Chapter Objectives, Concept Recap and Chapter Review that help students to quickly grasp key concepts at strategic points in the book

CHAPTER

2 Variables, Arithmetic Expressions and Input/Output

Chapter Objectives

Upon completion of this chapter, you will be able to:

- Declare variables to be used in C program.
- Read keyboard input from the user.
- Control the output format with the printf statement.
- Construct complex mathematical expressions.

In order for a computer program to be useful, it must have functions for performing calculations as well as providing immediate response to user input. In this chapter you will learn how to handle variables and to perform arithmetic calculations.

Lesson 2.1 Variables: Naming, Declaring, Assigning and Printing Values

Topics

- Naming variables
- Declaring data types
- Using assignment statements
- Displaying variable values
- Elementary assignment statements

Variables are crucial to virtually all C programs. You have learnt about variables in algebra, and you will find that, in C, variables are used in much the same manner.

Suppose, for instance, that you want to calculate the area of 10,000 triangles, all of different sizes. And suppose that the following information is given:

1. The length of each of the three sides.
2. The size of each of the three angles.

8. **Why was so much attention given to the *printf* statement?** There are two reasons. One is that you will write *printf* statements very frequently and other aspects of programming will become easier for you if you are comfortable writing *printf* statements. It is a good idea to become proficient at writing them now so that you can easily go on to other programming issues. The other reason is that improperly written *printf* statements are the source of many errors for beginning programmers. If you understand *printf* statements, you will substantially reduce your programming errors.

Concept Recap

1. The form of a define directive is

```
#define symbolic_name  replacement
```

where *symbolic_name* that occurs throughout the rest of program will be replaced by *replacement* during compilation by the preprocessor.

2. The complete format specification is

```
%[flag][field width][.precision]type
```

where format string components enclosed by [ ] are optional.

3. The output of a floating point number in scientific notation is

```
[sign]d.ddd  e[sign]ddd
```

where d represents a digit. Note that a number in this form is equivalent to

[sign] d.ddd × 10[sign]ddd

Exercises

1. True or false:
   a. The statement `printf("%-3d",123);` displays –123.
   b. The statement `printf("%+2d",123);` displays +12.
   c. The statement `printf("%-2f",123);` displays 12.0.
   d. The statement `printf("%+f.3",123);` displays .123.
   e. The format specification for an `int` type data should not contain a decimal point and precision; for instance, `%8.2d` is illegal.

2. Find errors, if any, in these statements:
   a. `#DEFINE PI 3.1416`
   b. `#define PI 3.1416;`
   c. `#define PI=3.14; More_AccuratePI=3.1416;`
   d. `printf("%f",123.4567);`
   e. `printf("%d %d %f %f",1,2,3.3,4.4);`

5. Hand calculate the values of x, y and z in the following program and then run the program to check your results:

```
#include <stdio.h>
void main(void)
{
     float a=2.5,b=2,c=3,d=4,e=5,x,y,z;
     x=  a *  b -  c + d  /e ;
     y=  a * (b -  c)+ d  /e ;
     z=  a * (b - (c + d) /e) ;
     printf("x= %10.3f, y= %10.3f, z=%10.3f",x,y,z);
}
```

6. Calculate the value of each of the following arithmetic expressions:

```
13/36, 36/4.5, 3.1*4, 3-2.6, 12%5, 32%7
```

Solutions

1. a. False b. False c. True d. False e. True f. False g. False
   h. False i. True j. False k. True l. False m. False
3. a. 30, 30, 30
   b. 31, 31, 30
   c. 32, 33, 33
   d. program crash due to division by zero.
6. 0, 8.0, 12.4, 0.4, 2, 4

Chapter Review

In this chapter, we have learnt how to control the output of program variables using the format specifications. We also discuss how to declare variables in your C program, as well as how to process data using arithmetic operators. Then we use *scanf* to read some values from the keyboard into our program, and use *printf* to print the values of a variable to the screen. Finally, we studied the issues relating to arithmetic operations in C expressions.

Now you can use all that you have learnt in this chapter to write programs that can achieve complex tasks such as scientific calculations.

Simple sample programs consisting of source code accompanied by guided observations, and output

The C language provides several methods for looping. The simplest one is the while loop. A while loop contains just two parts, a test condition part and an execution part. When a program reaches a while statement, the test condition will be checked. If the condition is true, the execution part will be executed and continued to be executed until the test condition becomes false. When the test condition becomes false, the execution part is bypassed and program control is transferred to a point after the end of the while loop.

Look for the line of text in the source code with the keyword `while`. From what you know about statement blocks and relational expressions, can you determine which expression represents the test condition? Which are the statements in the execution part? Look at the output. How many times has the loop been executed? Why did it execute this many times?

Source Code

```
#include <stdio.h>
 void main(void)
 {
     int  i;

     i = 1;

     while (i<= 5 )
     {
        printf (" Loop number %d in the while_loop\n",i);
        i++;
     }

}
```

Test expression.

Statement block is repeatedly executed until test expression becomes false.

Incrementing counter variable.

