Chapter 3

Basics in C

Lesson 3_1 - Data and Variables (1)

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 learned 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 given information is:

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

In order to write an algebraic equation to determine the area you need to make up your own variable names. You might choose as variable names:

1. Lengths - a
- b
- c
2. Angles - α
- β
- γ

Or you could name the variables:

1. Lengths - l_1
- l_2
- l_3
2. Angles - θ_1
- θ_2
- θ_3

Or you could name the variables something completely different. It is entirely up to you what to name them and you most likely would choose variable names that for some reason are most comfortable to you.

For programming in C, the situation is quite similar. You choose the variable names, and it is best for you to choose names with which you are most comfortable. A major difference between typical C programs and typical algebraic expressions is that the variables in most algebraic expressions consist of just one or two characters with maybe a subscript or superscript. Variables in C programs often consist of entire words rather

than single characters. Why? Because as you will find, programs can get to be quite long and there simply are not enough single characters to represent all of the necessary variables. Also, you will find that it will be easier to understand your own programs when you pick them up after a few weeks of leaving them idle if you have given very descriptive names to each variable.

For instance, for the triangle area program you may use the variable names:

```
1. Lengths     - length1
          - length2
          - length3
2. Angle  - angle1
          - angle2
          - angle3
```

Or if you wanted to be even more descriptive, you could name your variables:

```
1. Lengths     - side_length1
          - side_length2
          - side_length3
2. Angles - angle_opposite_length1
          - angle_opposite_length2
          - angle_opposite_length3
```

These variable names are much less ambiguous than their algebraic counterparts. Unfortunately, expressions using these variable names look much more cumbersome than the ones using simple algebraic notation. However, this is a disadvantage with which we simply must live.

In C, there are rules that you must follow in choosing your variable names. For instance, for many compilers, you are not allowed to use more than 31 characters for one variable name. This and other rules will be discussed in this lesson.

In addition, you must "declare" all of your variable names near the beginning of your program. "Declaring" means to essentially list all of your variable names and indicate what types of variables they are. For instance, variables can be of the integer or real (float) type (or other types which will be discussed later).

Look at the example program in this lesson and see if you can determine which are the variables and of what type they are. Also, see how the printf statements can be used to display the values of the variables. (Hint: The % sign is a key character used in printf statements for displaying the values of variables.)

To make you output look neat, it is necessary that the variables occupy the correct number of spaces. Look very closely at the number of spaces occupied by the variables in the output. Can you relate the number of spaces occupied to the way the variables are printed using the printf statement? Can you see that using the correct number of spaces in the printf statement creates a more professional looking output. Also included in this

program are several assignment statements. An assignment statement gives (assigns) a value to a variable. The variable retains that value until it is changed by another assignment statement. See if you can determine which are the assignment statements in this program.

Source Code

```
main()
{int    month;
 float  expense, income;

 month = 12;
 expense = 111.1;
 income = 100.;

 printf("For the %5dth month of the year\n"
        "the expenses were $%9.2f \n"
        "and the income was $%9.3f\n\n",
         month,expense,income);

 month = 11;
 expense = 82;

 printf("For the %2dth month of the year\n"
        "the expenses were $%5.2f \n"
        "and the income was $%6.2f\n\n",
         month,expense,income);
}
```

Output

```
For the 12 month of the year
the expenses were $111.10
and the income was $100.00

For the 11th month of the year
the expenses were $82.10
and the income was $100.00
```

Explanation

1. *How do we declare variables?*

- Variable names in C must be declared. The statement

  ```
  int month;
  ```

  declares the variable month to be of the int type (which means integer and must be typed in lower-case). An int type data does not contain a decimal point.

2. *How do we declare more than one variable?*

- Variables of the same type may be declared in a statement. However, each of them must be separated from the other by a comma, e.g., the statement

  ```
  float expense, income;
  ```

  declares the variables expense and income to be of the float (which must be typed in lower case) type. A float type data contains a decimal point with or without a fraction. For example, 1., 1.0, and 0.6 are float type data. When data without a decimal point is assigned to a float type variable, the C compiler will automatically place a decimal point after the last digit.

3. How do we name variables?

- Variables in C programs are identified by name. The naming convention must obey the following rules:

Component	Requirement
The 1st character in name	Alphabetic characters a-z, A-Z, $, and _.
The other characters in name	Any alphanumeric character, i.e., a-z, A-Z, $, _, and 0-9.
The maximum number of characters in a name	Depends on your compiler and the method of compiling, however for most compilers, the maximum number is 31 characters.

Use of C reserved words, also called keywords, in name	Not allowed, i.e., do not use float, int, ..., etc. A complete list of reserved word is:
	
	`auto` `break` `case`
	`char` `const`
	`continue`
	`default` `do` `double`
	`else` `enum` `extern`
	`far` `float` `for`
	`goto` `if` `int`
	`long` `register` `return`
	`short` `signed` `sizeof`
	`static` `struct` `switch`
	`typedef` `union`
	`unsigned`
	`void` `volatile` `while`

Use of standard identifies such as printf	Allowed, however it is not recommended that standard identifiers be used as variable names because it is very confusing.
Use of . , + - * / % &		Not allowed since they are not alphanumeric.
Use of uppercase characters or	Allowed. However, many programmers use

mixed-case characters	lower-case characters for variable names and uppercase for constant names.
Use of blank within name	Not allowed.

- Examples of illegal variable names

```
1apple, pear%, float, In come, In.come, while, union
```

4. *What is an assignment statement?*

- An assignment statement assigns a value to a variable. For example, the statement

```
month = 12;
```

assigns the integer value 12 to int type variable month. In general, a C assignment statement takes the form of

```
Variable_name = Value;
```

where the statement assigns the Value on the right side of the equal sign to the variable on the left side of the equal sign. The Value can be a constant, a variable with a known value, or other, such as a function or an expression which return a value (see the next few lessons for more details).

5. *How do we display the value of a variable or constant on the screen?*

- The printf() function can be used to display the value of a variable or constant on the screen. The syntax is:

```
printf(format string, argument list);
```

where the format string contains two types of objects. The first one is the plain characters (optional) which will be displayed directly to the screen and the second one is the conversion specification(s) which will be used to convert, format, and display argument(s) from the argument list. Each argument must have a format specification. Otherwise, the results will be unpredictable. For example, in the statement

```
printf("month=%5d,month);
```

The format string is “month=%5d”. The plain characters month = will be displayed directly without any modification, but the conversion specification %5d will be used to convert, format, and display the argument month on the screen.

- The simplest printf() conversion specifications (also called format specifications) for displaying int and float type values have the following forms:

```
%[field width]d                    e.g., %5d for int
%[field width][.precision]f   e.g., %9.2f for float
```

where format string components enclosed by [] are optional. (The characters [and] are not the part of the format string.) The field width is an integer representing the minimum number of character spaces reserved to display the entire argument (including the decimal point, digits before and after the decimal point, and sign). The precision is an integer representing the maximum number of digits after the decimal point. For example %5d will reserve 5 blank spaces for displaying and int type data, %9.2f will reserve a total of 9 blank spaces for a float type data, and 2 digits will be displayed after the decimal point. If your actual input data contains fewer digits after the decimal point, the C compiler will add additional zero(s) after the decimal point when displaying it. For example, for the statements

```
expense=111.1;
printf("the expenses were $%9.2f\n",expense);
```

the C compiler will add one zero to make the precision equal to two to give 111.10 for the output of expense.

Exercises

1. True or False:

   a. The following int type variable names are legal: 1CAT, 2DOGS, 3PEARS.
   b. The following float variable names are legal: CAT, DOGS2, PEARS3, main, printf.
   c. 5d or %8D are legal format specifiers for an int type variable or constant.
   d. 6.3f or %10.1F are legal format specifiers for a float type variable.
   e. The two statements below are identical:
      int ABC, DEF;
      int abc, def;.

2. The price of an apple is 50 cents, a pear 35 cents, and a melon 2 dollars. Write a program to display the prices as follows:

```
*****        ON SALE        *****
Fruit type                  Price
Apple                       $ 0.5
Pear                        $ 0.35
Melon                       $ 2
```

Solutions

Chapter 3

1.

A	B	C	D	E
F	T	F	F	F

Lesson 3_2 - Data and Variables (2)

Topics:
- More about format specifiers and their components
- Scientific notation
- Using the define directive to define constants
- Displaying constant values

Not only may you be interested in controlling the number of spaces that your displayed values will occupy, but you may also be interested in displaying the values either left or right justified (meaning that the value is at the leftmost or rightmost edge of the region allocated to the variable). Look at the program for this lesson. **Can you determine what format specifiers can be used to left justify the variable output?**

When working with very large or very small numbers, scientific notation is convenient. For example, to represent:

57,650,000

the scientific notation would be:

5.765×10^7

which the C compiler would display as:

5.765e + 007 or 5.765E + 007

By using scientific notation, the C compiler decides the value of the exponent, and thus it is possible to display an extremely large number in a small number of spaces. The programmer need only decide on the number of significant digits to display. When you look at the program for this lesson, see how scientific notation is specified. Can you determine how to display the correct number of significant digits?

You will also find that there will be times when you will need to use values which do not change. For instance, we know that PI is approximately 3.1416. For a program which involves areas of circles, it is convenient to simply write PI in the equations. This can be done by defining PI as a constant at the beginning of a program. Look at this program to see how constants are defined. Notice where the define directive appears in the program. Is there a semicolon at the end of the define directive?

Source Code

```
#define DAYS_IN_YEAR 365
#define PI 3.1416
main()
{float income;

 printf(“Days in year=\n”
 “%+5d \n%-5d \n%1d \n% d \n%d \n%0.5d \n\n”,

 DAYS_IN_YEAR, DAYS_IN_YEAR, DAYS_IN_YEAR,
 DAYS_IN_YEAR, DAYS_IN_YEAR, DAYS_IN_YEAR);

 printf(“PI=\n”
 %+9.5f \n%-9.5f \n%1.3f \n%f\n\n”,
 PI,       PI,      PI,     PI);

 income=1234567890.12;

 printf(“income=\n”
 “%15.4e \n%-15.4e \n%5.2e \n%e     \n%E,n,n”,
 income,  income,   income,   income, income):
}
```

Output

```
Days in year=
b+365
365bb
365bb
bb365
365bb
00365

PI=
b+3.14160
3.14160bb
3.142
3.141600

income=
bbbb1.2346e+009
1.2346e+009bbbb
1.23e+009
1.234568e+009
1.234568E+009
```

Note:

b represents blank.

The b’s do not appear in the actual output

Explanation

1. *How do we define a constant?*

- We use a preprocessor directive to define a constant. The preprocessor is a system program which is part of the C compiler. It performs various operations prior to the translation of source code into object code. In C, preprocessing directive begin with the symbol # (which must begin the line). A semicolon must not be used at the end of the preprocessing directive. Only the preprocessing directive should be on the line. For example, the line

  ```
  #define DAYS_IN_YEAR   365
  ```

  defines DAYS_IN_YEAR as a constant which has a value of 365.

