Chapter

3

Operators and Expressions

3.1 INTRODUCTION

C supports a rich set of operators. We have already used several of them, such as =, +, –, *, & and <. An *operator* is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. They usually form a part of the mathematical or logical *expressions*.

C operators can be classified into a number of categories. They include:

1. Arithmetic operators.
2. Relational operators.
3. Logical operators.
4. Assignment operators.
5. Increment and decrement operators.
6. Conditional operators.
7. Bitwise operators.
8. Special operators.

An expression is a sequence of operands and operators that reduces to a single value. For example,

10 + 15

is an expression whose value is 25. The value can be any type other than ***void***.

3.2 ARITHMETIC OPERATORS

C provides all the basic arithmetic operators. They are listed in Table 3.1. The operators +, –, *, and / all work the same way as they do in other languages. These can operate on any built-in data type allowed in C. The unary minus operator, in effect, multiplies its single operand by −1. Therefore, a number preceded by a minus sign changes its sign.

Table 3.1 *Arithmetic Operators*

Operator	*Meaning*
+	Addition or unary plus
−	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division

Integer division truncates any fractional part. The modulo division operation produces the remainder of an integer division. Examples of use of arithmetic operators are:

```
a – b   a + b
a * b   a / b
a % b   -a * b
```

Here **a** and **b** are variables and are known as *operands*. The modulo division operator % cannot be used on floating point data. Note that C does not have an operator for *exponentiation*. Older versions of C does not support unary plus but ANSI C supports it.

Integer Arithmetic

When both the operands in a single arithmetic expression such as a+b are integers, the expression is called an *integer expression*, and the operation is called *integer arithmetic*. Integer arithmetic always yields an integer value. The largest integer value depends on the machine, as pointed out earlier. In the above examples, if **a** and **b** are integers, then for **a** = 14 and **b** = 4 we have the following results:

```
a – b  =  10
a + b  =  18
a * b  =  56
a / b  =  3 (decimal part truncated)
a % b  =  2 (remainder of division)
```

During integer division, if both the operands are of the same sign, the result is truncated towards zero. If one of them is negative, the direction of trunction is implementation dependent. That is,

6/7 = 0 and -6/-7 = 0

but −6/7 may be zero or −1. (Machine dependent)

Similarly, during modulo division, the sign of the result is always the sign of the first operand (the dividend). That is

```
-14 % 3  = -2
-14 % -3 = -2
 14 % -3 = 2
```

Example 3.1 The program in Fig. 3.1 shows the use of integer arithmetic to convert a given number of days into months and days.

Program
```
main ()
{
  int months, days ;

  printf("Enter days\n") ;
  scanf("%d", &days) ;

  months = days / 30 ;
  days = days % 30 ;
  printf("Months = %d Days = %d", months, days) ;
}
```

Output
```
Enter days
265
Months = 8 Days = 25
Enter days
364
Months = 12 Days = 4
Enter days
45
Months = 1 Days = 15
```

Fig. 3.1 *Illustration of integer arithmetic*

The variables months and days are declared as integers. Therefore, the statement

```
months = days/30;
```

truncates the decimal part and assigns the integer part to months. Similarly, the statement

```
days = days%30;
```

assigns the remainder part of the division to days. Thus the given number of days is converted into an equivalent number of months and days and the result is printed as shown in the output.

Real Arithmetic

An arithmetic operation involving only real operands is called *real arithmetic*. A real operand may assume values either in decimal or exponential notation. Since floating point values are rounded to the number of significant digits permissible, the final value is an approximation of the correct result. If **x**, **y**, and **z** are **floats**, then we will have:

$x = 6.0/7.0 = 0.857143$

$y = 1.0/3.0 = 0.333333$

z = –2.0/3.0 = –0.666667

The operator % cannot be used with real operands.

Mixed-mode Arithmetic

When one of the operands is real and the other is integer, the expression is called a *mixed-mode arithmetic* expression. If either operand is of the real type, then only the real operation is performed and the result is always a real number. Thus

15/10.0 = 1.5

whereas

15/10 = 1

More about mixed operations will be discussed later when we deal with the evaluation of expressions.

3.3 RELATIONAL OPERATORS

We often compare two quantities and depending on their relation, take certain decisions. For example, we may compare the age of two persons, or the price of two items, and so on. These comparisons can be done with the help of *relational operators*. We have already used the symbol '<', meaning 'less than'. An expression such as

a < b or 1 < 20

containing a relational operator is termed as a *relational expression*. The value of a relational expression is either *one* or *zero*. It is *one* if the specified relation is *true* and *zero* if the relation is *false*. For example

10 < 20 is true

but

20 < 10 is false

C supports six relational operators in all. These operators and their meanings are shown in Table 3.2.

Table 3.2 *Relational Operators*

Operator	*Meaning*
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to
!=	is not equal to

A simple relational expression contains only one relational operator and takes the following form:

ae-1 relational operator *ae-2*

ae-1 and *ae-2* are arithmetic expressions, which may be simple constants, variables or combination of them. Given below are some examples of simple relational expressions and their values:

4.5 <= 10	TRUE	
4.5 < -10	FALSE	
-35 >= 0	FALSE	
10 < 7+5	TRUE	
a+b = c+d	TRUE	only if the sum of values of a and b is equal to the sum of values of c and d.

When arithmetic expressions are used on either side of a relational operator, the arithmetic expressions will be evaluated first and then the results compared. That is, arithmetic operators have a higher priority over relational operators.

Relational expressions are used in *decision statements* such as **if** and **while** to decide the course of action of a running program. We have already used the **while** statement in Chapter 1. Decision statements are discussed in detail in Chapters 5 and 6.

Relational Operator Complements

Among the six relational operators, each one is a complement of another operator.

>	is complement of	<=
<	is complement of	>=
==	is complement of	!=

We can simplify an expression involving the *not* and the *less than* operators using the complements as shown below:

Actual one	*Simplified one*
!(x<y)	x >= y
!(x>y)	x <= y
!(x!=y)	x == y
!(x<=y)	x > y
!(x>=y)	x < y
!(x == y)	x != y

3.4 LOGICAL OPERATORS

In addition to the relational operators, C has the following three *logical operators*.

&&	meaning logical	AND
||	meaning logical	OR
!	meaning logical	NOT

The logical operators && and || are used when we want to test more than one condition and make decisions. An example is:

a > b && x == 10

An expression of this kind, which combines two or more relational expressions, is termed as a *logical expression* or a *compound relational expression*. Like the simple relational expressions, a logical expression also yields a value of *one* or *zero*, according to the truth table shown in Table 3.3. The logical expression given above is true only if **a** > **b** is *true* and **x** == 10 is *true*. If either (or both) of them are false, the expression is *false*.

Table 3.3 *Truth Table*

op-1	*op-2*	*Value of the expression*	
		op-1 && op-2	*op-1 || op-2*
Non-zero	Non-zero	1	1
Non-zero	0	0	1
0	Non-zero	0	1
0	0	0	0

Some examples of the usage of logical expressions are:

1. if (age > 55 && salary < 1000)
2. if (number < 0 || number > 100)

We shall see more of them when we discuss decision statements.

3.5 ASSIGNMENT OPERATORS

Assignment operators are used to assign the result of an expression to a variable. We have seen the usual assignment operator, '='. In addition, C has a set of '*shorthand*' assignment operators of the form

v op= *exp*;

Where *v* is a variable, *exp* is an expression and *op* is a C binary arithmetic operator. The operator **op**= is known as the shorthand assignment operator.

The assignment statement

```
v op= exp;
```

is equivalent to

```
v = v op (exp);
```

with **v** evaluated only once. Consider an example

```
x += y+1;
```

This is same as the statement

```
x = x + (y+1);
```

The shorthand operator += means 'add y+1 to x' or 'increment x by y+1'. For y = 2, the above statement becomes

```
x += 3;
```

and when this statement is executed, 3 is added to x. If the old value of x is, say 5, then the new value of x is 8. Some of the commonly used shorthand assignment operators are illustrated in Table 3.4.

Table 3.4 *Shorthand Assignment Operators*

Statement with simple assignment operator	*Statement with shorthand operator*
a = a + 1	a += 1
a = a – 1	a –= 1
a = a * (n+1)	a *= n+1
a = a / (n+1)	a /= n+1
a = a % b	a %= b

The use of shorthand assignment operators has three advantages:

1. What appears on the left-hand side need not be repeated and therefore it becomes easier to write.
2. The statement is more concise and easier to read.
3. The statement is more efficient.

These advantages may be appreciated if we consider a slightly more involved statement like

```
value(5*j-2) = value(5*j-2) + delta;
```

With the help of the += operator, this can be written as follows:

```
value(5*j-2) += delta;
```

It is easier to read and understand and is more efficient because the expression 5*j–2 is evaluated only once.

Example 3.2 Program of Fig. 3.2 prints a sequence of squares of numbers. Note the use of the shorthand operator *= .

The program attempts to print a sequence of squares of numbers starting from 2. The statement

```
a *= a;
```

which is identical to

```
a = a*a;
```

replaces the current value of **a** by its square. When the value of **a** becomes equal or greater than **N** (=100) the **while** is terminated. Note that the output contains only three values 2, 4 and 16.