Output

```
Loop number 1 in the while_loop
Loop number 2 in the while_loop
Loop number 3 in the while_loop
Loop number 4 in the while_loop
Loop number 5 in the while_loop
```

Explanation

1. **What is the meaning of `while(i<=5)` {statements}?** It means that, while the variable `i` is less than or equal to 5, the statements between the braces are executed repeatedly. When the variable `i` becomes greater than 5, the statements between the braces are not executed. In general, the structure of a C *while loop* is

and the way the loops are executed? (Hint: The first statement indicates how the loop begins, the second statement indicates how the loop ends, and the third statement indicates how the loop goes from the beginning to the end.) In the first loop, note where semicolons are located. In the second loop, note the use of braces. Can you figure out why braces are used in one loop and not the other?

Source Code

```
#include <stdio.h>
void main(void)
{
    int day, hour, minutes;

     for(day=1; day<=3; day++)
         printf("Day=%2d\n", day);

     for (hour=5; hour>2; hour--)
     {
         minutes = 60 * hour;
         printf("Hour = %2d, Minutes=%3d\n",hour, minutes);
     }
}
```

Initialisation.

Test expression.

"Increment" expression.

Body of for loop.

Output

```
Day= 1
Day= 2
Day= 3
Hour = 5, Minutes=300
Hour = 4, Minutes=240
Hour = 3, Minutes=180
```

Explanation

1. **What is a for loop?** A *for loop* is another iterative control structure. For example, the statements

```
for (day=1; day<=3; day++)
        printf("Day=%2d\n", day);
```

cause the `printf()` function to display the value of `day` three times; that is, from `day` equals 1 to `day` equals 3. The for loop takes one of the following forms:

```
for (loop_expressions)
     single statement for_loop body;
```

or

Explanation of code clearly presented in question and answer format

Output

```
Before increment, i= 1, j= 1
After increment,  i= 2, j= 2,
                  k= 1, h= 2
m= 1, p=1.0
n= 1, q=1.5

Original k1=10, k2=20, k3=30, k4=40, k5=50
New      k1=12, k2=18, k3=60, k4=20, k5= 0

a= 7, b= 6, c= 5
d=4.0, e=3.0

x= a + b -c /d *e  =  9.250
y= a +(b -c) /d *e =  7.750
z=((a + b)-c /d)*e = 35.250
```

Explanation

1. **How do we initialise variables?** There are two ways to initialise variables.
   - Method 1: Use an assignment statement to initialise a variable, for example,

     ```
     e=3;
     ```

   - Method 2: Initialise a variable in a declaration statement, for example,

     ```
     float a=7, b=6;
     ```

2. **Assuming that *int* variables *i* and *j* are equal to 1, is the meaning of *k* = `i++;` the same as `h = ++j`?** No. In the first statement, the value of `i` is first assigned to the variable `k`. After the assignment, the variable *i* is incremented by the *post-increment* operator ++ from 1 to 2. Therefore, after executing

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

   `i` = 2 and `k` = 1. However, for `h = ++j`, the value of `j` is first incremented by the *pre-increment* operator ++ from 1 to 2. After the increment, the new `j` value, which now is equal to 2, is assigned to the variable `h`. Therefore, after executing

   ```
   h=++j;
   ```

   j = 2 and *h* = 2. In other words, the statement

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


Explanation

1. **What is the effect of the loop expression `i+=2`?** In this lesson's outer for loop, it is an increment expression that increases the value of `i` by 2 for each loop.

   You will also find that not all of your loops involve addition as the increment expression. For instance, an equally valid expression is `i*=2`. What is used depends entirely on the problem being solved.

2. **What is a nested for loop?** A nested for loop has at least one loop within a loop. Each loop is like a layer and has its own counter variable, its own loop expression and its own loop body. In a nested loop, for each value of the outermost counter variable, the complete inner loop will be executed once. This means that the inner loop will be executed more frequently than the outer loop. The example in this lesson has two counter variables, `i` and `j`, where `i` is the outer loop counter and `j` is the inner loop counter. The outer loop is executed three times, when `i` = 1, 3 and 5. For each `i` value, the `j` loop is executed four times. Since the `j` values in each `j` loop can be 1, 2, 3 and 4, the total number of times that the inner loop is executed is 3 * 4 or 12 times. A conceptual illustration of the nested for loop for this lesson's program is shown in Fig. 4.12. Observe from this figure how the value of `j`

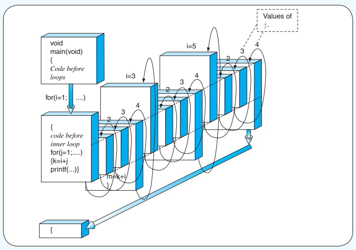

Fig. 4.12 *Nested for loop for this lesson's program. The unlabelled numbers are the values of `j`. For simplicity, the test expressions and their proper locations are not shown.*

Application Programs illustrating the usefulness of the C language for solving engineering and computer science problems

3. Modify the program so that it can handle four pair of lines and, therefore, find four intersection points. You should put in a loop to do this.
4. Modify the program so that it can handle a variable (n) number of lines. The input data file would be

```
n
m1 x1
m2 x2
m3 x3
.
.
.
mn xn
```

Application Program 4.2: Area Calculation – For Loop

Problem Statement

Calculates the areas of four different right triangles. As illustrated, for loops can be used to repeat execution of C statements. As such, they can be used in programs that perform the same tasks as those done by many of the application programs in Chapter 3.