- The structure of a define directive is

  ```
  #define Symbolic_name Value
  ```

  where the Symbolic_name represents the constant name and Value is the value assigned to the constant. Note that constants can only be defined one at a time and their values cannot be altered later in the program using an assignment or other statements. the preprocessor will replace any Symbolic_name in the program with the given value. For example, this symbolic name DAYS_IN_YEAR in the statement

  ```
  printf("Days in year=%5d\n",DAYS_IN_YEAR);
  ```

  will be replaced by 365 at the very beginning of compilation by the preprocessor. The statement above will be "re-written" to be:

  ```
  printf("Days in year=%5d\n",365);
  ```

  The symbolic_name in a define directive is known as a constant macro. For this example, the constant macro (DAYS_IN_YEAR) is replaced with the value 365 throughout the program at the very beginning of compilation.

2. *How do we name constants?*

- Constants in C programs are identified by name. The naming convention for constants is the same as the one for variables. Many C programmers use uppercase characters to name constants and use lower-case characters to name variables. This book follows this approach. Constant and variable names are also called user-defined identifiers. These contrast with standard identifiers which have special meaning such as printf or scanf.

3. What is the complete structure of format specifiers?

- The complete structure of format specifiers is:

  `%[flag][fieldwidth][.precision]type`

  where format string components enclosed by [] are optional . (The characters [and] are not the part of the format string.) The meanings of these components may vary slightly from compiler to compiler. The Microsoft C 6.0 Compiler defines the meaning and usage of each component as shown in the table below. (If you do not use the Microsoft C 6.0 Compiler, you should check the manual of the compiler you use.)

Component	Usage
flag=-	Left-justifies the output with the given fieldwidth.
flag=+	Right-justifies the output. Displays plus sign if result is positive.
flag=zero	Adds leading zeros to reach minimum fieldwidth.
flag=blank	Right-justifies the output. If result is positive, the output begins with a blank. If result is negative, the output begins with a minus.
fieldwidth	It is an integer which represents the minimum number of character spaces reserved to display the entire output (including the decimal point, digits before and after the decimal point, and sign). If the specified fieldwidth is not given or is less than the actual field width, all characters of the value, as long as they are within the limit of precision specification, will be displayed, i.e., the fieldwidth specification never "truncate" the output value.
precision	For floating data type, precision specifies the number of digits after the decimal point. The default precision for float type data is 6. Precision can also be used for integer type data. Here, the precision specifies he minimum number of digits to be displayed. If the data to be displayed has fewer digits than the specified precision, the compiler will add leading zero of the output.
type=d	For int type data.
type=f	The output is converted to decimal notation in the format of [sign]ddd.ddd..., where the number of digits after the decimal point is equal to the specified precision.
type=e or E	The output is converted to scientific notation in the format of [sign]d.ddd...e[sign]ddd, where the number of digits before the decimal point is one; the number of digits after the decimal point is equal to the specified precision; the number of exponent

	digits is 3.

- The table below shows the meaning of using different formats for displaying an int constant DAYS_IN_YEAR=365, a float type constant PI=3.1416, and a float type constant income = 1234567890.12 (note: the letter b in the Display column indicates that a blank is displayed).

Format	Flag	Field width	Type	Precision	Display	Note
%+5d	+	5	d	none	b+365	Right adjusted output, + sign added, total characters displayed=5.
%-5d	-	5	d	none	365bb	Flag is -, so output is left adjusted.
%1d	none	1	d	none	365	Specified fieldwidth is less than the actual width, all characters in the value are displayed, no truncation occurs.
% d	blank	none	d	none	bb365	Flag is blank, so output is prefixed with blank, default fieldwidth for int is 5.
%0.5d	zero	0	d	5	00365	Flag is zero, so output is prefixed with zeros, precision is 5, so the number of characters to be printed is 5.
%d	none	none	d	none	365	Fieldwidth is undefined, all characters in the value are displayed, no truncation occurs. No blanks are added. Value is left justified.
%+9.5f	+	9	f	5	b+3.141 60	Total digits, including blanks, is 9.
%-9.5f	-	9	f	5	3.14160 bb	Flag=-, left adjusted output.
%1.3f	none	1	f	3	3.142	Use precision 3, note the result is 3.142, not 3.141.
%f	none	none	f	none	3.141600	Use default precision , 6.
%+15.4e	+	15	e	4	bbbb1.2 346e+0 09	Flag=+, right adjusted output, total digits is 15, precision is 4.
%-15.4e	-	15	e	4	1.2346e +009bb bb	Same as above, but Flag=-, so output is left adjusted.
%5.2e	none	5	e	2	1.23e+0 09	Precision is 2. Fieldwidth is too short, so C uses minimum fieldwidth for output.
%E	none	none	E	none	1.23456 8E+009	Precision is undefined, so C uses default precision of 6. Fieldwidth is too short, so C uses minimum fieldwidth for output.

- Note that if the specified fieldwidth is not given or is less than the actual field width, all characters of the value, as long as they are within the limit of precision specification, will be displayed, i.e., the fieldwidth specification never "truncates" the output value.

4. *How does the Microsoft C compiler convert a float number to a scientific notation?*

- It converts a float number to scientific notation using the format (d or D represents a digit):

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

  where the number of digits before the decimal point is one; the number of digits after the decimal point is equal to the specified precision; the number of exponent digits is 3. Note that a number in this form is equivalent to

  [sign] d.ddd...*10$^{[sign]ddd}$

  For example, when we use the format %15.4e, i.e., fieldwidth=15 and precision=4 to convert the number

  ```
  123456789.12
  ```

  to its scientific notation which is

  ```
  bbbb1.2346e+009
  ```

  where b represents blank and the number is equivalent to

  $1.2346*10^{9}$ or
  1234600000.0

  We lose some accuracy after the conversion because the specified precision is not high enough.

5. *Given the same value and using the same format, will programs created using different compilers display exactly the same output?*

- No. In general, given the same value and using the same format, the output displayed by programs created with different compilers may be slightly different.

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 specifier for an int type data should not contain a decimal point and precision, e.g., %8.2d is illegal.

2. Find error(s), in the statements below:

```
a. #DEFINE Pi 3.1416
b. #define Pi 3.1416;
c. #define PI=3.14; AccuratePI=3.1416;
d. printf("%f",123.4567);
e. printf(%d %d %f %f",1,2,3.3.4.4");
```

3. Write a program to display the following output:

```
12345678901234567890123456789012345

income      expense       Name
+111.1      -999.99       Tom
+222.2      -999.88       Dennis
+333.3      -777.77       Jerry
```

4. Use four different flags but the same fieldwidth and precision, four different fieldwidths but the same flag and precision, and four different precisions but the same flag and fieldwidth (i.e., a total of 12 format specifiers) to display an int type variable A and a float type variable B, where A=12345 and B=9876.54321.

Solutions

1.
```
A B C D E
F F F F F
```

2.
```
a. #define Pi 3.1416
b. #define Pi 3.1416
c. #define PI=3.14
   #define AccuratePI 3.1416
d. no error
e. no error
```

Lesson 3_3 - Arithmetic Statements (1)

Topics:
- Operands
- Arithmetic operators and their properties
- Arithmetic expressions

Arithmetic expressions in c look much like the arithmetic expressions you used in algebra. The first section of the example program for this lesson shows some of the operations that can be performed in C arithmetic expressions. Look at this section of the program and see how addition, subtraction, multiplication and division are performed.

Note that is this section of the program are the statements:

i = i + 1

and

j = j + 1

Clearly, these two statements would not make sense if you were to use them in a math class. However, in C, not only do these statements (and statements of this type) make sense, they are actually used quite commonly in programs. **What do they mean?**

To answer this question you must recall that an assignment statement does. An assignment statement assigns the value of the expression on the right side of the equal sign to the variable which is located on the left side of the equal sign. Keep this in mind as you look at the output from the first printf statement and determine what the statements i=i+1 and
j=j+1 do.

In the second section of this program are expressions with operators whose functions are not quite so obvious. The % sign is especially tricky. See if you can figure out what it does. (Hint: it has something to do with division.)

Also the ++ and -- are operators, but there are no equal signs in the statements for these. They do, though, have an impact on the values of the variables either preceding or succeeding them. **What effect do they have on these variables?**

Source Code

```
main()
{int i,j,k,l,m,n;
 float a,b,c,d,e,f,g,h,x,y;

 i=5;         j=5;
```

```
      k=11;  l=3;
      x=3.0; y=4.0;

      printf("...... Initial values ......\n");
       printf("i=%4d, j=%4d\nk=%4d, l=%d\nx=%4.2f, y=%4.2f\n\n",
               i,j,k,l,x,y);

/*--------------- Section 1 ---------------*/
      a=x+y;
      b=x-y;
      c=x*y;
      d=x/y;
      e=d+3.0;
      f=d+3;
      i=i+1;
      j=j+1;

      printf("...... Section 1 output ......\n");
      printf("a=%5.2f, b=%5.2f\nc=%5.2f, d=%5.2f\n"
      "e=%5.2f, f=%5.2f\ni=%5d, j=%5d \n\n",
      a,b, c,d, e,f, i,j);

/*--------------- Section 2 ---------------*/
      m=k%l;
      n=l%k;
      i++;
      ++j;
      e--;
      --f;

      printf("...... Section 2 output ......\n");
      printf("m=%4d, n=%4d\ni=%4d, j=%4d\n"
      "e=%4.2f, f=%4.2f\n",m,n, i,j, e,f);
     }
```

Output

```
......  Initial values ......

 i=   5, j= 5
 k=  11, l= 3
 x=3.00, y=4.00

...... Section 1 output ......

 a= 7.00, b=-1.00
 c=12.00, d= 0.75
 e= 3.75, f= 3.75
 i=    6, j=    6

...... Section 2 output ......

 m=   2, n=   3
 i=   7, j=   7
```

```
e=2.75, f=2.75
```

Explanation

1. *What is an arithmetic expression?*

   - An arithmetic is a formula for computing a value. For example, the expression x+y computes x plus y.

2. *What are the components of an arithmetic expression?*

   - An arithmetic expression consists of operand(s) and operator(s). For example, the expression *-x+y consists of two operands x and y and two operators + and -.*

3. *What can be an operand?*

   - An operand can be a variable, such as x or y, or a constant, such as 3.1416, or anything that represents a value, such as a function (see lesson 3_5 for details).

4. *What are the meanings of the operators ++, -- and %?*

   - ++ is an increment operator which can be placed before or after (but not both) a variable. The operator will increase the value of the variable by one. For example, assuming a variable i is equal to one, then after the statement

     ```
     i++;      or
     ++i;
     ```

     is executed, the value of i will become 2. Note that the C statement

     ```
     i++;      or
     ++i;
     ```

     can be understood as the statement

     ```
     i=i+1;
     ```

     which also causes the value of the variable i to increase by one. Similarly, the operator -- is a decrement operator which decreases the value of a variable by one. Also, the statement

     ```
     i--;      or
     --i;
     ```

     can be understood as the statement

     ```
     i=i-1;
     ```

% is a remainder operator which must be placed between two integer variables or constants. Assuming k and l are two integer variables, the meaning of k%l is the remainder of k divided by l. For example, if k=11 and l=3, then k%l is equivalent to 11%3, which is equal to 2. The operator % is pronounced "mod". So the above example would be k mod l.