Program

```
#define  N 100
#define  A 2
main()
{
     int a;
     a = A;
     while( a < N )
     {
            printf("%d\n", a);
            a *= a;
     }
}
```

Output

```
2
4
16
```

Fig. 3.2 *Use of shorthand operator *=*

3.6 INCREMENT AND DECREMENT OPERATORS

C allows two very useful operators not generally found in other languages. These are the increment and decrement operators:

```
++ and --
```

The operator ++ adds 1 to the operand, while -- subtracts 1. Both are unary operators and takes the following form:

```
++m; or m++;
--m; or m--;

++m; is equivalent to m = m+1; (or m += 1;)
--m; is equivalent to m = m-1; (or m -= 1;)
```

We use the increment and decrement statements in **for** and **while** loops extensively.

While ++m and m++ mean the same thing when they form statements independently, they behave differently when they are used in expressions on the right-hand side of an assignment statement. Consider the following:

```
m = 5;
y = ++m;
```

In this case, the value of y and m would be 6. Suppose, if we rewrite the above statements as

```
m = 5;
y = m++;
```

then, the value of y would be 5 and m would be 6. *A prefix operator first adds 1 to the operand and then the result is assigned to the variable on left. On the other hand, a postfix operator first assigns the value to the variable on left and then increments the operand.*

Similar is the case, when we use ++ (or – –) in subscripted variables. That is, the statement

```
a[i++] = 10;
```

is equivalent to

```
a[i] = 10;
i = i+1;
```

The increment and decrement operators can be used in complex statements. Example:

```
m = n++ -j+10;
```

Old value of n is used in evaluating the expression. n is incremented after the evaluation. Some compilers require a space on either side of n++ or ++n.

Rules for ++ and – – Operators

- Increment and decrement operators are unary operators and they require variable as their operands.
- When postfix ++ (or – –) is used with a variable in an expression, the expression is evaluated first using the original value of the variable and then the variable is incremented (or decremented) by one.
- When prefix ++(or – –) is used in an expression, the variable is incremented (or decremented) first and then the expression is evaluated using the new value of the variable.
- The precedence and associatively of ++ and – – operators are the same as those of unary + and unary –.

3.7 CONDITIONAL OPERATOR

A ternary operator pair "? :" is available in C to construct conditional expressions of the form

```
exp1 ? exp2 : exp3
```

where *exp1, exp2*, and *exp3* are expressions.

The operator ? : works as follows: *exp1* is evaluated first. If it is nonzero (true), then the expression *exp2* is evaluated and becomes the value of the expression. If *exp1* is false, *exp3* is evaluated and its value becomes the value of the expression. Note that only one of the expressions (either *exp2* or *exp3*) is evaluated. For example, consider the following statements.

```
a = 10;
b = 15;
x = (a > b) ? a : b;
```

In this example, x will be assigned the value of b. This can be achieved using the **if..else** statements as follows:

```
if (a > b)
   x = a;
else
   x = b;
```

3.8 BITWISE OPERATORS

C has a distinction of supporting special operators known as *bitwise operators* for manipulation of data at bit level. These operators are used for testing the bits, or shifting them right or left. Bitwise operators may not be applied to **float** or **double**. Table 3.5 lists the bitwise operators and their meanings. They are discussed in detail in Appendix I.

Table 3.5 *Bitwise Operators*

Operator	*Meaning*
&	bitwise AND
|	bitwise OR
^	bitwise exclusive OR
<<	shift left
>>	shift right

3.9 SPECIAL OPERATORS

C supports some special operators of interest such as comma operator, **sizeof** operator, pointer operators (**&** and *) and member selection operators (. and –>). The comma and **sizeof** operators are discussed in this section while the pointer operators are discussed in Chapter 11. Member selection operators which are used to select members of a structure are discussed in Chapters 10 and 11. ANSI committee has introduced two preprocessor operators known as "string-izing" and "token-pasting" operators (# and ##). They will be discussed in Chapter 14.

The Comma Operator

The comma operator can be used to link the related expressions together. A comma-linked list of expressions are evaluated *left to right* and the value of *right-most* expression is the value of the combined expression. For example, the statement

```
value = (x = 10, y = 5, x+y);
```

first assigns the value 10 to **x**, then assigns 5 to y, and finally assigns 15 (i.e. 10 + 5) to **value**. Since comma operator has the lowest precedence of all operators, the parentheses are necessary. Some applications of comma operator are:

In **for** loops:

```
for ( n = 1, m = 10, n <=m; n++, m++)
```

In **while** loops:

```
while (c = getchar( ), c != '10')
```

Exchanging values:

```
t = x, x = y, y = t;
```

The sizeof Operator

The **sizeof** is a compile time operator and, when used with an operand, it returns the number of bytes the operand occupies. The operand may be a variable, a constant or a data type qualifier.

Examples: m = **sizeof**(sum);
n = **sizeof**(long int);
k = **sizeof**(235L);

The **sizeof** operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer. It is also used to allocate memory space dynamically to variables during execution of a program.

Example 3.3 In Fig. 3.3, the program employs different kinds of operators. The results of their evaluation are also shown for comparison.

Notice the way the increment operator ++ works when used in an expression. In the statement

```
c = ++a - b;
```

new value of **a** (= 16) is used thus giving the value 6 to c. That is, a is incremented by 1 before it is used in the expression. However, in the statement

```
d = b++ + a;
```

the old value of **b** (=10) is used in the expression. Here, b is incremented by 1 after it is used in the expression.

We can print the character % by placing it immediately after another % character in the control string. This is illustrated by the statement

```
printf("a%%b = %d\n", a%b);
```

The program also illustrates that the expression

```
c > d ? 1 : 0
```

assumes the value 0 when c is less than d and 1 when c is greater than d.

Program

```
main()
{
    int a, b, c, d;

    a = 15;
    b = 10;
    c = ++a - b;

    printf("a = %d b = %d c = %d\n",a, b, c);
```

```
            d = b++ +a;

            printf("a = %d b = %d d = %d\n",a, b, d);
            printf("a/b = %d\n", a/b);
            printf("a%%b = %d\n", a%b);
            printf("a *= b = %d\n", a*=b);
            printf("%d\n", (c>d) ? 1 : 0);
            printf("%d\n", (c<d) ? 1 : 0);
        }
```

Output

```
a = 16 b = 10 c = 6
a = 16 b = 11 d = 26
a/b = 1
a%b = 5
a *= b = 176
0
1
```

Fig. 3.3 *Further illustration of arithmetic operators*

3.10 ARITHMETIC EXPRESSIONS

An arithmetic expression is a combination of variables, constants, and operators arranged as per the syntax of the language. We have used a number of simple expressions in the examples discussed so far. C can handle any complex mathematical expressions. Some of the examples of C expressions are shown in Table 3.6. Remember that C does not have an operator for exponentiation.

Table 3.6 *Expressions*

Algebraic expression	*C expression*
a x b - c	a * b - c
(m+n) (x+y)	(m+n) * (x+y)
$\left(\frac{ab}{c}\right)$	a * b/c
$3x^2 + 2x + 1$	3 * x * x + 2 * x + 1
$\left(\frac{x}{y}\right) + c$	x/y+c

3.11 EVALUATION OF EXPRESSIONS

Expressions are evaluated using an assignment statement of the form *variable = expression;*

Variable is any valid C variable name. When the statement is encountered, the expression is evaluated first and the result then replaces the previous value of the variable on the left-hand side. All variables used in the expression must be assigned values before evaluation is attempted. Examples of evaluation statements are

```
x = a * b - c;
y = b / c * a;
z = a - b / c + d;
```

The blank space around an operator is optional and adds only to improve readability. When these statements are used in a program, the variables a, b, c, and d must be defined before they are used in the expressions.

Example 3.4 The program in Fig. 3.4 illustrates the use of variables in expressions and their evaluation.

Output of the program also illustrates the effect of presence of parentheses in expressions. This is discussed in the next section.

Program

```
main()
{
    float a, b, c, x, y, z;

    a = 9;
    b = 12;
    c = 3;

    x = a - b / 3 + c * 2 - 1;
    y = a - b / (3 + c) * (2 - 1);
    z = a - (b / (3 + c) * 2) - 1;

    printf("x = %f\n", x);
    printf("y = %f\n", y);
    printf("z = %f\n", z);
}
```

Output

```
x = 10.000000
y = 7.000000
z = 4.000000
```

Fig. 3.4 *Illustrations of evaluation of expressions*

3.12 PRECEDENCE OF ARITHMETIC OPERATORS

An arithmetic expression without parentheses will be evaluated from *left to right* using the rules of precedence of operators. There are two distinct priority levels of arithmetic operators in C:

High priority * / %

Low priority + –

The basic evaluation procedure includes 'two' left-to-right passes through the expression. During the first pass, the high priority operators (if any) are applied as they are encountered. During the second pass, the low priority operators (if any) are applied as they are encountered. Consider the following evaluation statement that has been used in the program of Fig. 3.4.

x = a–b/3 + c*2–1

When a = 9, b = 12, and c = 3, the statement becomes

x = 9–12/3 + 3*2–1

and is evaluated as follows

First pass

Step1: x = 9–4+3*2–1
Step2: x = 9–4+6–1

Second pass

Step3: x = 5+6–1
Step4: x = 11–1
Step5: x = 10

These steps are illustrated in Fig. 3.5. The numbers inside parentheses refer to step numbers.