This application program uses a for loop to perform the same task as Application Program 3.1, which calculates the areas of four different right triangles as an example of using patterns to write programming statements.

Look at this portion of the program development in Application Program 3.1 and then read Application Program 4.2 and follow the steps through the for loop.

Compare both application programs. Closely follow the flow of Application Program 4.2, statement by statement. Use your calculator and the loop in the program to fill in this table.

i	area	horizleg	vertleg

Pay particular attention to the way the variables horizleg and vertleg are used. You can see that they are initialised before the for loop. Once in the for loop, the area is first calculated and then printed. Then new values of horizleg

Modification Exercises

1. Modify the program to perform the following tasks:
   a. Create a table of values that go from `degC=0` to `degC=100` in increments of 1 degree.
   b. Create a table with `degF` in the left column and `degF` incrementing by 5 degrees from 250 to 1300.

Application Program 4.4: Temperature Unit Conversions – Loop and If-Else Control Structure

Problem Statement

Write a program that converts an input temperature from degrees Celsius to degrees Fahrenheit and vice versa. The program will terminate when a negative degree is inputted.

Solution

Relevant Equations

The equation developed in Application Program 4.3 is

```
F = C*(9/5) + 32
```

where F is degrees Fahrenheit and C is degrees Celsius. In addition, our program wants to convert degrees Fahrenheit into degrees Celsius. The corresponding equation is

```
C = (F-32) *5/9
```

Algorithm

According to the problem statement, a loop is needed to process each input degree. Besides, the program needs to be able to distinguish which conversion to perform. The algorithm becomes

```
Read user input
Loop as long as input is non-negative
    If input is in degree Celsius
    Calculate the corresponding value in degree Fahrenheit
    Else if input is in degree Fahrenheit
    Calculate the corresponding value in degree Celsius
```

The source code follows this algorithm step by step. Read the program to see how it is done. Once again, follow through the while loop carefully to understand

Modification and Application Exercises for further reinforcement and practice

Comments

In this program, we defined the maximum number of points as 100 and read in the actual number of data points as the first item in the data file. Should we want to analyse more than 100 points, we would need to change this value.

We would like to comment here about developing efficient code. Because we are very concerned in developing efficient code, we are concerned in assessing the efficiency of our algorithms. Part of assessing the efficiency of an algorithm that involves comparisons is evaluating how many comparisons are made in executing the algorithm. Determining the number of comparisons is not necessarily straightforward, different situations require different numbers of comparisons to be made. For instance, for our algorithm to evaluate the median of a list of n numbers, we see that if the median is the first value in our list (just by chance) we will make only n comparisons (because just one pass through the list gives us the median).

However, should the median be the last value in the list (again by chance) we would make n comparisons for each of the n values; that is, n^2 comparisons to perform a median evaluation. If we had 1000 values in our list, this would mean we would make $1000^2 = 1$ million comparisons. You can see that, for this particular algorithm, the number of comparisons can be quite great. Therefore, developing a more efficient algorithm may be quite beneficial. We will not develop one here; however, we want to make you aware that a part of engineering and computer science involves the search for efficient algorithms. You may very well take courses later in your educational career that focus on algorithm development.

Modification Exercises

1. Replace the do-while loop with a while loop that needs no break statement.
2. Make x[ ] an array of doubles rather than integers.
3. Modify the program to handle 12 lists of wave height data (one for each month in a year) in the input file. The input data file would be as follows:

$n1$
$h_1\ h_2\ h_3 \ldots h_{n1}$
$n2$
$h_1\ h_2\ h_3 \ldots h_{n3}$
.
.
.
.
$n12$
$h_1\ h_2 \ldots . h_{n12}$


```
                printf ("%1d", a);
            a = !a;
            sum += count;
        } while (sum<SIZE2);
        printf ("\n");
    }
    fclose (outfile);
}
```

Comment

We have indeed created a file that is considerably smaller than the original file, and it can be used to recreate the original file should we want to do so.

Modification Exercises

1. Modify the program and input file to handle bitmaps of size 20 by 40. Is it easy to do?
2. Create a modular design for this program. Make four functions – one for reading the input file, one for printing the input file, one for compressing the file and one for expanding the file.

Application Exercises

6.1. The number of million gallons of sewage that are disposed of each day for a major city is measured continuously for about a month. The records saved in a file, EX6_1.DAT, are as follows:

```
123, 134, 122, 128, 116, 96, 83, 144, 143, 156, 128, 138,
121, 129, 117, 96, 87, 148, 149, 151, 129, 138, 127, 126,
115, 94, 83, 142
```

Write a program to calculate the frequency distribution using an interval of 10 million gallons per day. The input specification is to use the array sewage_amt[100] to read the number of millions of gallons from the file EX6_1.DAT. The output specification is to display the following data on the screen:

Day no.	Millions of gallons
1	123
2	134
3	122
...	...