5. *Is an arithmetic expression a complete C statement and how are arithmetic expressions used in assignment statements?*

- An arithmetic expression is not a complete C statement. The expression is only a component of a statement. The value evaluated from the expression may be stored in a variable using an assignment statement. For example, the arithmetic expression x/y is part of a C assignment statement

```
d = x/y;
```

The statement assigns the value obtained from the arithmetic expression on the right to the variable on the left. Thus, the assignment statement

```
i=i+1;
```

while not looking correct algebraically, is a valid C assignment statement. The arithmetic expression `i+1` creates a new value which is one greater than `i`. The assignment statement gives `i` this new value.

Exercises

1. True or False:

```
a. a+b is a correct arithmetic expression.
b. is a complete C statement.
c. If a=5, then a is equal to 6 after a++; is executed.
d. is equal to 2 and 3%5 is equal to 3.
e. is equal to 2 and 3.0%5 is equal to 3.0.
f. The meaning of the equal sign, =, in the statement
   a = x+y;.
   is equal, i.e., a is equal to x+y.
```

2. Write a program to calculate your expected average GPA in the current semester and display your output on the screen.

Solutions

1.
```
A B C D E F
T T T T F F
```

Lesson 3_4 - Arithmetic Statements (2)

Topics:
- Precedence of arithmetic operations
- Initializing variables
- Pitfalls in arithmetic statements

Before variables can be used in arithmetic expressions they must first be given numerical values. Giving variables their first numerical values is called initializing them. We will find that there are several different ways to initializing them. **What are the two different ways shown in this program for initializing variables?**

In the following program, the arithmetic expressions 6/4 and 6/4.0 are used twice each. The variables on the left side of the assignment statements using these expressions are either float or integer. Look at the output for these variables. Can you guess how the declared data type (float or int) influences how the C compiler assigns values to the variables? (Hint: an integer must always be an integer. It cannot be assigned a real value.)

Also included in the program are the compound operators +-, -=, *=, /=, and %=. By looking at the output for k1, k2, k3, k4, and k5, you can deduce what these operators do?

In this program are assignment statements using the ++ and -- operators. When trying to determine what these statements do, remember that assignment statements take the value of the EXPRESSION on the right side of the equal sign and give that value to the VARIABLE on the left side of the equal sign. Note that initially, both i and j are equal to 1. **Are the values of the EXPRESSIONS i++ and ++j the same? What does that tell you about how the C compiler defines the VALUES OF THESE TYPES OF EXPRESSIONS?** Also, note what has happened to the values of i and j after execution of these statements.

You have learned in your math classes that parentheses can be used in arithmetic expressions to control the order in which the operations are performed. Similarly, you can use parentheses in your C code to control the order of performance of operations. Also, C has strict rules about the order of operation of addition, subtraction, multiplication and division. These rules are established by setting the “precedence” of the operators.

Operators which have high precedence are executed first while those of lower precedence are executed later. For two operators of equal precedence, the one that is leftmost in the expression is executed first. Use your calculator to calculate the values of X, Y, and Z in the program below. Can you determine which operators are of higher precedence - addition, subtraction, multiplication or division? (Hint: Addition and subtraction have the same precedence and multiplication and division have the same precedence.)

Source Code

```
main()
{int i=1, j=1,]
     k1=10, k2=20, k3=30, k4=40, k5=50,
     k, l, m, n;
 float a=7,  b=6, c=5, d=4
     e, p, q, x, y, z;

 printf("Before increment, i=%2d, j=%2d\n",i,j);

 k=i++;
 l=++j;

 printf("After increment,     i=%2d, j=%2d"
        "                k=%2d, l=%2d \n\n",i,j,k,l);

 m=6/4;
 p=6/4;
 n=6/4.0;
 q=6/4.0;

 printf("m%2d, p=%3.1f\nn=%2d, q=%3.1f\n\n",m, p, n,
 q);
 printf("Original k1=%2d, k2=%2d, k3=%2d, k4=%2 k5=%2d\n"
        k1,k2,k3,k4,k5);

 k1 += 2;
 k2 -= i;
 k3 *= (8/4);
 k4 /= 2.0;
 k5 %= 2;

 printf("New   k1=%2d, k2=%2d, k3=%2d, k4=%2d, k5=%2d\n\n",
        k1,k2,k3,k4,k5);

 e=  3;
 x=  a + b -c  /d *e;
 y=  a +(b -c) /d *e;
 z=((a + b)-c  /d)*e;

 printf("a=%3.0f, b=%3.0f, c=%3.0f\nd=%3.1f, e=%3.1f\n\n",
        a,b,c,d,e);

 printf("x=  a + b -c  /d *e = %10.3f \n"
        "y=  a +(b -c) /d *e = %10.3f \n;
        "z=((a + b)-c  /d)*e = %10.3f\n", x,y,z);
}
```

Output

```
Before increment,   i= 1, j= 1
After increment,    i= 2, j= 2,
                    k= 1, l= 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 initialize variables?*

- **Method 1**: use an assignment statement to initialize a variable, e.g.,

```
e=3;
```

- **Method 2**: initialize a variable in a declaration statement, e.g.,

```
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 l = ++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 one to two. Therefore, after execution the first statement, i=1 and k=2. However, in the second statement, the value of j is fist incremented by the pre-increment operator ++ from one to two. After the increment, the new j value, which is equal to two now, will be assigned to the variable l. Therefore, after executing the second statement, j=2 and l=2.

  In other words, the statements

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

  is “equivalent to” statements

```
k=i;
```

```
i=i+1;
```

However, the statement

```
l=++j;
```

is “equivalent to” statements

```
j=j+1;
l=j;
```

- In other words, even though both expressions (i++ and ++i) cause the value of i to be increased by one, the value of the expression i++ is equal to the value of i prior to the increase while the value of the expression ++i is equal to the value of i after the increase. For example, for the statements:

```
h=7;
i=7;
j=i++;
k=++h;
```

The value of j is 7 and the value of k is 8 because j and k are assigned the values of the expressions i++ and ++j, respectively. Note that after executing the above four statements, the values of h and i are 8.

3. *What is the value of 6/4 or 6/4.0 or 6.0/4?*

- When one integer is divided by another integer, the fraction part of the quotient is discarded. Therefore, 6/4 is equal to 1. If the result is assigned to an int type variable m, the value of m will be 1; If the result is assigned to a float type variable p, the value will be 1.0. Clearly, this may not be the result that you would like to get for p. Read further to see how you can get different result for a similar calculation.

- In an arithmetic expression, if one operand is of int type and the other is of float type, the int type operand will be converted first to the float type before the expression is evaluated. Therefore, for the expression 6/4.0 or 6.0/4 both operands will be converted to 6.0/4.0 which is equal to 1.5. If the result is assigned to an int type variable n, the value of n will be 1 (the fraction part is discarded). If the result is assigned to a float type variable, the value will be 1.5.

- There is a very important lesson in this. When you are writing your code and a float type variable is on the left side of an assignment statement, to be safe, you should use decimal points for any numbers on the right side of the assignment statement. You may get the correct results without using decimal points, but we recommend that you use decimal points until you feel comfortable with mixed

variable type arithmetic (you may also convert an integer number to a number with a decimal point and vice versa, see Chapter 8 for details).

Also, when an int type variable is on the left side of an assignment statement, it is necessary that you make sure that the arithmetic expression on the right side of the assignment statement is an integer value or that you are deliberately dropping the fractional part.

4. *What are the meanings of operators +=, -=, *=, /=, and %=?*

- The operators +=, -=, *=, /= and %= are compound assignment operators. Each of them performs an arithmetic operation and an assignment operation. These operators require two operands, the left operand must be a variable, the right one can be a constant, a variable, or an arithmetic expression. In general, the two operands can be of integer or floating data type. However, the %/ operator requires that its two operands must be of integer type.

- The meaning of

  ```
  k1+=2;
  ```
  `(not k1 =+ 2;)` can be understood to be similar to the statement

  ```
  k1=k1+2;
  ```

  If the original value of k1 is equal to 20, the new value will be 20+2 or 22. Similarly, the statements above are also valid if we replace the arithmetic operator + with operators -, *, /, or %. For example,

  ```
  k1*=2;
  ```

  is similar to

  ```
  k1=k1*2;
  ```

5. *How do we control precedence in an arithmetic expression?*

- Parentheses can be used to control precedence. Any arithmetic operators located within the parentheses always have higher precedence than any outside the parentheses. When an arithmetic expression contains more than one pair of parentheses, the operators located in the innermost pair of parentheses have the highest precedence. For example, the + operator in the statement

  ```
  z = ((a+b)-c/d);
  ```

  has higher precedence than the - or / operator and `a+b` will be evaluated first.

6. *What will happen if all operators have the same level of precedence?*

- If all arithmetic operators are of equal precedence in an arithmetic expression, the leftmost operator is executed first.

7. *Can we use two consecutive arithmetic operators in an expression?*

- We cannot use two consecutive arithmetic operators in an arithmetic statement unless parentheses are used. For example, x/-y is not permissible but x/(-y) is permissible.

8. *What operators can be used in an arithmetic expression?*

- The table below shows the operators along with their properties that can be used in an arithmetic expression:

Arithmetic Operators

Operator	Name	Number of operands	Position	Associatively	Precedence
(	parentheses	unary	prefix	L to R	1
)	parentheses	unary	postfix	L to R	1
+	positive sign	unary	prefix	L to R	2
-	negative sign	unary	prefix	L to R	2
+	post-increment	unary	postfix	L to R	2
--	post-decrement	unary	postfix	L to R	2
+	pre-increment	unary	prefix	R to L	2
--	pre-decrement	unary	prefix	R to L	2
+=	addition & Assignment	binary	infix	R to L	2
-=	subtraction & Assignment	binary	infix	R to L	2
*=	multiplication & Assignment	binary	infix	R to L	2
/=	division & Assignment	binary	infix	R to L	2
%=	remainder & Assignment	binary	infix	R to L	2
%	remainder	binary	infix	L to R	3
*	multiplication	binary	infix	L to R	3

/	division	binary	infix	L to R	3
+	addition	binary	infix	L to R	4
-	subtraction	binary	infix	L to R	4
=	assignment	binary	infix	R to L	5

9. *What are the meaning of number of operands, position, associatively, and precedence in the table above?*

- Number of operands: It is the number of operands required by an operator. A binary operator, such as /, requires two operands while a unary operator, such as ++, needs only one.
- Positions: It is the location of an operator with respect to its operands. For a unary operator, its position is prefix if the operator is placed before its operand and postfix if it is placed after its operand; for a binary operator, the position is infix because it is always placed between its two operands. For example, the negation operator is -x is prefix, the post-increment operator in y++ is postfix, and the remainder operator in a %b is infix.

- Associatively: It specifies the direction of evaluation of the operators with the same precedence. For example, the operators + and - have the same level of precedence and both associate from left to right, so 1+2-3 is evaluated in the order of (1+2) -3 rather than 1+(2-3).

- Precedence: It specifies the order of evaluation of operators with their operands. Operators with higher precedence are evaluated first. For example, the operator * has higher precedence than -, so 1-2*3 s evaluated as 1-(2*3) rather than (1-2)*3. Note that in this example the '-' indicates subtraction and is a binary operator with precedence 4. The '-' can also be used as a negative sign which is a unary operator with precedence 2. For example, -2+3*4 is evaluates as (-2)+(3*4) rather than -(2+3*4)).

10. *What is a side effect?*

- The primary effect of evaluating an expression is arriving at a value for that expression. Anything else that occurs during the evaluation of the expression is considered a side effect.