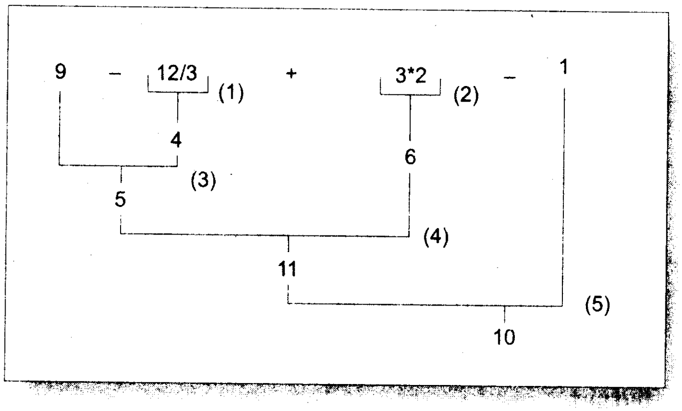


Fig. 3.5 *Illustration of hierarchy of operations*

However, the order of evaluation can be changed by introducing parentheses into an expression. Consider the same expression with parentheses as shown below:

9–12/(3+3)*(2–1)

Whenever parentheses are used, the expressions within parentheses assume highest priority. If two or more sets of parentheses appear one after another as shown above, the expression contained in the left-most set is evaluated first and the right-most in the last. Given below are the new steps.

First pass

Step1: 9-12/6 * (2-1)
Step2: 9-12/6 * 1

Second pass

Step3: 9-2 * 1
Step4: 9-2

Third pass

Step5: 7

This time, the procedure consists of three left-to-right passes. However, the number of evaluation steps remains the same as 5 (i.e equal to the number of arithmetic operators).

Parentheses may be nested, and in such cases, evaluation of the expression will proceed outward from the innermost set of parentheses. Just make sure that every opening parenthesis has a matching closing parenthesis. For example

$$9 - (12/(3+3) * 2) - 1 = 4$$

whereas

$$9 - ((12/3) + 3 * 2) - 1 = -2$$

While parentheses allow us to change the order of priority, we may also use them to improve understandability of the program. When in doubt, we can always add an extra pair just to make sure that the priority assumed is the one we require.

Rules for Evaluation of Expression

- First, parenthesized sub expression from left to right are evaluated.
- If parentheses are nested, the evaluation begins with the innermost sub-expression.
- The precedence rule is applied in determining the order of application of operators in evaluating sub-expressions
- The associativity rule is applied when two or more operators of the same precedence level appear in a sub-expression.
- Arithmetic expressions are evaluated from left to right using the rules of precedence.
- When parentheses are used, the expressions within parentheses assume highest priority.

3.13 SOME COMPUTATIONAL PROBLEMS

When expressions include real values, then it is important to take necessary precautions to guard against certain computational errors. We know that the computer gives approximate values for real numbers and the errors due to such approximations may lead to serious problems. For example, consider the following statements:

```
a = 1.0/3.0;
b = a * 3.0;
```

We know that (1.0/3.0) 3.0 is equal to 1. But there is no guarantee that the value of **b** computed in a program will equal 1.

Another problem is division by zero. On most computers, any attempt to divide a number by zero will result in abnormal termination of the program. In some cases such a division may produce meaningless results. Care should be taken to test the denominator that is likely to assume zero value and avoid any division by zero.

The third problem is to avoid overflow or underflow errors. It is our responsibility to guarantee that operands are of the correct type and range, and the result may not produce any overflow or underflow.

Example 3.5 Output of the program in Fig. 3.6 shows round-off errors that can occur in computation of floating point numbers.

Program

```
/*---------- Sum of n terms of 1/n ----------*/
  main()
  {
       float sum, n, term ;
       int count = 1 ;

       sum = 0 ;
       printf("Enter value of n\n") ;
         scanf("%f", &n) ;
       term = 1.0/n ;
       while( count <= n )
       {
            sum = sum + term ;
            count++ ;
       }
       printf("Sum = %f\n", sum) ;
  }
```

Output

```
  Enter value of n
  99
  Sum = 1.000001
  Enter value of n
  143
  Sum = 0.999999
```

Fig. 3.6 *Round-off errors in floating point computations*

We know that the sum of n terms of 1/n is 1. However, due to errors in floating point representation, the result is not always 1.

3.14 TYPE CONVERSIONS IN EXPRESSIONS

Implicit Type Conversion

C permits mixing of constants and variables of different types in an expression. C automatically converts any intermediate values to the proper type so that the expression can be evaluated without loosing any significance. This automatic conversion is known as *implicit type conversion.*

During evaluation it adheres to very strict rules of type conversion. If the operands are of different types, the 'lower' type is automatically converted to the 'higher' type before the operation proceeds. The result is of the higher type. A typical type conversion process is illustrated in Fig. 3.7.

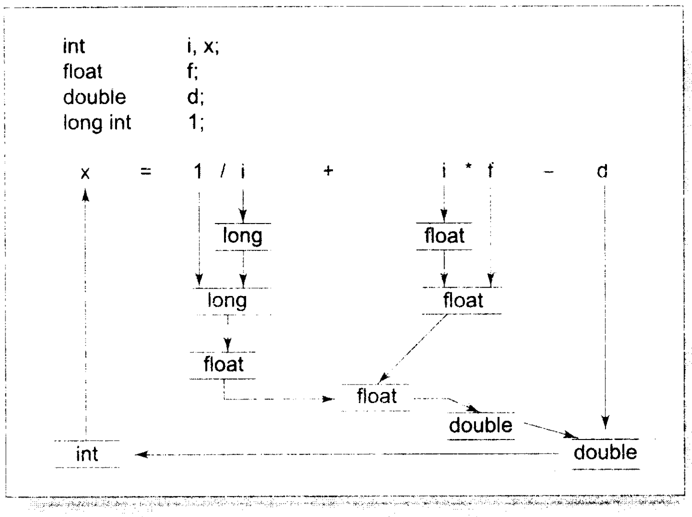


Fig. 3.7 *Process of implicit type conversion*

Given below is the sequence of rules that are applied while evaluating expressions.

All **short** and **char** are automatically converted to **int**; then

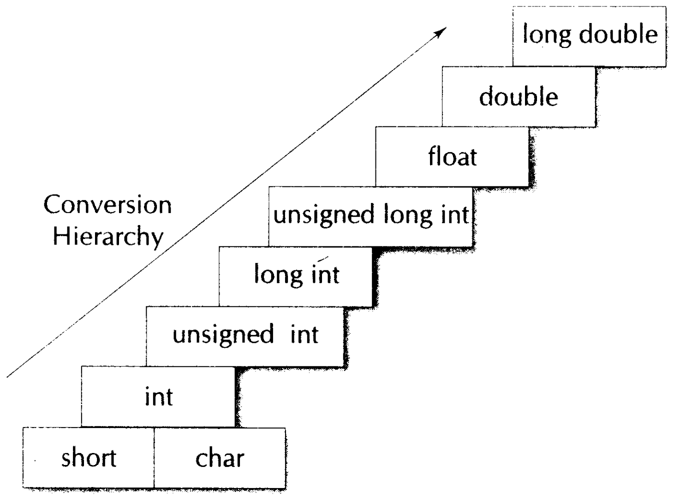
1. if one of the operands is **long double,** the other will be converted to **long double** and the result will be **long double**;
2. else, if one of the operands is **double,** the other will be converted to **double** and the result will be **double**;
3. else, if one of the operands is **float,** the other will be converted to **float** and the result will be **float**;
4. else, if one of the operands is **unsigned long int**, the other will be converted to **unsigned long int** and the result will be **unsigned long int**;
5. else, if one of the operands is **long int** and the other is **unsigned int,** then
   (a) if **unsigned int** can be converted to **long int**, the **unsigned int** operand will be converted as such and the result will be **long int**;

   (b) else, both operands will be converted to **unsigned long int** and the result will be **unsigned long int**;

6. else, if one of the operands is **long int,** the other will be converted to **long int** and the result will be **long int**;
7. else, if one of the operands is **unsigned int,** the other will be converted to **unsigned int** and the result will be **unsigned int.**

Conversion Hierarchy

Note that, C uses the rule that, in all expressions except assignments, any implicit type conversions are made from a lower size type to a higher size type as shown below:



Note that some versions of C automatically convert all floating-point operands to double precision.

The final result of an expression is converted to the type of the variable on the left of the assignment sign before assigning the value to it. However, the following changes are introduced during the final assignment.

1. **float** to **int** causes truncation of the fractional part.
2. **double** to **float** causes rounding of digits.
3. **long int** to **int** causes dropping of the excess higher order bits.

Explicit Conversion

We have just discussed how C performs type conversion automatically. However, there are instances when we want to force a type conversion in a way that is different from the automatic conversion. Consider, for example, the calculation of ratio of females to males in a town.

ratio = female_number/male_number

Since **female_number** and **male_number** are declared as integers in the program, the decimal part of the result of the division would be lost and **ratio** would represent a wrong figure. This problem can be solved by converting locally one of the variables to the floating point as shown below:

ratio = (**float**) female_number/male_number

The operator (**float**) converts the **female_number** to floating point for the purpose of evaluation of the expression. Then using the rule of automatic conversion, the division is performed in floating point mode, thus retaining the fractional part of result.

