f\n”, 3.446 ); // prints 3.45
printf( “%.1f\n”, 3.446 ); // prints 3.4
```

Common Programming Error
Using precision in a conversion specifier in the format control string of a scanf statement is wrong. Precision are used only in printf conversion specifier.

- Although floating-point numbers are not always “100% precise”, they have numerous applications. For example, when we speak of a “normal” body temperature of 98.6, we do not need to be precise to a large number of digits. When we view the temperature on a thermometer and read it as 98.6, it may actually be 98.5999473210643. The point here is that calling this number simply 98.6 is fine for most applications.

- Another way floating-point numbers develop is through division. When we divide 10 by 3, the result is 3.333333… with the sequence of 3s repeating infinitely. The computer allocates only a fixed amount of space to hold such a value, so the stored floating-poing value can be only an approximation.

Common Programming Error
Using floating-point numbers in a manner that assumes they’re represented precisely can lead to incorrect results. Floating-point numbers are represented only approximately by most computers.

Error-Prevention Tip
Do not compare floating-point values for equality.

Assignment Operators

- C provides severel assignment operators for abbreviating assignment expressions. Any statement of the form

  variable = variable operator expression;

  where operator is one of the binary operators +, -, *, / or % can be written in the form

  variable operator = expression;

Assume: int c = 3, d = 5, e = 4, f = 6, g = 12;

Assignment operator	Sample expression	Explanation	Assigns
+=	c += 7	c = c + 7	10 to c
-=	d -= 4	d = d - 4	1 to d
*=	e *= 5	e = e * 5	20 to e
/=	f /= 3	f = f / 3	2 to f
%=	g %= 9	g = g % 9	3 to g

Increment and Decrement Operators

- C also provides the unary increment operator, ++, and the unary decrement operator, --. If a variable c is to be incremented by 1, the increment operator ++ can be used rather than the expressions c = c + 1 or c += 1.
- If increment or decrement operators are placed before a variable, they're referred to as the preincrement or predecrement operators, respectively.
- If increment or decrement operators are placed after a variable, they're referred to as the postincrement or postdecrement operators, respectively.
- Preincrementing (predecrementing) a variable causes the variable to be incremented (decremented) by 1, then its new value is used in the expression in which it appears.
- Postincrementing (postdecrementing) a variable causes the current value of the variable to be used in the expression in which it appears, then the variable value is incremented (decremented) by 1.

Increment and Decrement Operators

- Figure 1.9 demonstrates the difference between the preincrementing and postincrementing version of the ++ operator.

```
1   // Preincrementing and postincrementing.
2   #include <stdio.h>
3
4   // function main begins program execution
5   int main( void )
6   {
7      int c; // define a variable
8
9      // demonstrate postincrement
10     c = 5; // assing 5 to c
11     printf( "%d\n", c );
12     printf( "%d\n", c++  );
13     printf( "%d\n", c );
14
15     // demonstrate preincrement
16     c = 5; // assing 5 to c
17     printf( "%d\n", c );
18     printf( "%d\n", ++c  );
19     printf( "%d\n", c );
20  } // end function main
```

Fig. 1.9. Preincrementing and postincrementing