  For instance, the primary effect of the C statement (assuming i is originally 7)

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

  is that the expression on the right side of the assignment statement is found to have a value of 7. The side effect of the above statement is the value of i is incremented by one (to make i equal to 8). Consider the following C statement:

```
j = (i=4) + (k=3) - (l=2);
```

Its primary effect is to arrive at the value of the expression on the right side of the assignment statement (which is 5). It has three side effects which occur during the evaluation of the expression. They are to:

1. Set i equal to 4
2. Set k equal to 3
3. Set l equal to 2

11. What are the dangers of side effects?

- At times, side effects can be confusing. For the statement

```
k = (k=4) * (j=3);
```

the result of k will be 12 instead of 4. It is best not to use side effects except in their simplest form such as:

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

or

```
i = j = k = 5;
```

Note that because the associatively of the assignment operator is from right to left, multiple assignment statements such as the one above can be written. The order of operation is:

1) k = 5
2) j = k
3) i = j

Also, an expression

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

is evaluated in the order

1) k = 2 + n + 1;
2) j = k;
3) i = j;

because the addition operation has a higher precedence than the assignment operator.

Exercises

Chapter 3

1. Based on the following statements

```
int i=10, j=20, k, m,n;
    float a,b,c,d,e=12.0;
```

Determine whether each of the following statements is true or false:

```
a. i =+ 2; is a valid C statement.
b. i %= e; is a valid C statement.
c. i *= (i+j*e/123.45); is a valid C statement.
d. k=i/j; k is equal to 0.5.
e. i+=j;  i is equal to 30 and j is equal to 20.
f. k=1/3+1/3+1/3; is equal to 1.
g. k=1/3.+1.0/3+1.0/3.0 is equal to 1.
h. a=1/3+1/3+1/3 is equal to 1.0.
i. a=1./3+1/3.+1.0/3.0 is equal to 1.0.
```

2. Hand calculate the values of X, Y, and Z in the program below and then run the program to check your results.

```
main()
{
 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);
}
```

Solutions

1.
```
A B C D E F G H I
F F T F T F F F T
```

Lesson 3_5 - Library Functions (math)

Topics:
- Double data type
- Library Functions
- Using standard header file

Your calculator makes it very easy for you to perform such operations as sin, log, and square root by having single buttons for them. Similarly, the C compiler makes it easy for you to perform these operations by providing mathematical library functions which you can call from your program. This lesson illustrates the use of some of these library functions. Note that just like the printf function, these library functions require parentheses for enclosing the argument(s).

Use your calculator to verify the results obtained in the output for this lesson. What are the necessary units for the argument for the sin function? What kind of log does the log function take?

The C compiler has these functions in its library, but it is necessary for you to tell the compiler more information about them. Can you guess which statement in this program tells the C compiler where the extra information is located?

We have previously discussed float and int type variables. In this lesson, the double type variable is introduced. Compare the output for the variables c and g for this lesson. They are different. Which is more accurate? Use your calculator to check this.

Source Code

```
#include <math.h>
main()
{double x=3.0, y=4.0, a,b,c,d,e,f;
 float g;

 a=sin(x);
 b=exp(x);
 c=log(x);

 d=sqrt(x);
 e=pow(x,y);
 f=sin(y)+exp(y)-log10(y)*sqrt(y)/pow(3.2,4.4);
 g=log(x);

 printf("x=%4.1f  y=%4.1f \n\n\
 a=sin(x)      = %11.4f\n\
 b=exp(x)      = %11.4f\n\
 c=log(x)      = %11.9f\n\n\
 d=sqrt(x)     = %11.4f\n\
 e=pow(x,y)    = %11.4f\n\
```

```
 f=sin(y)+exp(y)log10(y)*sqrt(y)/pow(3.2,4.4) = %11.4f\n\n\
 g=log(x)      = %11.9f\n",x,y, a,b,c,d,e,f,g);
}
```

Output

```
x=  3.0   y= 4.0

a=sin(x)       =      0.1411
b=exp(x)       =     20.0855
c=log(x)       = 1.098612289

d=sqrt(x)      =      1.7321
e=pow(x,y)          =     81.0000
f=sin(y)+exp(y)-log10(y)*sqrt(y)/pow(3.2,4.4)=
     53.8341

g=log(x)       = 1.098612309
```

Explanation

1. *What is the difference between the double and the float data types?*

- The double data type is another floating-point data type in C. Unlike the float data type which is for a single-precision real constant, the double data type is for a double-precision real constant. Essentially, variables which are declared as being data type carry more significant digits with them during calculations than do variables which are declared as float data type. Carrying a large number of digits may be important when a large number of calculations are to be done. The drawback in declaring all variables as being of the double data type is that more memory is required to store double data type variables than float type variables.

  Consider the following example which illustrates the effect of the number of digits carried in a calculation. You should try this on your calculator. Suppose you are multiplying a number by π 100 times. You will essentially be computing π^{100}. The influence on the number of significant digits used for π is the following. Using 5 significant digits for π gives:

  $(3.1416)^{100} = 5.189061599 * 10^{49}$

  while using 8 significant digits for π gives:

  $(3.1415926)^{100} = 5.187839464 * 10^{49}$

  Here, it can be seen that the first estimate of π has five significant digits, however, $(3.1416)^{100}$ is accurate only for the first three digits. This illustrates that accuracy is reduced after numerous arithmetic operations. Since one computer program can easily do one million operations, one can begin to understand the

need for initially carrying many digits. Note that for this lesson's program, it was not necessary to declare the variables to be the double data type. The float data type would have been sufficient. Float and double data types are compared in the table below. Note that the double data-type can store greater values than the float data type:

Item	Float	Double
Required memory	4 bytes	8 bytes
Values	1.17549944E-38 to 3.4028235E+38	2.2250738E-308 to 1.7976935E+308
Precision	6	15
Simplest format	%f, %e, %E	%1f, %e, %E

- Given the low precision of the float data type, we recommend that you use double in your program. We will see that there are numerous other data types in C. Through this text we will introduce the different data types and when it is appropriate to use them.

- Note that the simplest double format is %1f. This format is used extensively throughout this book. It is also acceptable to use %f with the printf function, however, %f is not acceptable for the scanf function which is covered in the next lesson.

2. *What are the meanings of the functions in this lesson?*

- The meaning of these C mathematical library functions are shown below (note that the input argument(s) x or y and the return value of each of these functions are of double type.

Function name	Calculating
sin(x)	the sine of x, x is in radians
exp(x)	the natural exponential of x
log(x)	the natural logarithm of x
sqrt(x)	the square-root of x
pow(x,y)	x raised to the power of y

There are other mathematical C library functions which may take different types of data as input and may return different types of data as output. In general, C library functions may vary slightly from compiler to compiler. You should check your C compiler manual for details). The table below lists a few more math library functions.

Function name	Example	Description
abs(x)	y=abs(x);	Gets the absolute value of an int type argument, x and y are of type int.
fabs(x)	y=fabs(x);	Gets the absolute value of a double type argument, x and y are of type double.
sin(x)	y=sin(x);	Calculates the sine of an angle in radians, x and y are of type double.
sinh(x)	y=sinh(x);	Calculates the hyperbolic sine of x, x and y are of type double.
cos(x)	y=cos(x);	Calculates the cosine of an angle in radians, x and y are of type double.
cosh(x)	y=cosh(x);	Calculates the hyperbolic cosine of x, x and y are of type double.
tan(x)	y=tan(x);	Calculates the tangent of an angle in radians, x and y are of type double.
tanh(x)	y=tanh(x);	Calculates the hyperbolic tangent of x, x and y are of type double.
log(x)	y=log(x);	Evaluates the natural logarithm of x, x and y are of type double.
log10(x)	y=log10(x);	Evaluates the logarithm to the base 10 of x x and y are of type double.

3. *How do we use C mathematical functions?*

- To use C math functions, you need to add the following statement:

  #include <math.h>

  at the beginning of your program. The statement begins with the character #, followed by the lower-case word 'include', and open angle bracket, the filename to be included, and ends with a closed angle bracket. The filename within the coupled angle brackets is a standard header file provided by the C compiler.

4. *What is a header file?*

- A header file is an ASCII file which has an extension '.h'. In general, a header is a collection of information and is referred to at the beginning of a C program. For example, the header file 'math.h' contains constant definitions and C function declarations for the math library. By default, the file is usually located in the '\INCLUDE' sub directory. For Microsoft C Version 6.0, math.h is a 107 line file. For example, the file contains the prototype of the exponential function, pow(x,y), which looks something like

```
double pow(double, double);
```

the first double tells you that the pow() function returns a double type output; the second and the third indicate that the pow() function must have two double parameters as input. By calling the header file at the beginning of a program, we inform the compiler that we may use some of the library functions in the header file. The compiler will select those needed functions and connect it to your program during compilation.

5. *What is a C preprocessor directive?*

- The include statement is, strictly speaking, not a standard C statement because it does not end with a semicolon ';'. In reality, the statement is called a C preprocessor directive because it directs the C compiler to do some pre-processing work before compiling. For example, when the C compiler sees

```
#include <math.h>
```

it first looks for the header file math.h, if the file is found, the compiler will "put" the relevant part of math.h file in the place where the include directive is located. As an experiment, you may delete the include directive from 3_4.C, replace it with the 107 line math.h header file copied from \include\math.h, and then compile the C file. If you do not make any mistakes, the new 3_4.EXE file not only has the same size as the old one, but also generates the same output. However, your new 3_4.C source code is much larger than our old 3_4.C and looks cumbersome to read. By now you believe hat using an include directive is a much better way to deal with an external header file.

6. *What are the advantages of using the include directive?*

- You don't need to write a library function yourself.
- Your program and the library functions are two independent programs.
- Your program is short.
- The header file need only be typed once and can be used by many C programs.
- You have more flexibility and freedom to update you program.

7. *How does the C compiler find a header file?*

There are two methods to guide your C compiler to find the header file.

- **Method 1**: Write the full path of the header file in your source code. For example, if your math.h file is in the C:\MC6\INCLUDE directory, then you could type

```
#include <C:\MC6\INCLUDE\math.h>
```

at the beginning of your program.

- **Method 2**: Type a SET command such as

  SET INCLUDE=C:\MC6\INCLUDE

  before you compile your program. The set command is a DOS command and sets the DOS system environment so that the C compiler can find where the include files are. It can also be part of the AUTOEXEC.BAT file which is executed when your computer boots. Typically, this command is put into the AUTOEXEC.BAT or other BATCH files.

Exercises

1. True or False:

   a. `#include <Math.H>` is a correct C preprocessor directive.
   b. A header file must be placed at the beginning of a C program.
   c. In C, the value of sin(30) is equal to 0.5.
   d. In C, the value of log(100) is equal to 2.0.