Note that in no way does the operator (**float**) affect the value of the variable **female number.** And also, the type of **female number** remains as **int** in the other parts of the program.

The process of such a local conversion is known as *explicit conversion* or *casting a value*. The general form of a cast is:

(type-name)expression

Where *type-name* is one of the standard C data types. The expression may be a constant, variable or an expression. Some examples of casts and their actions are shown in Table 3.7.

Table 3.7 *Use of Casts*

Example	*Action*
x = (**int**) 7.5	7.5 is converted to integer by truncation.
a = (**int**) 21.3/(int)4.5	Evaluated as 21/4 and the result would be 5.
b = (**double**)sum/n	Division is done in floating point mode.
y = (**int**) (a+b)	The result of a+b is converted to integer.
z = (**int**)a+b	a is converted to integer and then added to b.
p = cos((**double**)x)	Converts x to double before using it.

Casting can be used to round-off a given value. Consider the following statement:

```
x = (int) (y+0.5);
```

If **y** is 27.6, **y**+0.5 is 28.1 and on casting, the result becomes 28, the value that is assigned to **x**. Of course, the expression, being cast is not changed.

Example 3.6 Figure 3.8 shows a program using a cast to evaluate the equation

$$\text{sum} = \sum_{i=1}^{n} (1/i)$$

Program

```
main()
{
    float  sum ;
    int    n ;

    sum = 0 ;
```

```
            for( n = 1 ; n <= 10 ; ++n )
            {
              sum = sum + 1/(float)n ;
              printf("%2d %6.4f\n", n, sum) ;
            }
      }
      Output

          1  1.0000
          2  1.5000
          3  1.8333
          4  2.0833
          5  2.2833
          6  2.4500
          7  2.5929
          8  2.7179
          9  2.8290
         10  2.9290
```

Fig. 3.8 *Use of a cast*

3.15 OPERATOR PRECEDENCE AND ASSOCIATIVITY

Each operator in C has a precedence associated with it. This precedence is used to determine how an expression involving more than one operator is evaluated. There are distinct *levels of precedence* and an operator may belong to one of these levels. The operators at the higher level of precedence are evaluated first. The operators of the same precedence are evaluated either from 'left to right' or from 'right to left', depending on the level. This is known as the *associativity* property of an operator. Table 3.8 provides a complete list of operators, their precedence levels, and their rules of association. The groups are listed in the order of decreasing precedence. Rank 1 indicates the highest precedence level and 15 the lowest. The list also includes those operators, which we have not yet been discussed.

It is very important to note carefully, the order of precedence and associativity of operators. Consider the following conditional statement:

if (x == 10 + 15 && y < 10)

The precedence rules say that the ***addition*** operator has a higher priority than the logical operator (&&) and the relational operators (== and <). Therefore, the addition of 10 and 15 is executed first. This is equivalent to :

if (x == 25 && y < 10)

The next step is to determine whether **x** is equal to 25 and **y** is less than 10. If we assume a value of 20 for x and 5 for y, then

x == 25 is FALSE (0)

y < 10 is TRUE (1)

Note that since the operator < enjoys a higher priority compared to ==, y < 10 is tested first and then x == 25 is tested.

Finally we get:

if (FALSE && TRUE)

Because one of the conditions is FALSE, the complex condition is FALSE.

In the case of **&&**, it is guaranteed that the second operand will not be evaluated if the first is zero and in the case of \|\|, the second operand will not be evaluated if the first is non-zero.

Table 3.8 *Summary of C Operators*

Operator	*Description*	*Associativity*	*Rank*
( )	Function call	Left to right	1
[ ]	Aray element reference		
+	Unary plus		
–	Unary minus	Right to left	2
++	Increment		
– –	Decrement		
!	Logical negation		
~	Ones complement		
*	Pointer reference (indirection)		
&	Address		
sizeof	Size of an object		
(type)	Type cast (conversion)		
*	Multiplication	Left to right	3
/	Division		
%	Modulus		
+	Addition	Left to right	4
–	Subtraction		
<<	Left shift	Left to right	5
>>	Right shift		
<	Less than	Left to right	6
<=	Less than or equal to		
>	Greater than		
>=	Greater than or equal to		
==	Equality	Left to right	7
!=	Inequality		
&	Bitwise AND	Left to right	8
^	Bitwise XOR	Left to right	9
|	Bitwise OR	Left to right	10
&&	Logical AND	Left to right	11
||	Logical OR	Left to right	12
?:	Conditional expression	Right to left	13
= *= /= %= += -= &= ^= |= <<= >>=	Assignment operators	Right to left	14
,	Comma operator	Left to right	15

Rules of Precedence and Associativity

- Precedence rules decides the order in which different operators are applied
- Associativity rule decides the order in which multiple occurrences of the same level operator are applied

3.16 MATHEMATICAL FUNCTIONS

Mathematical functions such as cos, sqrt, log, etc. are frequently used in analysis of real-life problems. Most of the C compilers support these basic math functions. However, there are systems that have a more comprehensive math library and one should consult the reference manual to find out which functions are available. Table 3.9 lists some standard math functions.

Table 3.9 *Math functions*

Function	*Meaning*
Trigonometric	
acos(x)	Arc cosine of x
asin(x)	Arc sine of x
atan(x)	Arc tangent of x
atan 2(x,y)	Arc tangent of x/y
cos(x)	Cosine of x
sin(x)	Sine of x
tan(x)	Tangent of x
Hyperbolic	
cosh(x)	Hyperbolic cosine of x
sinh(x)	Hyperbolic sine of x
tanh(x)	Hyperbolic tangent of x
Other functions	
ceil(x)	x rounded up to the nearest integer
exp(x)	e to the x power (e^x)
fabs(x)	Absolute value of x.
floor(x)	x rounded down to the nearest integer
fmod(x,y)	Remainder of x/y
log(x)	Natural log of x, x > 0
log10(x)	Base 10 log of x, x > 0
pow(x,y)	x to the power y (x^y)
sqrt(x)	Square root of x, x > = 0

Note:
1. **x** and **y** should be declared as **double.**
2. In trigonometric and hyperbolic functions, **x** and **y** are in radians.
3. All the functions return a **double**.

As pointed out earlier in Chapter 1, to use any of these functions in a program, we should include the line:

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

in the beginning of the program.

Just Remember

- ✍ Use *decrement* and *increment* operators carefully. Understand the difference between **postfix** and **prefix** operations before using them.
- ✍ Add parentheses wherever you feel they would help to make the evaluation order clear.
- ✍ Be aware of side effects produced by some expressions.
- ✍ Avoid any attempt to divide by zero. It is normally undefined. It will either result in a fatal error or in incorrect results.
- ✍ Do not forget a semicolon at the end of an expression.
- ✍ Understand clearly the precedence of operators in an expression. Use parentheses, if necessary.
- ✍ Associativity is applied when more than one operator of the same precedence are used in an expression. Understand which operators associate from right to left and which associate from left to right.
- ✍ Do not use *increment* or *decrement* operators with any expression other than a *variable identifier*.
- ✍ It is illegal to apply modules operator % with anything other than integers.
- ✍ Do not use a variable in an expression before it has been assigned a value.
- ✍ Integer division always truncates the decimal part of the result. Use it carefully. Use casting where necessary.
- ✍ The result of an expression is converted to the type of the variable on the left of the assignment before assigning the value to it. Be careful about the loss of information during the conversion.
- ✍ All mathematical functions implement *double* type parameters and return *double* type values.
- ✍ It is an error if any space appears between the two symbols of the operators ==, !=, <= and >=.
- ✍ It is an error if the two symbols of the operators !=, <= and >= are reversed.
- ✍ Use spaces on either side of binary operator to improve the readability of the code.
- ✍ Do not use increment and decrement operators to floating point variables.
- ✍ Do not confuse the equality operator == with the assignment operator =.

CASE STUDIES

1. Salesman's Salary

A computer manufacturing company has the following monthly compensation policy to their salespersons:

Minimum base salary	:	1500.00

Bonus for every computer sold : 200.00
Commission on the total monthly sales : 2 per cent

Since the prices of computers are changing, the sales price of each computer is fixed at the beginning of every month. A program to compute a sales-person's gross salary is given in Fig. 3.9.

Program

```
#define BASE_SALARY 1500.00
#define BONUS_RATE   200.00
#define COMMISSION     0.02

main()
{
    int quantity ;
    float gross_salary, price ;
    float bonus, commission ;

    printf("Input number sold and price\n") ;
    scanf("%d %f", &quantity, &price) ;

    bonus        = BONUS_RATE * quantity ;
    commission   = COMMISSION * quantity * price ;
    gross_salary = BASE_SALARY + bonus + commission ;

    printf("\n");
    printf("Bonus        = %6.2f\n", bonus) ;
    printf("Commission   = %6.2f\n", commission) ;
    printf("Gross salary = %6.2f\n", gross_salary) ;
}
```

Output

```
Input number sold and price
5 20450.00
Bonus           = 1000.00
Commission      = 2045.00
Gross salary    = 4545.00
```

Fig. 3.9 *Program of salesman's salary*

Given the base salary, bonus, and commission rate, the inputs necessary to calculate the gross salary are, the price of each computer and the number sold during the month.