2. Find math.h in your C compiler include directory. Then copy and paste it to the program below. Compile, link and run the program. Do you get the same output as the one from C3_4.EXE? Compare the sizes of the source code, object code, and the executable code of the program below and C3_4.C program. Summarize your findings.

```
/* Copy the MATH.H file (without modification)
   below this line */

main()
{float x=3.0,y, z;
 y = 4.0;
 z=pow(x,y);
 printf("x=%4.1f, y=%4.1f, pow(x,y)=%7.2f",x,y,z);
}
```

3. The program below can be compiled and linked without error. But you get an error message when you run it. Why?

```
#include <math.h>
main()
{float X=-111.11, Y=0.5, Z;
 Z=pow(X,Y);
 printf("X=%10.2f, Y=%10.2f, Z=%10.2f\n",X,Y,Z);
}
```

4. Write a program to calculate the unknown values below:

```
Alpha(degree) Alpha(radian) sin(2*Alpha)
   30.0           ?               ?
   45.0           ?               ?
```

Solutions

1.
```
A B C D
T F F T
```

Lesson 3_6 - Input Data From Keyboard

Topics:
- Using the scanf() function
- Inputting data from the keyboard
- The address operator '&'

None of the programs in any of the previous lessons have had input going into them during execution. These programs had only output, and for these the output device was the screen (or monitor). Most commonly, your programs will have both input and output. Your program can instruct the computer to retrieve data from various input devices. Input devices include:

```
1) the keyboard
2) a mouse
3) a joystick
4) the hard disk drive
5) a floppy disk drive
```

to name a few. The program below illustrates how input can be retrieved by a C program from the keyboard. Programs which have input from the keyboard create a dialogue between the program and the user during execution of the program. Examine this program and see how a dialogue can be established between the program and the user. What does the scanf function do in the program below?

Source Code

```
main()
{double income, expense;
 int month, hour, minute;

 printf("What month is it?\n");
 scanf("%d, &month);
 printf("You have entered month=%5d\n",month);

 printf("Please enter your income and expenses\n");
 scanf("%1f %1f",&income,&expense);
 printf("Entered income=%8.21f, expenses=%8.21f\n",
        income,expense);

 printf("Please enter the time, e.g.,12:45\n");
 scanf("%d : %d",&hour,&minute);

 printf("Entered Time = %2d:%2d\n",hour,minute);
}
```

On Screen Dialogue

Program Output	`What month is it?`
Keyboard input	`12`
Program output	`You have entered month = 12`
Program output	`Please enter your income and expenses`
Keyboard input	`32 43`
Program output	`Entered income = 32.00, expenses= 43.00`
Program output	`Please enter the time, e.g., 12:45`
Keyboard input	`12:15`
Program output	`Entered Time = 12:15`

Explanation

1. *How do we input data from the keyboard?*

- An easy way to input data from the keyboard is by using the scanf() function. The syntax of the function is

  scanf(format string, argument list);

  where the format string converts characters in the input into values of a specific type, the argument list contains the variable(s) into which the input data are stored, a comma must be used to separate each argument in the list from the other. For example, the statement

  scanf("%1f%1f,&income,&expense);

  will convert the first input data to double type value using the %1f format specifier and store the double value in the variable income. Similarly, the second input is stored in the variable expense. Note that you must precede each variable name with an & when you read a value. The reason is that the argument in the scanf() function uses a pointer to the variable (which will be discussed in Chapter 7). For now, you don't need to understand the concept of pointers to use the scanf() function. Just remember to add & in front of the variable. If you want to read an int type variable, use %d instead of %1f as the format specifier.

2. *What are the components of a format string in the scanf() function?*

- The format string may consist of format specifiers, such as %d or %1f, blanks, and character(s) to be input. If the format string contains character(s), you must match the character(s) when you input from the keyboard. For example, the statement

scanf("%d : %d",&hour,&minute);

contains a colon "":" in the format string, if you want to input hour=12 and minute=34, the valid input is

12 : 34

If you omit the colon, the data are read incorrectly. In general, the format string in the scanf() function should be kept as simple as possible. Otherwise you will have trouble to correctly input your data.

Exercises

1. Based on the statements

```
int cat, dog;
double weight;

find error(s) in each of the statements below:

a. scanf("%d %d"),cat,dog;
b. scanf(%d %d,cat,dog);
c. scanf("%d %f",cat,dog);
d. scanf("%d %d",&cat,&dog);
e. scanf("%d,\n, %1f",&cat,&weight);
```

2. Write a program to input all your grades in the last semester from the keyboard and then display your input and the average GPA on the screen.

Solutions

1.

```
a.scanf("%d %d",&cat,&dog);
b.scanf("%d %d",&cat,&dog);
c.scanf("%d %d",&cat,%dog);
d.no error.no error, but you have to type in two commas
  when you input the data.
```

Lesson 3_7 - Input Data From File

Topics:
- Opening and closing a file
- Reading data from a file
- Using the fscanf() function

You will find that if your input data is lengthy and you are planning to execute your program many times, it is not convenient to input your data from the keyboard. This is especially true if you want to make only minor changes to the input data each time you execute the program.

For instance, if your income is the same every month and only your expenses change, it is cumbersome to repeatedly type the same number for each month. It is more convenient to set up a file (which can be created using a word processing type program, also called editor) which has your income and expenses in it. Your program can read that file during execution instead of receiving the input from the keyboard. If you want to rerun the program with different input data, you can simply edit the input file first and then execute the program.

This lesson's program illustrates how to read data from an input file. In the program below, the file name is C3_7.IN. You must remember, though, that when you create your input file using your editor that you give that file the same name that you have specified in the code for your program. When you execute your program, the C compiler searches for a file of that name and reads it. If that file does not exist, the C compiler will give you an error message when you execute your program.

Look at the program below. What is the name of the standard function used to read a file?
You also need to open your file before you use it. Can you see which statements are used to open your file? Can you see which statement is used to close your file?

Source Code

```
#include <stdio.h>
main()
{double xx;
 int  ii, kk;
 FILE *inptr;
 inptr=fopen ("C3_7.IN","rt");

 fscanf(inptr,"%d",&ii);
 fscanf(inptr,"%d %lf",&kk,&xx);

 fclose(inptr);
```

```
  printf("ii=%5d\nkk=%5d\nxx=%9.31f\n",ii, kk, xx);
}
```

Input file C3_7.IN

```
36
123  456.78
```

Output

```
ii=  36
kk=  123
xx=  456.780
```

Explanation

1. *What are file and FILE?*

- ♦ A file is a collection of information in an electronic format. It may contain your personnel data, a CIA secret document, or Hollywood's latest video movie. Information in a file is stored in certain section(s) of external device(s), such as tapes or disks. Unlike a number, such as +1234, which can be determined by its size and sign, a file is more complicated and contains more features. For example, a file must have a name so that you or a computer can identify it. A file may be opened for reading, i.e., get data from it, or writing, i.e., store data in it. A file can be in text format, such as the source code of this program, 3_7.C, or binary format, such as the execution code of this program, 3_7.EXE. In addition, a file needs a temporary storage area to declare its size and other information so that it can be placed correctly by the computer operating system. In order to keep all of these features in one place, C "invents" a new data type (in reality it is a data structure, this will be explained in more detail in Chapter 8) named name FILE which is somewhat similar by slightly more complicated than the other data types you have learned, such as int and float.

- ♦ FILE is a C derived data type defined in the C standard header file stdio.h. To include stdio.h, we need to add an include directive

  ```
  #include <stdio.h>
  ```

  at the beginning of the program. Without the include file, stdio.h, the C compiler will not understand what FILE stands for and will generate an error message.

- ♦ When you want to manipulate a file, you use the C data type FILE to declare a special type of variable called file_pointer (see note 3 below), and then use this file_pointer to handle your file. This means that there is no direct relation between the C data type FILE and your actual file, i.e., you cannot use the following statement

  ```
  FILE "3_7.IN";
  ```

to declare your file. Instead, you must use FILE to declare a file_pointer, and then use the file_pointer to manipulate your file. The process is schematically shown below:

```
FILE → file_pointer → actual_file
```

2. *What function is most commonly used to read data from a file?*

- In C, we usually use the fscanf() function to read data from a file. In general, the syntax of the fscanf() function is

  fscanf(file_pointer, format_string, argument_list);

  The fscanf() function reads all the contents in the argument_list using the given format_string from a file which has a file pointer of file_pointer (see note 3 for explanation of file pointer). For example, in the statement

  fscanf(inptr, “%d %1f”,&kk,&xx);

  the values in the argument list, kk and xx, are read using a format string

  “ %d %1f”

  from an external file which has a file pointer called inptr. Note that all input argument names must be preceded with an &.

3. *What is a file pointer?*

- A file pointer is a topic which will be discussed in detail in Chapter 7. For now, just remember that a file pointer is a variable name which must be preceded by n asterisk when you define it and must be defined in a statement that begins with FILE. For example, the statement

  FILE *intptr;

  declares *inptr to be a file pointer. The naming convention for file pointers (except the asterisk character) is the same as the naming convention for other C conventional data types. Examples of legal and illegal file pointer names are shown below:

```
Legal file pointers:     FILE *apple, *IBM93, *HP7475;
Illegal file pointers: FILE *+apple, *93IBM, 75HP75;
```

4. *What function is used to open an input file?*

- Before input can be red from an external file, the file must be opened using the fopen() function whose syntax is:

  file_pointer = fopen (file_name,access_mode);

  For example, in the statement

  inptr = fopen (“C3_6.IN”,”rt”);

  the file_name is C3_6.IN, the file_pointer is named inptr, and the access_mode is “rt”, where ‘r’ mans the file is opened for reading and ‘t’ means the input is read in text mode. You may choose any valid name for file_pointer or file_name to open a file. Note that the file_name and the access_mode are character strings. Hence, they must be enclosed with a pair of double quotes. However, the file pointer is not a character string, so no double quotes surround inptr.

5. *Do we need to close an input file?*

- It is good practice to close files after they have been used. However, if no fclose() statements are used, C will automatically close all open files, after execution is completed. To close a file, use the fclose() function whose syntax is

  fclose(file_pointer);

  Note that, the function uses the file pointer, not the filename, to close a file. For example, the statement

  fclose(inptr);

  uses file pointer intpr to close the file 3_7.in.

Exercises

1. True or False:

   a. We use the scanf() function to read input from the keyboard.
   b. We use the fscanf() function to read input from a file.
   c. You must open an external file before you can read your input data.
   d. It is a good practice to close an input file once you do not need it.
   e. You must define a file pointer before you can open the file.

2. Find error(s), if any, in each statement below.

```
a. #INCLUDE <stdio.h>.
b. file myfile;.
c. *myfile = fopen (C3_6.DAT,rt);.
d. fscanf("myfile","%4d %5d\n",WEEK,YEAR);.
e. close("myfile");.
```

3. Write a program to read your grades from last semester from an input file named "4GRADE.REP" which has one line of data consisting of 4 grades only (no characters), e.g.,

```
4.0 3.3 2.7 3.7
```

compute your average GPA, and write all the input and average GPA on the screen and in a report file named "MYGRADE.REP".

Solutions

1.
```
A B C D E
T T T T T
```

2.
```
a. #include <stdio.h>
b. FILE *myfile;
c. myfile = fopen (“C3_6.DAT”,”rt”);
d. fscanf(myfile,”%4d %5d\n”,&WEEK,&YEAR);
e. fclose(myfile);
```

Lesson 3_8 - Output Data to File

Topics:
- Writing data to a file
- Using the fprintf() function

Previous programs have displayed all of their output to the screen. This may be convenient at times, however once the screen scrolls or clears, the output is lost.

In most cases you will want to have a more permanent record of your output. This can be obtained by writing your output to a file instead of to the screen. Once the output is in a file, you can use a file editor to view it. You can also use the editor to print the result on a printer.

The program below illustrates how to print output to a file. Just as is true for an input file, an output file:

1. Can have any acceptable DOS name.
2. Must be defined before it is used.
3. Must be opened before it is used.
4. Should be closed after it is used.

As you read the program compare and contrast it to the program in Lesson 3_7 which reads data from a file. Do you see any similarities?

Source Code

```
#include <stdio.h>
 main()
 {double income=123.45, expenses=987.65;
 int week=7, year=1996;
 FILE *myfile;

 myfile = fopen(“3_8.OUT”,”wt”);
 fprintf(myfile,”Week%5d\nYear=%5d\n”,week,year);
 fprintf(myfile,”Income =%7.21f\n Expenses=%8.31f,n”,
 income,expenses);
 fclose(myfile);
}
```

Output file 3_8.OUT

```
Week=     7
Year=     1996
Income  = 123.45
Expenses=987.650
```

Explanation

1. *What function do we use to write data to a file?*

   - In C, we use the fprintf() function to write data to a file, In general, the syntax of the fprintf() function is

     fprintf(file_pointer, format_string, argument_list);

     The fprintf() function writes the values of argument_list using the given format_string to a file which has a file pointer of file_pointer. For example, in the statement

     fprintf(myfile," Week = %5d\n Year = %5d\n",week,year);

     the values of argument_list, week and year, are written to an external file which has a file pointer named myfile using format string

     " Week = %5d\n Year = %5d\n"

2. *What function do we use to open an output file?*

   - Before data can be written to an external file, the file must be opened using the fopen() function whose syntax is:

     file_pointer = fopen (file_name,access_mode);

     where the definitions of the file_pointer and file_name are the same as those for opening an input file. However, the access_mode for writing is "wt" where 'w' means the file is opened for writing and 't' means the output is written in text mode. For example, the statement

     myfile = fopen ("3_8.OUT","wt");

     opens file 3_8.OUT for writing. If 3_8.OUT exists, the contents of the file will be overwritten.

3. *Do we need to close the output file?*

   - It is good practice to close files after they have been used. However, if no fclose statements are used, C will automatically close all open files after execution is completed. We also use the fclose() function to close n output file.

Exercises

1. True or False:

```
a. We use the printf() function to write output on the
   screen.
b. We use the fprintf() function to write output to an
   external file.
c. You must open an external file before you can write
   your output in it.
d. It is a good practice to close an output file once you
   don't need it.
e. You must define a file pointer before you can open an
   output file.
```

2. Find error(s), if any, in each statement below:

```
a. #include <stdio.h>
b. FILE myfile;
c. *myfile = fopen (TEST.OUT,wt);
   fprintf(*myfile," Week = %4d\n Year = %5d,n",
            &week,&year);
d. fclose("myfile);
```

3. Write a program to input all your grades in the last semester from the keyboard, compute your average GPA, and write all the input and average GPA on the screen and in a report file name "MYGRADE.REP".

Solutions

1.
```
A B C D E
T T T T T
```

2.
```
a. No error
b. FILE *myfile;
c. myfile = fopen ("TEST.OUT","wt");
d. fprintf(myfile," Week = %4d\n Year = %5d\n"
                week,year);
e. fclose(myfile);
```

Lesson 3_9 - Summary

In this chapter, you have learned int, float, and double data types, how to name and declare variables, input and output format specifications, assignment and arithmetic statements, input from the keyboard and from a file, and output to screen and to a file. The program below summarizes what you have learned in this chapter.

Source Code

```
#include <stdio.h>
main()
 {int      x;
 float  y;
 double z;
 FILE *in, *out;

 printf("Please type a number\n");
 scanf("%d",&x);

 in=fopen("3_9.IN","rt");
 fscanf(input,"%f %1f", &y,&z);

 out=fopen("3_9.OUT","wt");
 fprintf(out, "X=%d\nY=%5.1f\nZ=%5.2f\n",x,y,z);

 fclose(in);
 fclose(out);
}
```

Input file 3_9.IN

```
11.1      22.2
```

On Screen Dialogue

Program output Please type a number
Keyboard input 987

Output file 3_9 .OUT

```
x= 9.87
Y= 11.1
Z= 22.20
```

Exercises

1. Write a program to:

   a. Read the input file 3_9.DAT as shown below (b represents blank);

```
1bb1.1bb1.1
2bb2.2bb2.22
3bb3.3bb3.333
4bb4.4bb4.4444
5bb5.5bb5.55555
```

   b. Display the input file as it is on the screen.

   c. Calculate the average value in each column and write the output as below on the screen and to file 3_9.OUT:

```
       MONTH *** INCOME *** EXPENSES
       -----------------------------
       1         1.1        1.1
       2         2.2        2.22
       3         3.3        3.333
       4         4.4        4.4444
       5         5.5        5.55555
       -----------------------------
 Ave   3         3.3        3.33059
```

Application Program 3_1

Comment on development of programs

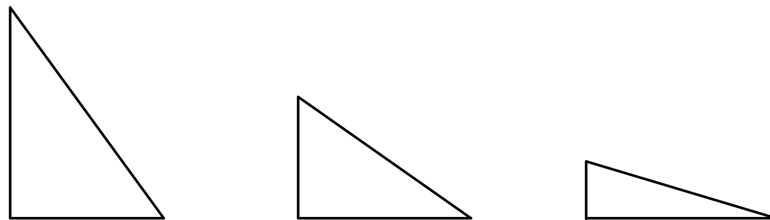
Creating a program which accomplishes the desired task may not be simple. As programs become more complex in this book, the following step-by-step procedure will be used to illustrate the development of the application programs:

1. Assemble the relevant equations.
2. Do a hand calculation of an example problem.
3. Write an algorithm (sometimes called pseudo-code) which uses the equations and follows the pattern of the hand calculation. We recommend that you write an informal algorithm which is roughly a line by line description of what the program does. It should be written in plain English.
4. Use the algorithm to write the actual source code.

This four step method of program development has evolved over the course of teaching by the authors. It has been successfully implemented by many students and will be used in this book to illustrate the development of the application programs. We recommend that you follow this procedure in writing your own programs. However, as you become more adept at programming you may be able to skip some of the steps or develop another method which suites your own style better.

Problem statement:

Write a program which computes the areas of four right triangles. The three of the triangles are shown below. You should deduce the dimensions of the fourth triangle from the pattern L_{h_1} exhibited by the first three. Use the pattern in writing your program



Solution

Assemble relevant equations:

Note that there is a pattern to the length of the legs. The lengths of the horizontal legs are 5, 5 + 1 = 6, 6 + 1 = 7, and the vertical legs are 7, 7/2 = 3.5, 3.5/2 = 1.75. Thus, we can

see that the fourth triangle has a horizontal leg length of 7 + 1 = 8 and a vertical leg length of 1.75/2 = 0.875.

We can see that the horizontal leg length can be computed from the following equations:

$$L_{h_2} = L_{h_1} + 1$$
$$L_{h_3} = L_{h_2} + 1$$
$$L_{h_4} = L_{h_3} + 1$$

where:

L_{h_1} = horizontal leg length of the first horizontal leg = 5.0

L_{h_2} = horizontal leg length of the second horizontal leg

L_{h_3} = horizontal leg length of the third horizontal leg

L_{h_4} = horizontal leg length of the fourth horizontal leg

Also the vertical leg length is:

$$L_{v_2} = L_{v_1} / 2$$
$$L_{v_3} = L_{v_2} / 2$$
$$L_{v_4} = L_{v_3} / 2$$

where:

L_{v_1} = vertical leg length of the first vertical leg
L_{v_2} = vertical leg length of the second vertical leg
L_{v_3} = vertical leg length of the third vertical leg
L_{v_4} = vertical leg length of the fourth vertical leg

Note that the area of a right triangle is:

$$A = 0.5 \ L_1 L_2$$

where L_1 and L_2 are the lengths of the legs.

Specific Example:

For this particular program, the results can be easily found using a hand calculator. For most real programs it is not possible to do this because of the very large number of

calculations that are performed by most real programs. The calculations below show the lengths and the areas.

Triangle 1

L_{h_1} = 5
L_{v_1} = 7
A_1 = (0.5) (5) (7) = 17.50

Triangle 2

L_{h_2} = 5 + 1 = 6
L_{v_2} = 7/2 = 3.5
A_2 = (0.5) (6) (3.5) = 10.50

Triangle 3

L_{h_3} = 6 + 1 = 7
L_{v_3} = 3.5/2 = 1.75
A_3 = (0.5) (7) (1.75) = 6.125

Triangle 4

L_{h_4} = 7 + 1 = 8
L_{v_4} = 1.75/2 = 0.875
A_4 = (0.5) (8) (0.875) = 3.50

Algorithm

One of the purposes of performing a sample calculation is to clearly outline all of the steps that are needed to arrive at a correct and complete result. The sample calculation above has been used as a guide to writing the algorithm shown below:

```
Begin
Declare variables
Initialize horizontal leg length of first triangle
Initialize vertical leg length of first triangle
Calculate area of first triangle
Calculate horizontal leg length of second triangle
Calculate vertical leg length of second triangle
Calculate area of second triangle

Calculate horizontal leg length of third triangle
Calculate vertical leg length of third triangle
Calculate area of third triangle

Calculate horizontal leg length of fourth triangle
Calculate vertical leg length of fourth triangle
Calculate area of fourth triangle
```

```
Print results onto the screen

End
```

Source Code

The below source code has been written directly from the algorithm.

```
main()
{float horizleg, vertleg, areal, area2, area3, area4;

 horizleg  = 5.0;
 vertleg   = 7.0;
 areal     = 0.5 * horizleg * vertleg;

 horizleg +- 1.0;
 vertleg  /= 2.0;
 area2     = 0.5 * horizleg * vertleg;

 horizleg += 1.0;
 vertleg  /= 2.0;
 area3     = 0.5 * horizleg * vertleg;

 horizleg += 1.0;
 vertleg  /= 2.0;
 area4     = 0.5 * horizleg * vertleg;

printf ("      \n\

}
```

Output

```
First triangle area   = 17.50
Second triangle area  = 10.50
Third triangle area   =  6.13
Fourth triangle area  =  3.50
```

Comments:

This program illustrates how patterns are used in programming. One can imagine that it would be very simple to write a program similar to this one which computes the areas of fifty triangles which follow the same pattern. As we illustrate more programming techniques you will see that it will be possible to write such a program with very few statements.

This particular example is somewhat contrived in that it is deliberately set up to have a pattern to it. You will find, though, that real problems will also have patterns and that part of the skill in writing more advanced programs is in recognizing patterns and writing efficient code which takes advantage of the patterns.

Application Program 3_2

Problem statement

Write a program which creates a table of degrees Celsius with the corresponding degrees Fahrenheit. Begin at 0 °C and proceed to 100 °C in 20 °C increments. Use no more than two variables in your program.

Solution

Assemble relevant equations:

The equation converting degrees Celsius to degrees Fahrenheit is:

$$F = \frac{9}{5} C + 32$$

Where:

```
C = degrees Centigrade
F = degrees Fahrenheit
```

Specific example

Once again, for this simple program, all the calculations can be done by hand and are shown below.