The gross salary is given by the equation:

Gross salary = base salary + (quantity * bonus rate)
+ (quantity * Price) * commission rate

2. Solution of the quadratic equation

An equation of the form

$$ax^2 + bx + c = 0$$

is known as the *quadratic equation*. The values of x that satisfy the equation are known as the *roots* of the equation. A quadratic equation has two roots which are given by the following two formulae:

$$\text{root 1} = \frac{-b + \text{sqrt}(b^2 - 4ac)}{2a}$$

$$\text{root 2} = \frac{-b - \text{sqrt}(b^2 - 4ac)}{2a}$$

A program to evaluate these roots is given in Fig. 3.10. The program requests the user to input the values of **a**, **b** and **c** and outputs **root 1** and **root 2**.

Program
```
#include <math.h>

main()
{
     float a, b, c, discriminant,
            root1, root2;

     printf("Input values of a, b, and c\n");
     scanf("%f %f %f", &a, &b, &c);

     discriminant = b*b - 4*a*c ;
     if(discriminant < 0)
       printf("\n\nROOTS ARE IMAGINARY\n");
     else
     {
       root1 = (-b + sqrt(discriminant))/(2.0*a);
       root2 = (-b - sqrt(discriminant))/(2.0*a);
       printf("\n\nRoot1 = %5.2f\n\nRoot2 = %5.2f\n",
                  root1,root2 );
     }
}
```
Output
```
Input values of a, b, and c
2 4 -16

Root1 = 2.00

Root2 = -4.00

Input values of a, b, and c
1 2 3
```

```
                    ROOTS ARE IMAGINARY
```

Fig. 3.10 *Solution of a quadratic equation*

The term (b^2–4ac) is called the *discriminant*. If the discriminant is less than zero, its square roots cannot be evaluated. In such cases, the roots are said to be imaginary numbers and the program outputs an appropriate message.

REVIEW QUESTIONS

3.1 State whether the following statements are *true* or *false*.
- (a) All arithmetic operators have the same level of precedence.
- (b) The modulus operator % can be used only with integers.
- (c) The operators <=, >= and != all enjoy the same level of priority.
- (d) During modulo division, the sign of the result is positive, if both the operands are of the same sign.
- (e) In C, if a data item is zero, it is considered false.
- (f) The expression *!(x<=y)* is same as the expression *x>y*.
- (g) A unary expression consists of only one operand with no operators.
- (h) Associativity is used to decide which of several different expressions is evaluated first.
- (i) An expression statement is terminated with a period.
- (j) During the evaluation of mixed expressions, an implicit cast is generated automatically.
- (k) An explicit cast can be used to change the expression.
- (l) Parentheses can be used to change the order of evaluation expressions.

3.2 Fill in the blanks with appropriate words.
- (a) The expression containing all the integer operands is called_________ expression.
- (b) The operator __________cannot be used with real operands.
- (c) C supports as many as ________relational operators.
- (d) An expression that combines two or more relational expressions is termed as ___________expression.
- (e) The ____________operator returns the number of bytes the operand occupies.
- (f) The order of evaluation can be changed by using _______ in an expression.
- (g) The use of_________ on a variable can change its type in the memory.
- (h) __________is used to determine the order in which different operators in an expression are evaluated.

3.3 Given the statement

int a = 10, b = 20, c;

determine whether each of the following statements are true or false.
- (a) The statement a = + 10, is valid.
- (b) The expression a + 4/6 * 6/2 evaluates to 11.
- (c) The expression b + 3/2 * 2/3 evaluates to 20.
- (d) The statement a + = b; gives the values 30 to a and 20 to b.

(e) The statement ++a++; gives the value 12 to a

(f) The statement a = 1/b; assigns the value 0.5 to a

3.4 Declared **a** as *int* and **b** as *float*, state whether the following statements are true or false.

(a) The statement a = 1/3 + 1/3 + 1/3; assigns the value 1 to a.

(b) The statement b = 1.0/3.0 + 1.0/3.0 + 1.0/3.0; assigns a value 1.0 to b.

(c) The statement b = 1.0/3.0 * 3.0 gives a value 1.0 to b.

(d) The statement b = 1.0/3.0 + 2.0/3.0 assigns a value 1.0 to b.

(e) The statement a = 15/10.0 + 3/2; assigns a value 3 to a.

3.5 Which of the following expressions are true?

(a) !(5 + 5 >=10)

(b) 5 + 5 = = 10 || 1 + 3 = = 5

(c) 5 > 10 || 10 < 20 && 3 < 5

(d) 10 ! = 15 && !(10<20) || 15 > 30

3.6 Which of the following arithmetic expressions are valid ? If valid, give the value of the expression; otherwise give reason.

(a) 25/3 % 2	(e) –14 % 3
(b) +9/4 + 5	(f) 15.25 + – 5.0
(c) 7.5 % 3	(g) (5/3) * 3 + 5 % 3
(d) 14 % 3 + 7 % 2	(h) 21 % (int)4.5

3.7 Write C assignment statements to evaluate the following equations:

(a) $\text{Area} = \pi r^2 + 2\pi rh$

(b) $\text{Torque} = \dfrac{2m_1m_2}{m_1 + m_2} \cdot g$

(c) $\text{Side} = \sqrt{a^2 + b^2 - 2ab\cos(x)}$

(d) $\text{Energy} = \text{mass}\left[\text{acceleration} \times \text{height} + \dfrac{(\text{velocity})^2}{2}\right]$

3.8 Identify unnecessary parentheses in the following arithmetic expressions.

(a) ((x–(y/5)+z)%8) + 25

(b) ((x–y) * p)+q

(c) (m*n) + (–x/y)

(d) x/(3*y)

3.9 Find errors, if any, in the following assignment statements and rectify them.

(a) `x = y = z = 0.5, 2.0. -5.75;`

(b) `m = ++a * 5;`

(c) `y = sqrt(100);`

(d) `p * = x/y;`

(e) `s = /5;`

(f) `a = b++ -c*2`

3.10 Determine the value of each of the following logical expressions if a = 5, b = 10 and c = –6

(a) a > b && a < c

(b) a < b && a > c

(c) a == c || b > a

(d) b > 15 && c < 0 || a > 0
(e) (a/2.0 == 0.0 && b/2.0 != 0.0) || c < 0.0

PROGRAMMING EXERCISES

3.1 Given the values of the variables x, y and z, write a program to rotate their values such that x has the value of y, y has the value of z, and z has the value of x.

3.2 Write a program that reads a floating-point number and then displays the right-most digit of the integral part of the number.

3.3 Modify the above program to display the two right-most digits of the integral part of the number.

3.4 Write a program that will obtain the length and width of a rectangle from the user and compute its area and perimeter.

3.5 Given an integer number, write a program that displays the number as follows:

First line	:	all digits
Second line	:	all except first digit
Third line	:	all except first two digits
....		
Last line	:	The last digit

For example, the number 5678 will be displayed as:

```
5 6 7 8
6 7 8
7 8
8
```

3.6 The straight-line method of computing the yearly depreciation of the value of an item is given by

$$\text{Depreciation} = \frac{\text{Purchase Price} - \text{Salvage Value}}{\text{Years of Service}}$$

Write a program to determine the salvage value of an item when the purchase price, years of service, and the annual depreciation are given.

3.7 Write a program that will read a real number from the keyboard and print the following output in one line:

```
Smallest integer     The given     Largest integer
not less than        number        not greater than
the number                         the number
```

3.8 The total distance travelled by a vehicle in t seconds is given by

$$\text{distance} = ut + (at^2)/2$$

Where u is the initial velocity (metres per second), a is the acceleration (metres per second^2). Write a program to evaluate the distance travelled at regular intervals of time, given the values of u and a. The program should provide the flexibility to the user to select his own time intervals and repeat the calculations for different values of u and a.

3.9 In inventory management, the Economic Order Quantity for a single item is given by

$$\text{EOQ} = \sqrt{\frac{2 \times \text{demand rate} \times \text{setup costs}}{\text{holding cost per item per unit time}}}$$

and the optimal Time Between Orders

$$\text{TBO} = \sqrt{\frac{2 \times \text{setup costs}}{\text{demand rate} \times \text{holding cost per item per unit time}}}$$

Write a program to compute EOQ and TBO, given demand rate (items per unit time), setup costs (per order), and the holding cost (per item per unit time).

3.10 For a certain electrical circuit with an inductance L and resistance R, the damped natural frequency is given by

$$\text{Frequency} = \sqrt{\frac{1}{LC} - \frac{R^2}{4C^2}}$$

It is desired to study the variation of this frequency with C (capacitance). Write a program to calculate the frequency for different values of C starting from 0.01 to 0.1 in steps of 0.01.

Chapter 4

Managing Input and Output Operations

4.1 INTRODUCTION

Reading, processing, and writing of data are the three essential functions of a computer program. Most programs take some data as input and display the processed data, often known as *information* or *results*, on a suitable medium. So far we have seen two methods of providing data to the program variables. One method is to assign values to variables through the assignment statements such as **x = 5; a = 0;** and so on. Another method is to use the input function **scanf** which can read data from a keyboard. We have used both the methods in most of our earlier example programs. For outputting results we have used extensively the function **printf** which sends results out to a terminal.

Unlike other high-level languages, C does not have any built-in input/output statements as part of its syntax. All input/output operations are carried out through function calls such as **printf** and **scanf**. There exist several functions that have more or less become standard for input and output operations in C. These functions are collectively known as the standard I/O library. In this chapter we shall discuss some common I/O functions that can be used on many machines without any change. However, one should consult the system reference manual for exact details of these functions and also to see what other functions are available.