C = 0

$$F = C \left(\frac{9}{5}\right) + 32 = 32$$

C = 20

$$F = C \left(\frac{9}{5}\right) + 32 = 68$$

C = 40

$$F = C \left(\frac{9}{5}\right) + 32 = 104$$

C = 60

$$F = C \left(\frac{9}{5}\right) + 32 = 140$$

C = 80

$$F = C \left(\frac{9}{5}\right) + 32 = 176$$

C = 100

$$F = C\left(\frac{9}{5}\right) + 32 = 212$$

Algorithm:

This algorithm is written from the sample calculations with the addition of the printing of the headings and the results.

```
Begin
Declare variables
Print headings of table


Set C = 0
Calculate F
Print C and F

Set C = 20
Calculate F
Print C and F

Set C =  40
Calculate F
Print C and F

Set C 60
Calculate F
Print C and F

Set C = 80
Calculate F
Print C and F

End
```

Source Code

This source code has been written from the algorithm. Note that this code has made use of the fact that the values of degrees centigrade are increments of 20.

```
main()
{float degC, degF;

 printf("Table of Celsius and Fahrenheit degrees\n\n"
 "             Degrees             Degrees \n"
 "             Celsius             Fahrenheit \n");

 degC     = 0.;
 degF     = degC * 9./5. +32.;
 printf("%16.2f %20.2f\n", degC, degF);
```

```
    degC     +=  20.;
    degC     = degC *  9./5. +32.;
    printf("%16.2f %20.2f\n", degC, degF);

    degC     +=  20.;
    degF     = degC *  9./5. +32.;
    printf("%16.2f %20.2f\n", degC, degF);

    degC     += 20.;
    degF     = degC * 9./5.  + 32.;
    printf("%16.2f %20.2f\n", degC, degF);


    degC     +=  20.;
    degF     = degC *  9./5. +32.;
    printf("%16.2f  %20.2f\n", degC, degF);

    degC     += 20.;
    degF     = degC * 9./5. +32.;
    printf("%16.2f %20.2f\n", degC, degF);
   }
```

Output

Table of Celsius and Fahrenheit degrees

```
                   Degrees     Degrees
                   Celsius     Fahrenheit
                      0.00          32.00
                     20.00          68.00
                     40.00         104.00
                     60.00         140.00
                     80.00         176.00
                    100.00         212.00
```

Comments

First, we can see immediately that this program has the same three statements written repeatedly. Had we wanted to display the results for every single degree between 0 and 100 instead of every twentieth degree, the program would have been extremely long but with the same three statements written over and over again. We will learn more advanced programming techniques in Chapter 4 which will allow us to write a program which can accomplish the same task but with many fewer statements.

Second, we could have used the programming technique illustrated in the previous application which had a single printf statement at the end of the program instead of one immediately after each calculation of degF. However, this would have necessitated the use variables.

For instance, the program could have been:

```
main()
{float degC1, degC2, degC3, degC4, degC5, degC6,
       degF1, degF2, degF3, degF4, degF5, degF6;
 printf     ("Table   of   Celsius   and   Fahrenheit
 degrees\n\n"
         "       Degrees           Degrees \n"
         "       Celsius           Fahrenheit\n");

 degC1    = 0.;
 degF1    = degC1 * 9./5. +32.;

 degC2    = 20.;
 degF2    = degC2 * 9./5. +32.;

 degC3    = 40.;
 degF3    = degC3 * 9./5. +32.;

 degC4    = 60.;
 degF4    = degC4 * 9./5. +32.;


 degC5    = 80.;
 degF5    = degC5 * 9./5. +32.;

 degC6    = 100.;
 degF6    = degC6 * 9./5. +32.;

 printf   ("\n"
 "%20.2f %20.2f\n%20.2f %20.2f\n%20.2f %20.2f\n"
 "%20.2f %20.2f\n%20.2f %20.2f\n%20.2f %20.2f\n",
 degC1, degF1, degC2, degF2, degC3, degF3,
 degC4, degF4, degC5, degF5, degC6, degF6);
}
```

With this program 12 variables have been used instead of just two. Variables take up space in the memory of the computer, so the program with 12 variables would occupy more memory than the program with just two variables. You will learn that efficient programming means, in part, to write a program which takes as little memory as possible. For this very small program, either programming technique could be used on today's computers. However, for very large programs the memory needed by the program may be v important. So, it is good to develop efficient programming habits now that you are just learning programming. Reducing memory size is only a part of developing efficient program Comments on other ways to make your program efficient will be made throughout this book.

It should also be noted that it is necessary to make your program understandable to someone than you. The reason for this is that it is common for programs to be developed by teams of people and for programs to undergo several versions. This means that it is

possible that someone who has never seen a particular program may be responsible for modifying it. Thus, your program is more valuable if it is easily understood.

Sometimes you will find that there is a conflict between understandability and efficiency. In other words, efficient programs may not be understandable, and understandable programs may not be efficient. You should consult your employer or your course instructor for guidance in determining the most important characteristics that your program should have.

You can begin to see now that there are many ways to write even the simplest of programs. One can argue that there is no right or wrong way provided the program gives the correct result. However, one can say that it is best to write code that is efficient and understandable.

Application Program 3_3

Problem statement

Write a program which calculates the volume of paint which is needed to paint a room. The paint is to be put on four walls and a ceiling. The room has dimensions:

h = height of our walls
l_1, l_2, l_3, l_4 = lengths of four walls
l_c = length of ceiling
w_c = width of ceiling

The paint thickness can be considered to be constant:

t_1 = thickness of first coat = 0.08 cm
t_2 = thickness of second coat = 0.03 cm

Only one coat of paint is to be put on the ceiling. Two coats will be put on the walls.

The dimensions are to be read from an input file called PAINT.DAT. The contents of PAINT.DAT are:

first line	h			
second line	l_1	l_2	l_3	l_4
third line	l_c	w_c		

These values are all real numbers and the units are meters.

Display the results on the screen in the form;

```
The volume of the paint required to paint the room is:
             .... m3
```

Solution

1. Assemble all relevant equations

The volume of paint on the first wall is (other walls can be calculated similarly):

$$V_1 = l_1 \, h \, (t_1 + t_2) \quad (3_3.1)$$

The volume of paint on the ceiling is:

$$V_c = l_c \, w_c \, t_1 \quad (3_3.2)$$

The total volume of paint used is:

$$V_t = V_1 + V_2 + V_3 + V_4 + V_c \qquad (3_3.3)$$

where:

V_1 = volume of paint on wall number 1
V_c = volume of paint on ceiling
V_t = total volume of paint

2. Specific example

Consider the following dimensions:

$h = 3$ m
$l_1 = 6$ m
$l_2 = 9$ m
$l_3 = 7$ m
$l_4 = 9$ m
$l_c = 9$ m
$w_c = 7$ m

Compute the paint volumes (note that 1/100 is to convert from cm to m):

First wall $V_1 = (6)(3)(0.08+0.03)(1/100) = 0.0198\ m^3$ (3_3.1)
Second wall $V_2 = (9)(3)(0.08+0.03)(1/100) = 0.0297\ m^3$ (3_3.1)
Third wall $V_3 = (7)(3)(0.08+0.03)(1/100) = 0.0231\ m^3$ (3_3.1)
Fourth wall $V_4 = (9)(3)(0.08+0.03)(1/100) = 0.0297\ m^3$ (3_3.1)
Ceiling $V_c = (9)(7)(0.08)(1/100) = 0.0504\ m^3$ (3_3.2)

Total volume
$V_t = 0.0198+0.0297+0.0231+0.0297+0.0504 = 0.1527\ m^3$ (3_3.3)

3. Algorithm

This algorithm follows the example problem, and includes the parts of the program (such as opening the data file) which are not needed in the hand calculation.

```
Define constants
Declare variables
Open data file
Read input data

Calculate paint volume for first wall
Calculate paint volume for second wall
Calculate paint volume for third wall
Calculate paint volume for fourth wall
Calculate paint volume for ceiling
Calculate total volume
Print result to screen
```

4. Source code

Chapter 3

This source code is developed by using the equations and the algorithm. Note that the variable names used in the program are much more descriptive than the single letter variables used in the equations.

```
main()
{#define THK_COAT1 0.08
 #define THK_COAT2 0.03

 float length_wall, length_wall2,  length_wall3,  length_wall 4,
 vol_wall1,         vol_wall2,          vol_wall3,
     vol_wall4,
 length_ceil,       width_ceil,    height,        vol_tot;

     /*        Open input file               */

 FILE     *inptr;
 inptr =  fopen ("PAINT.DAT","rt");

     /*        Read input file               */

 fscanf ("%f",            &height);
 fscanf ("%f %f %f %f",  &length_wall1,&length_wall2,
                          &length_wall3,&length_wall4);
 fscanf ("%f %f",         &length_ceil, &width_ceil);

     /*        Compute paint volumes    */

 vol_wall1=length_wall1 * height * (THK_COAT1+THK_COAT2);
 vol_wall2=length_wall2 * height * (THK_COAT1+THK_COAT2);
 vol_wall3=length_wall3 * height * (THK_COAT1+THK_COAT2);
 vol_wall4=length_wall4 * height * (THK_COAT1+THK_COAT2);
 vol_ceil =length_ceil * width  * (THK_COAT1);
 vol_tot  =vol_wall1 + vol_wall2 + vol_wall3 +
 vol_wall4+vol_ceil;

     /*        Print  results to the screen  */

 printf ("\nThe volume of the paint \n"
        " required to paint the room is:    \n%7.4f   m^3",
        vol_tot);
}
```

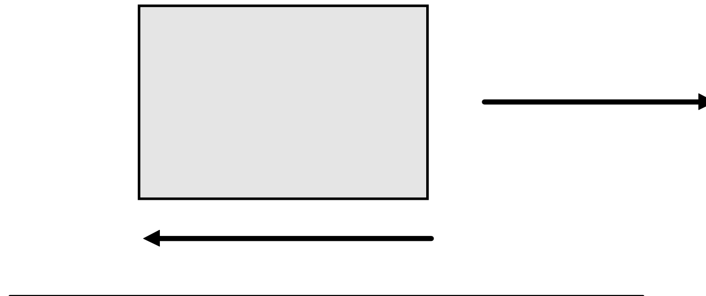
5. Comments

This is a relatively simple program. It can be seen for simple programs such as this one, much of the program is occupied by such things as declaring variables, reading the input file, and printing the results. For more complex programs, most of the program is involved with doing the actual calculations.

Application Program 3_4

Problem statement

Write a program which computes the necessary force to move a block across a plane. Friction resists the movement of the block. Consider three different blocks.



The three different blocks have the following dimensions:

Block	Height (m)	Length (m)	Width (m)
1	0.5	2.0	1.5
2	1.2	0.75	0.2
3	0.8	2.2	1.3

The density of the material of which the blocks are made is 5.7 kN/m^3. The coefficient of friction of each block on the plane is 0.35.

The data file containing the dimensions of each block is FRICTION.DAT. The data file has tree lines - each with a block number, height, length and width as listed above. Print to the screen the force required to move the block.