It may be recalled that we have included a statement

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

in the Sample Program 5 in Chapter 1, where a math library function cos(x) has been used. This is to instruct the compiler to fetch the function cos(x) from the math library, and that it is not a part of C language. Similarly, each program that uses a standard input/output function must contain the statement

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

at the beginning. However, there might be exceptions. For example, this is not necessary for the functions **printf** and **scanf** which have been defined as a part of the C language.

The file name **stdio.h** is an abbrevation for *standard input-output header* file. The instruction **#include** <*stdio.h*> tells the compiler 'to search for a file named **stdio.h** and place its contents at this point in the program'. The contents of the header file become part of the source code when it is compiled.

4.2 READING A CHARACTER

The simplest of all input/output operations is reading a character from the 'standard input' unit (usually the keyboard) and writing it to the 'standard output' unit (usually the screen). Reading a single character can be done by using the function **getchar.** (This can also be done with the help of the **scanf** function which is discussed in Section 4.4.) The **getchar** takes the following form:

variable_name = getchar( );

variable_name is a valid C name that has been declared as **char** type. When this statement is encountered, the computer waits until a key is pressed and then assigns this character as a value to **getchar** function. Since **getchar** is used on the right-hand side of an assignment statement, the character value of **getchar** is in turn assigned to the variable name on the left. For example

```
char name;
name = getchar();
```

will assign the character 'H' to the variable **name** when we press the key H on the keyboard. Since **getchar** is a function, it requires a set of parentheses as shown.

Example 4.1 The program in Fig. 4.1 shows the use of **getchar** function in an interactive environment.

The program displays a question of YES/NO type to the user and reads the user's response in a single character (Y or N). If the response is Y, it outputs the message

My name is BUSY BEE

otherwise, outputs.

You are good for nothing

Note: There is one line space between the input text and output message.

```
Program
     #include <stdio.h>
     main()
     {
       char answer;
       printf("Would you like to know my name?\n");

       printf("Type Y for YES and N for NO: ");
       answer = getchar(); /* .... Reading a character...*/
       if(answer == 'Y' || answer == 'y')
         printf("\n\nMy name is BUSY BEE\n");
       else
```

```
            printf("\n\nYou are good for nothing\n");
        }
Output
    Would you like to know my name?
    Type Y for YES and N for NO: Y
    My name is BUSY BEE
    Would you like to know my name?
    Type Y for YES and N for NO: n
    You are good for nothing
```

Fig. 4.1 *Use of* ***getchar*** *function to read a* ***character*** *from keyboard*

The **getchar** function may be called successively to read the characters contained in a line of text. For example, the following program segment reads characters from keyboard one after another until the 'Return' key is pressed.

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

---------

char character;
character = ' ';
while(character != '\n')
{
        character = getchar();
}
---------

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

WARNING

The **getchar()** function accepts any character keyed in. This includes RETURN and TAB. This means when we enter single character input, the newline character is waiting in the input queue after **getchar()** returns. This could create problems when we use **getchar()** in a loop interactively. A dummy **getchar()** may be used to 'eat' the unwanted newline character. We can also use the **fflush** function to flush out the unwanted characters.

Note: We shall be using decision statements like **if**, **if...else** and **while** extensively in this chapter. They are discussed in detail in Chapters 5 and 6.

Example 4.2 The program of Fig. 4.2 requests the user to enter a character and displays a message on the screen telling the user whether the character is an alphabet or digit, or any other special character.

This program receives a character from the keyboard and tests whether it is a letter or digit and prints out a message accordingly. These tests are done with the help of the following functions:

```
isalpha(character)
isdigit(character)
```

For example. **isalpha** assumes a value non-zero (TRUE) if the argument **character** contains an alphabet; otherwise it assumes 0 (FALSE). Similar is the case with the function **isdigit**.

Program:

```
    #include <stdio.h>
    #include <ctype.h>
    main()
    {
      char character;
      printf("Press any key\n");
      character = getchar();
      if (isalpha(character) > 0)
        printf("The character is a letter.");
      else
        if (isdigit (character) > 0)
          printf("The character is a digit.");
        else
          printf("The character is not alphanumeric.");
    }
```

Output

```
      Press any key
      h
      The character is a letter.
      Press any key
      5
      The character is a digit.
      Press any key
      *
      The character is not alphanumeric.
```

Fig. 4.2 *Program to test the **character** type*

C supports many other similar functions, which are given in Table 4.1. These character functions are contained in the file **ctype.h** and therefore the statement

```
#include <ctype.h>
```

must be included in the program.

Table 4.1 *Character Test Functions*

Function	*Test*
isalnum(c)	Is c an alphanumeric character?
isalpha(c)	Is c an alphabetic character?
isdigit(c)	Is c a digit?
islower(c)	Is c lower case letter?
isprint(c)	Is c a printable character?

(Contd.)

Table 4.1 *(Contd.)*

Function	*Test*
ispunct(c)	Is c a punctuation mark?
isspace(c)	Is c a white space character?
isupper(c)	Is c an upper case letter?

4.3 WRITING A CHARACTER

Like **getchar**, there is an analogous function **putchar** for writing characters one at a time to the terminal. It takes the form as shown below:

putchar (*variable_name*);

where ***variable_name*** is a type **char** variable containing a character. This statement displays the character contained in the ***variable_name*** at the terminal. For example, the statements

```
answer = 'Y';
putchar (answer);
```

will display the character Y on the screen. The statement

```
putchar ('\n');
```

would cause the cursor on the screen to move to the beginning of the next line.

Example 4.3 A program that reads a character from keyboard and then prints it in reverse case is given in Fig. 4.3. That is, if the input is upper case, the output will be lower case and vice versa.

The program uses three new functions: **islower, toupper**, and **tolower**. The function **islower** is a conditional function and takes the value TRUE if the argument is a lowercase alphabet; otherwise takes the value FALSE. The function **toupper** converts the lowercase argument into an uppercase alphabet while the function **tolower** does the reverse.

```
Program
    #include <stdio.h>
    #include <ctype.h>
    main()
    {
      char alphabet;
      printf("Enter an alphabet");
      putchar('\n'); /* move to next line */
      alphabet = getchar();
      if (islower(alphabet))
        putchar(toupper(alphabet));
      else
        putchar(tolower(alphabet));
    }
Output
        Enter an alphabet
```

```
a
A
Enter an alphabet
Q
q
Enter an alphabet
z
Z
```

Fig. 4.3 *Reading and writing of alphabets in reverse case*

4.4 FORMATTED INPUT

Formatted input refers to an input data that has been arranged in a particular format. For example, consider the following data:

15.75 123 John

This line contains three pieces of data, arranged in a particular form. Such data has to be read conforming to the format of its appearance. For example, the first part of the data should be read into a variable **float**, the second into **int**, and the third part into **char**. This is possible in C using the **scanf** function. (**scanf** means *scan* formatted.)

We have already used this input function in a number of examples. Here, we shall explore all of the options that are available for reading the formatted data with **scanf** function. The general form of **scanf** is

scanf("*control string*", *arg1, arg2,* ...... *argn*);

The *control string* specifies the field format in which the data is to be entered and the arguments *arg*1, *arg2,* ....., *argn* specify the address of locations where the data is stored. Control string and arguments are separated by commas.

Control string (also known as *format string*) contains field specifications, which direct the interpretation of input data. It may include:

- Field (or format) specifications, consisting of the conversion character %, a data type character (or type specifier), and an *optional* number, specifying the field width.
- Blanks, tabs, or newlines.

Blanks, tabs and newlines are ignored. The data type character indicates the type of data that is to be assigned to the variable associated with the corresponding argument. The field width specifier is optional. The discussions that follow will clarify these concepts.

Inputting Integer Numbers

The field specification for reading an integer number is:

% *w* d

The percent sign (%) indicates that a conversion specification follows. *w* is an integer number that specifies the *field width* of the number to be read and d, known as data type character, indicates that the number to be read is in integer mode. Consider the following example:

```
scanf ("%2d %5d", &numl, &num2);
```

Data line:

```
50  31426
```

The value 50 is assigned to **numl** and 31426 to **num2**. Suppose the input data is as follows:

```
31426 50
```

The variable **numl** will be assigned 31 (because of %2d) and **num2** will be assigned 426 (unread part of 31426). The value 50 that is unread will be assigned to the first variable in the next **scanf** call. This kind of errors may be eliminated if we use the field specifications without the field width specifications. That is, the statement

```
scanf("%d %d", &numl, &num2);
```

will read the data

```
31426 50
```

correctly and assign 31426 to **numl** and 50 to **num2.**

Input data items must be separated by spaces, tabs or newlines. Punctuation marks do not count as separators. When the **scanf** function searches the input data line for a value to be read, it will always bypass any white space characters.

What happens if we enter a floating point number instead of an integer? The fractional part may be stripped away! Also, **scanf** may skip reading further input.

When the **scanf** reads a particular value, reading of the value will be terminated as soon as the number of characters specified by the field width is reached (if specified) or until a character that is not valid for the value being read is encountered. In the case of integers, valid characters are an optionally signed sequence of digits.

An input field may be skipped by specifying * in the place of field width. For example, the statement

```
scanf("%d %*d %d", &a, &b)
```

will assign the data

```
123 456 789
```

as follows:

```
123 to a
456 skipped (because of *)
789 to b
```

The data type character **d** may be preceded by '**l**' (letter ell) to read long integers and **h** to read short integers.

Note: We have provided white space between the field specifications. These spaces are not necessary with the numeric input, but it is a good practice to include them.

Example 4.4 Various input formatting options for reading integers are experimented in the program shown in Fig. 4.4.

Program

```
    main()
    {
      int a,b,c,x,y,z;
      int p,q,r;
      printf("Enter three integer numbers\n");
      scanf("%d %*d %d",&a,&b,&c);
      printf("%d %d %d \n\n",a,b,c);
      printf("Enter two 4-digit numbers\n");
      scanf("%2d %4d",&x,&y);
      printf("%d %d\n\n", x,y);
      printf("Enter two integers\n");
      scanf("%d %d", &a,&x);
      printf("%d %d \n\n",a,x);
      printf("Enter a nine digit number\n");
      scanf("%3d %4d %3d",&p,&q,&r);
      printf("%d %d %d \n\n",p,q,r);
      printf("Enter two three digit numbers\n");
      scanf("%d %d",&x,&y);
      printf("%d %d",x,y);
    }
```

Output

```
      Enter three integer numbers
      1 2 3
      1 3 -3577
      Enter two 4-digit numbers
      6789 4321
      67 89
      Enter two integers
      44 66
      4321 44
    Enter a nine-digit number
    123456789
    66 1234 567
    Enter two three-digit numbers
    123 456
    89 123
```

Fig. 4.4 *Reading integers using* ***scanf***

The first **scanf** requests input data for three integer values **a, b,** and **c,** and accordingly three values 1, 2, and 3 are keyed in. Because of the specification %*d the value 2 has been skipped and 3 is assigned to the variable **b**. Notice that since no data is available for **c**, it contains garbage.

The second **scanf** specifies the format %2d and %4d for the variables **x** and **y** respectively. Whenever we specify field width for reading integer numbers, the input numbers should not contain more digits that the specified size. Otherwise, the extra digits on the right-hand side will be truncated and assigned to the next variable in the list. Thus, the second **scanf** has truncated the four digit number 6789 and assigned 67 to **x** and 89 to **y**. The value 4321 has been assigned to the first variable in the immediately following **scanf** statement.

Note: It is legal to use a non-whitespace character between field specifications. However, the **scanf** expects a matching character in the given location. For example,

```
scanf"%d-%d", &a, &b);
```

accepts input like

```
123-456
```

to assign 123 to **a** and 456 to **b**.

Inputting Real Numbers

Unlike integer numbers, the field width of real numbers is not to be specified and therefore **scanf** reads real numbers using the simple specification **%f** for both the notations, namely, decimal point notation and exponential notation. For example, the statement

```
scanf("%f %f %f", &x, &y, &z);
```

with the input data

```
475.89 43.21E-1 678
```

will assign the value 475.89 to **x**, 4.321 to **y**, and 678.0 to **z**. The input field specifications may be separated by any arbitrary blank spaces.

If the number to be read is of **double** type, then the specification should be **%lf** instead of simple **%f.** A number may be skipped using **%*f** specification.

Example 4.5 Reading of real numbers (in both decimal point and exponential notation) is illustrated in Fig. 4.5.

```
Program
    main()
    {
      float x,y;
      double p,q;

      printf("Values of x and y:");
      scanf("%f %e", &x, &y);
      printf("\n");
      printf("x = %f\ny = %f\n\n", x, y);
```

```
        printf("Values of p and q:");
        scanf("%lf %lf", &p, &q);
        printf("\n\np = %.12lf\np = %.12e", p,q);
    }
Output
        Values of x and y:12.3456 17.5e-2
        x = 12.345600
        y = 0.175000

        Values of p and q:4.142857142857 18.5678901234567890

        p = 4.142857142857
        q = 1.856789012346e+001
```

Fig. 4.5 *Reading of real numbers*

Inputting Character Strings

We have already seen how a single character can be read from the terminal using the **getchar** function. The same can be achieved using the **scanf** function also. In addition, a **scanf** function can input strings containing more than one character. Following are the specifications for reading character strings:

%ws or %wc

The corresponding argument should be a pointer to a character array. However, %c may be used to read a single character when the argument is a pointer to a **char** variable.

Example 4.6 Reading of strings using **%wc** and **%ws** is illustrated in Fig. 4.6.

The program in Fig. 4.6 illustrates the use of various field specifications for reading strings. When we use %wc for reading a string, the system will wait until the w^{th} character is keyed in.

Note that the specification **%s** terminates reading at the encounter of a blank space. Therefore, **name2** has read only the first part of "New York" and the second part is automatically assigned to **name3**. However, during the second run, the string "New-York" is correctly assigned to **name2**.

```
Program
    main()
    {
      int no;
      char namel[15], name2[15], name3[15];

      printf("Enter serial number and name one\n");
      scanf("%d %15c", &no, namel);
      printf("%d %15s\n\n", no, namel);
```

```
        printf("Enter serial number and name two\n");
        scanf("%d %s", &no, name2);
        printf("%d %15s\n\n", no, name2);
        printf("Enter serial number and name three\n");
        scanf("%d %15s", &no, name3);
        printf("%d %15s\n\n", no, name3);
    }
Output
        Enter serial number and name one
        1 123456789012345
        1 123456789012345r
        Enter serial number and name two
        2 New York
        2           New
        Enter serial number and name three
        2           York
        Enter serial number and name one
        1 123456789012
        1 123456789012r
        Enter serial number and name two
        2 New-York
        2      New-York
        Enter serial number and name three
        3 London
        3      London
```

Fig. 4.6 *Reading of strings*

Some versions of **scanf** support the following conversion specifications for strings:

%[characters]

%[^characters]

The specification **%[characters]** means that only the characters specified within the brackets are permissible in the input string. If the input string contains any other character, the string will be terminated at the first encounter of such a character. The specification **%[^characters]** does exactly the reverse. That is, the characters specified after the circumflex (^) are not permitted in the input string. The reading of the string will be terminated at the encounter of one of these characters.

Example 4.7 The program in Fig. 4.7 illustrates the function of %[ ] specification.

```
Program-A
    main()
    {
      char address[80];
```

```
        printf("Enter address\n");
        scanf("%[a-z]", address);
        printf("%-80s\n\n", address);
    }
Output
        Enter address
        new delhi 110002
        new delhi
```

```
Program-B
    main()
    {
        char address[80];

        printf("Enter address\n");
        scanf("%[^\n]", address);
        printf("%-80s", address);
    }
Output
        Enter address
        New Delhi 110 002
        New Delhi 110 002
```

Fig. 4.7 *Illustration of conversion specification%[] for strings*

Reading Blank Spaces

We have earlier seen that %s specifier cannot be used to read strings with blank spaces. But, this can be done with the help of %[ ] specification. Blank spaces may be included within the brackets, thus enabling the **scanf** to read strings with spaces. Remember that the lowercase and uppercase letters are distinct. See Fig. 4.7.

Reading Mixed Data Types

It is possible to use one **scanf** statement to input a data line containing mixed mode data. In such cases, care should be exercised to ensure that the input data items match the control specifications *in order* and *type*. When an attempt is made to read an item that does not match the type expected, the **scanf** function does not read any further and immediately returns the values read. The statement

```
scanf ("%d %c %f %s", &count, &code, &ratio, name);
```

will read the data

```
15 p 1.575 coffee
```

correctly and assign the values to the variables in the order in which they appear. Some systems accept integers in the place of real numbers and vice versa, and the input data is converted to the type specified in the control string.

Note: A space before the *%c* specification in the format string is necessary to skip the white space before p.

Detection of Errors in Input

When a **scanf** function completes reading its list, it returns the value of number of items that are successfully read. This value can be used to test whether any errors occurred in reading the input. For example, the statement

```
scanf("%d %f %s, &a, &b, name);
```

will return the value 3 if the following data is typed in:

```
20 150.25 motor
```

and will return the value 1 if the following line is entered

```
20 motor 150.25
```

This is because the function would encounter a string when it was expecting a floating-point value, and would therefore terminate its scan after reading the first value.

Example 4.8 The program presented in Fig.4.8 illustrates the testing for correctness of reading of data by **scanf** function.

The function **scanf** is expected to read three items of data and therefore, when the values for all the three variables are read correctly, the program prints out their values. During the third run, the second item does not match with the type of variable and therefore the reading is terminated and the error message is printed. Same is the case with the fourth run.