Solution

1. Assemble relevant equations

Because friction is the only force resisting movement of the block, the force required to move the block is equal to the frictional resistance. You have learned from your first physics class that frictional resistance is a function of the coefficient of friction and the normal force on the plane of contact:

$$F = \mu N \qquad (3_4.1)$$

where:

F = frictional resistance
μ = coefficient of friction

```
N = normal force on the plane of contact
```

For this particular case, the normal force is equal to the weight of the block, that is

```
N = w                                                (3_4.2)
```

where:

```
W block weight
```

The block weight can be computed from the volume and the density, and the volume can be computed from the dimensions as follows:

```
W = Vd                                               (3_4.3)
V = hlw                                              (3_4.4)
```

where:

```
V = block volume
d = material density
h = block height
l = block length
w = block width
```

2. Perform an example calculation

In many cases you will have to make up the example problem yourself, which means that you will need to create the information that goes into the input file. In this case, however, the input data is given. The below calculation sequence uses the information about the first block:

```
d = 5.7 kn/m3
μ = 0.35
h = 0.5 m
l = 2.0 m
w = 1.5 m
V = (0.5)(2.0)(1.5) = 1.5 m3                         (eqn.
3_4.4)
W = (1.5)(5.7)      = 8.55 kN                    (eqn. 3_4.3)
N = 8.55 kN                                      (eqn. 3_4.2)
F = (0.35)(8.55)    = 2.9925 kN                  (eqn. 3_4.1)
```

3. Algorithm

```
Define constants
Declare variables
Open the input data file

Read first block information
     Compute   V
```

```
Compute   W
Compute   N
Compute   F
Print     F
```

```
Read second block information
     Compute   V
     Compute   W
     Compute   N
     Compute   F
     Print     F

Read third block information
     Compute   V
     Compute   W
     Compute   N
     Compute   F
     Print     F
```

4. Source code

The program shown below is one which satisfies the requirements of the problem statement. The algorithm has been used to write the source code.

```
#include <stdio.h>
main()
{
#define DENSITY               5.7
#define FRICTION_COEFF   0.35

 int      block_number;
 float    weight, height, length, width, volume,
          normal_force, frictional_force, movement_force;

 FILE     *inptr;
 inptr =  fopen (“FRICTION.DAT”,”rt”);
 printf   (“\n\
 Block number  Force required for movement(KN)\n\n”);

 fscanf   (inptr, “%d %f %f %f”,
           &block_number, &height, &length, &width);

 volume              = height * length * width;
 weight              = volume * DENSITY;
 normal_force        = weight;
 frictional_force    = FRICTION_COEFF * Normal_force:
 movement_force           = frictional_force;
 printf(“\n%10d                                 %30.7f
 \n”,block_number,movement_force);

 fscanf (inptr, “%d %f %f %f”,
         &block_number, &height, &length, &width);

 volume              = height * length * width;
 weight              = volume * DENSITY;
 normal_force        = weight;
```

```
frictional_force   = FRICTION_COEFF * normal_force;
movement_force          = frictional_force;
printf("\n%10d                                  %30.7f
\n",block_number,movement_force);

fscanf (inptr, "%d %f %f %f",
        &block_number, &height, &length, &width);

volume             = height * length * width;
weight             = volume * DENSITY;
normal_force       = weight;
frictional_force   = FRICTION_COEFF * normal_force;
movement_force          = frictional_force;
printf("\n%10d                                  %30.7f
\n",block_number,movement_force);
}
```

Output

```
Block number        Force  required  for movement (kN)

     1                             2.9925
     2                             0.3591
     3                             4.56456
```

Comments

Use your calculator to check the results shown above. Note that once again several statement have been repeated. Also note that every program an be written many ways. Do you have any ideas on how to change this program to fit your own personal style of programming?

APPLICATIONS EXERCISES

Use the four step procedure outlined in this chapter to write the following programs.

3_1. Write a program which creates a table of Olympic competition running distances in meters, kilometers, yards and miles. The following distances should be used:

```
100 m
200 m
400 m
800 m
```

Use the pattern exhibited in these distance to write your program. Call your program OLYM.C. (Note: 1m = 0.001 km = 1.094 yd = 0.0006215 mi.)

Input specifications

No external input (meaning no data input from the keyboard or file). All distances are real numbers.

Output specifications

Print the results to the screen in the following manner:

```
              Table of Olympic running distances

      Meters     Kilometers          Yards          Miles

         100            ---            ---            ---
         200            ---            ---            ---
         400            ---            ---            ---
         800            ---            ---            ---
```

Right justify the numbers in the table

3_2. Write the program described in Application Exercise 3_1 with the output going to a file called OLYM.OUT. Left justify the numbers in the table.

3_3. Write a program which computes the length of the hypotenuse of 5 right triangles based on the lengths of the two legs. Call your program HYPLENG.C.

Input specifications

Read the input data from the keyboard by prompting the user in the following way:

```
Screen output     Input the values of the leg lengths
                  for five right triangles

Keyboard input         leg1 leg2
Keyboard input         leg1 leg2
Keyboard input         leg1 leg2
Keyboard input         leg1 leg2
```

All input values are real numbers.

Output specifications

Print the result to the file HYPLENG.OUT with the following format:

```
            Hypotenuse lengths of five triangles

   Triangle        leg1           leg 2        hypotenuse
    number        length         length          length

      1            ---            ---             ---
      2            ---            ---             ---
      3            ---            ---             ---
      4            ---            ---             ---
```

Right justify all of the numbers in the table.

3_4. Write a program which computes the values of the two acute angles of a right triangle given the lengths of the two legs. Call your program ANGLE.C. Create the input data file ANGLE.DAT before executing your program.

Input should come from data file ANGLE.DAT with the following form:

```
line 1      leg1      leg2
line 2      leg1      leg2
line 3      leg1      leg2
line 4      leg1      leg2
line 5      leg1      leg2
```

All of the values are real numbers.

Output specifications

The output results should be in degrees, not radians. Make sure that in your program you convert from radians to degrees. The output should go to file ANGLE.OUT and have the following format:

Acute angles of five triangles

Triangle number	acute angle 1	acute angle 2
1	---	---
2	---	---
3	---	---
4	---	---
5	---	---

3_5. Write a program which is capable of displaying the distances from the sun to the four planets closest to the sun in centimeters and inches given the kilometer distances as follows:

Planet	Distance from the sun (million km)
Mercury	58
Venus	108.2
Earth	149.5
Mars	227.8

Input specifications

No external input. The distances listed above can be initialized in the source code.

Output specifications

Print the results to the screen in the form of the table shown below:

Planet	Distance (million km)	from the sun (cm)	(inches)
Mercury	58	---	---
Venus	108.2	---	---
Earth	149.5	---	---
Mars	227.8	---	---

Note: In order to fit the numbers properly in the table, you must use scientific notation.

3_6. The distance that a car (undergoing constant acceleration) will travel is given by the expression

$$s = v_0 t + 1/2 a t^2$$

where:

```
s = distance traveled
v0 = initial velocity
t = time of travel
a = acceleration
```

Write a program which computes this distance give v_0, t and a. Call your program DISTANCE.C.

Input specifications

The input should come from the file DISTANCE.DAT with the following format:

```
line 1      v0    t
line 2      a1
line 3      a2
line 4      a3
line 5      a4
line 6      a5
line 7      v0    t
line 8      a1
line 9      a2
line 10     a3
line 11     a4
line 12     a5
```

All of the above numbers are real numbers. An example data file is:

```
10      5
3
4
5
6
7
10      10
3
4
5
6
7
```

Output specifications

Print the results to the file DISTANCE.OUT in the following form:

```
                Car under constant acceleration

 Initial          time        acceleration      distance
Velocity
```

10	5	3	---
		4	---
		5	---
		6	---
10	10	3	---
		4	---
		5	---
		6	---
		7	---

3_7. The general gas law for an ideal gas is given by:

```
PV/T  =  constant
```

where:

```
P = pressure
V = volume
T = temperature (Rankine or Kelvin)
```

which leads to the equation:

$$P_1V_1/T_1 = P_2V_2/T_2$$

for a given mass of gas.

Write a computer program which computes the temperature of a gas which is originally at:

P_1 = 5 atmospheres
V_1 = 30 liters
T_1 = 273 deg Kelvin

Call your program TEMPER.C.

Input specifications

The input data should come from the file TEMPER.DAT and consists of five lines:

line 1	P_2	V_2
line 2	P_3	V_3
line 3	P_4	V_4
line 4	P_5	V_5
line 5	P_6	V_6

A sample data file is:

```
2  40
3  80
```

```
6  50
1  15
2  70
```

All of the above values are real.

Output specifications

Your output should be to the screen and consist of the following table.

```
The below listed pressure, volume and temperature
conditions can occur for a given mass of an ideal
gas which is originally at P = 5 atm, V = 30 1,
and T = 273 K
```

Case	P(atm)	V(1)	T(K)
1	2	40	---
2	3	80	---
3	6	50	---
4	1	15	---
5	2	70	---

3_8. Ohm's law for a steady electrical current can be written as:

```
V = I R
```

where:

```
V = potential difference across a conductor
I = current in the conductor
R = resistance of the conductor
```

Write a program (called OHM.C) which is capable of filling in the blanks in the following table:

Case	V (Volts)	I (Amps)	R (Ohms)
1	10	2	--
2	--	5	7
3	3	--	4

Input specifications

The input data should come from the keyboard and be treated as real numbers. You should prompt the user in the following manner:

```
"For case 1, enter the voltage and current."
"For case 2, enter the current and resistance."
"For case 3, enter the voltage and resistance."
```

Output specifications

Print the completed table to the screen.

3_9. The pressure at depth in water is given by:

$$P = h\ \gamma_w$$

where:

```
p = pressure
h = depth
```
γ_w = weight density of water

Write a program which determines the pressure at five different depths. Call your program PRESS.C. Use metric units ($\gamma w = 9.8\ kN/m^2$).

Input specifications

Create a data file called PRESS.DAT with your editor. In the data file list the five depths on one line:

```
depth1    depth2    depth3    depth4    depth5
```

An example data file is:

```
10.       15.       828.      1547.         431.2
```

All of the above data are real.

Output specifications

Print the results to file PRESS.OUT in the following form:

```
Depth        Pressure
 (m)          (kPa)

 ---           ---
 ---           ---
 ---           ---
 ---           ---
 ---           ---
```

3_10. The period of one swing of a simple pendulum is given by:

$$T = 2\pi\sqrt{\frac{T}{G}}$$

where (in metric units):

```
T = period (sec)
l = length of pendulum (m)
g = gravitational acceleration = 9.81 m/sec²
```

Write a program which is capable of completing the following table:

Length (m)	Period (sec)
0.5	---
1.0	---
---	10.
---	20.
0.32	---

Input specifications

Prompt the user to input the data from the screen in a manner similar to that described in the previous exercise.

Output specifications

Print the completed table to the screen.

3_11. The kinetic energy of an object in motion is expressed as:

$$K = \frac{1}{2} m v^2$$

where:

```
k = kinetic energy of object
m = mass of object
v = velocity of object
```

The work done by a force pushing on an object in the direction of the object's motion is:

$$W = F s$$

where:

```
W  = work done by the force
F  = force on object
```

```
s  = distance traveled by the object during the time the
     object is pushed
```

For an object pushed horizontally from rest, k = W so that:

$$F\ s = \frac{1}{2}\ m\ v^2$$

Assume that one person can push with the force of 0.8 kN and that we have a car of m = 1000 kg. Write a program which can complete the following table:

Distance shed (m)	Final velocity (m/sec)	Number of people required to push
5	10	---
---	10	15
20	---	8

Input specifications

Prompt the user to enter the data from the keyboard.

Output specifications

Print the completed table to the screen.

Chapter 3