In the last run, although data items do not match the variables, no error message has been printed. When we attempt to read a real number for an **int** variable, the integer part is assigned to the variable, and the truncated decimal part is assigned to the next variable.
Note that the character '2' is assigned to the character variable c.

```
Program
    main()
    {
      int a;
      float b;
      char c;
      printf("Enter values of a, b and c\n");
      if (scanf("%d %f %c", &a, &b, &c) == 3)
        printf("a = %d b = %f c = %c\n" , a, b, c);
      else
        printf("Error in input.\n");
    }
Output
      Enter values of a, b and c
          12 3.45 A
```

```
a = 12    b = 3.450000    c = A
Enter values of a, b and c
23 78 9
a = 23    b = 78.000000    c = 9
Enter values of a, b and c
8 A 5.25
Error in input.
Enter values of a, b and c
Y 12 67
Error in input.
Enter values of a, b and c
15.75 23 X
a = 15    b = 0.750000    c = 2
```

Fig. 4.8 *Detection of errors in scanf input*

Commonly used **scanf** format codes are given in Table 4.2

Table 4.2 *Commonly used scanf Format Codes*

Code	*Meaning*
%c	read a single character
%d	read a decimal integer
%e	read a floating point value
%f	read a floating point value
%g	read a floating point value
%h	read a short integer
%i	read a decimal, hexadecimal or octal integer
%o	read an octal integer
%s	read a string
%u	read an unsigned decimal integer
%x	read a hexadecimal integer
%[..]	read a string of word(s)

The following letters may be used as prefix for certain conversion characters.

h	for short integers
l	for long integers or double
L	for long double

Points to Remember While Using scanf

If we do not plan carefully, some 'crazy' things can happen with **scanf**. Since the I/O routines are not a part of C language, they are made available either as a separate module of the C library or as a part of the operating system (like UNIX). New features are added to these routines from time to time as new versions of systems are released. We should consult the system reference manual before using these routines. Given below are some of the general points to keep in mind while writing a **scanf** statement.

1. All function arguments, except the control string, *must* be pointers to variables.
2. Format specifications contained in the control string should match the arguments in order.
3. Input data items must be separated by spaces and must match the variables receiving the input in the same order.
4. The reading will be terminated, when **scanf** encounters a 'mismatch' of data or a character that is not valid for the value being read.
5. When searching for a value, **scanf** ignores line boundaries and simply looks for the next appropriate character.
6. Any unread data items in a line will be considered as part of the data input line to the next **scanf** call.
7. When the field width specifier *w* is used, it should be large enough to contain the input data size.

Rules for scanf

- Each variable to be read must have a filed specification.
- For each field specification, there must be a variable address of proper type.
- Any non-whitespace character used in the format string must have a matching character in the user input.
- Never end the format string with whitespace. It is a fatal error!
- The scanf reads until:
  - A whitespace character is found in a numberic specification, or
  - The maximum number of characters have been read or
  - An error is detected, or
  - The end of file is reached

4.5 FORMATTED OUTPUT

We have seen the use of **printf** function for printing captions and numerical results. It is highly desirable that the outputs are produced in such a way that they are understandable and are in an easy-to-use form. It is therefore necessary for the programmer to give careful consideration to the appearance and clarity of the output produced by his program.

The **printf** statement provides certain features that can be effectively exploited to control the alignment and spacing of print-outs on the terminals. The general form of **printf** statement is

printf("*control string*", *arg1*, *arg2*, ..., *argn*);

Control string consists of three types of items:

1. Characters that will be printed on the screen as they appear.

2. Format specifications that define the output format for display of each item.
3. *Escape sequence* characters such as \n, \t, and \b.

The control string indicates how many arguments follow and what their types are. The arguments *arg1, arg2, ..., argn* are the variables whose values are formatted and printed according to the specifications of the control string. The arguments should match in number, order and type with the format specifications.

A simple format specification has the following form:

%w.p type-specifier

where *w* is an integer number that specifies the total number of columns for the output value and *p* is another integer number that specifies the number of digits to the right of the decimal point (of a real number) or the number of characters to be printed from a string. Both ***w*** and *p* are optional. Some examples of formatted **printf** statement are:

```
printf("Programming in C");
printf(" ");
printf("\n");
printf("%d", x);
printf("a = %f\n b = %f", a, b);
printf("sum = %d", 1234);
printf("\n\n");
```

printf never supplies a *newline* automatically and therefore multiple **printf** statements may be used to build one line of output. A *newline* can be introduced by the help of a newline character '\n' as shown in some of the examples above.

Output of Integer Numbers

The format specification for printing an integer number is

%wd

where ***w*** specifies the minimum field width for the output. However, if a number is greater than the specified field width, it will be printed in full, overriding the minimum specification. ***d*** specifies that the value to be printed is an integer. The number is written *right-justified* in the given field width. Leading blanks will appear as necessary. The following examples illustrate the output of the number 9876 under different formats:

Format	Output
printf("%d", 9876)	$\boxed{9}\boxed{8}\boxed{7}\boxed{6}$
printf("%6d", 9876)	$\boxed{}\boxed{}\boxed{9}\boxed{8}\boxed{7}\boxed{6}$
printf("%2d", 9876)	$\boxed{9}\boxed{8}\boxed{7}\boxed{6}$

printf("%-6d", 9876) | 9 | 8 | 7 | 6 | | |

printf("%06d", 9876) | 0 | 0 | 9 | 8 | 7 | 6 |

It is possible to force the printing to be left-*justified* by placing a *minus* sign directly after the % character, as shown in the fourth example above. It is also possible to pad with zeros the leading blanks by placing a 0 (zero) before the field width specifier as shown in the last item above. The minus (–) and zero (0) are known as *flags*.

Long integers may be printed by specifying **ld** in the place of **d** in the format specification. Similarly, we may use **hd** for printing short integers.

Example 4.9 The program in Fig. 4.9 illustrates the output of integer numbers under various formats.

Program

```
    main()
    {
        int m = 12345;
        long n = 987654;

        printf("%d\n",m);
        printf("%10d\n",m);
        printf("%010d\n",m);
        printf("%-10d\n",m);
        printf("%10ld\n",n);
        printf("%10ld\n",-n);
    }
```

Output

```
        12345
             12345
        0000012345
        12345
             987654
           - 987654
```

Fig. 4.9 *Formatted output of integers*

Output of Real Numbers

The output of a real number may be displayed in decimal notation using the following format specification:

% *w*.p f

The integer ***w*** indicates the minimum number of positions that are to be used for the display of the value and the integer ***p*** indicates the number of digits to be displayed after the decimal point (*preci-*

sion). The value, when displayed, is *rounded to p decimal places* and printed *right-justified* in the field of *w* columns. Leading blanks and trailing zeros will appear as necessary. The default precision is 6 decimal places. The negative numbers will be printed with the minus sign. The number will be displayed in the form [–] mmm-nnn.

We can also display a real number in exponential notation by using the specification

```
% w.p e
```

The display takes the form

```
[ - ] m.nnnne[ ± ]xx
```

where the length of the string of n's is specified by the precision *p*. The default precision is 6. The field width **w** should satisfy the condition.

w ≥ p+7

The value will be rounded off and printed right justified in the field of **w** columns.

Padding the leading blanks with zeros and printing with *left-justification* are also possible by using flags 0 or – before the field width specifier **w**.

The following examples illustrate the output of the number y = 98.7654 under different format specifications:

Format	*Output*
printf("%7.4f",y)	`98.7654`
printf("%7.2f",y)	`  98.77`
printf("%-7.2f",y)	`98.77  `
printf("%f",y)	`98.7654`
printf("%10.2e",y)	`  9.88e+01`
printf("%11.4e",-y)	`-9.8765e+01`
printf("%-10.2e",y)	`9.88e+01  `
printf("%e",y)	`9.876540e+01`

Some systems also support a special field specification character that lets the user define the field size at run time. This takes the following form:

```
printf("%*.*f", width, precision, number);
```

In this case, both the field width and the precision are given as arguments which will supply the values for **w** and **p**. For example,

```
printf("%*.*f",7,2,number);
```

is equivalent to

```
printf("%7.2f",number);
```

The advantage of this format is that the values for *width* and *precision* may be supplied at run time, thus making the format a *dynamic* one. For example, the above statement can be used as follows:

```
int width = 7;
int precision = 2;
........
........
printf("%*.*f", width, precision, number);
```

Example 4.10 All the options of printing a real number are illustrated in Fig. 4.10.

```
Program
    main()
    {
      float y = 98.7654;
      printf("%7.4f\n", y);
      printf("%f\n", y);
      printf("%7.2f\n", y);
      printf("%-7.2f\n", y);
      printf("%07.2f\n", y);
      printf("%*.*f", 7, 2, y);
      printf("\n");
      printf("%10.2e\n", y);
      printf("%12.4e\n", -y);
      printf("%-10.2e\n", y);
      printf("%e\n", y);
    }
Output
      98.7654
      98.765404
      98.77
      98.77
      0098.77
      98.77
      9.88e+001
      -9.8765e+001
      9.88e+001
      9.876540e+001
```

Fig. 4.10 *Formatted output of real numbers*

Printing of a Single Character

A single character can be displayed in a desired position using the format

[illegible]

The character will be displayed *right-justified* in the field of *w* columns. We can make the display *left-justified* by placing a minus sign before the integer w. The default value for w is 1.

Printing of Strings

The format specification for outputting strings is similar to that of real numbers. It is of the form

[illegible]

where *w* specifies the field width for display and *p* instructs that only the first p characters of the string are to be displayed. The display is *right-justified.*

The following examples show the effect of variety of specifications in printing a string "NEW DELHI 110001", containing 16 characters (including blanks).

```
Specification                Output

              1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
%s            N E W   D E L H I   1 1 0 0 0 1

%20s                  N E W   D E L H I   1 1 0 0 0 1

%20.10s                                   N E W   D E L H I

%.5s          N E W   D

%-20.10s      N E W   D E L H I

%5s           N E W   D E L H I   1 1 0 0 0 1
```

Example 4.11 Printing of characters and strings is illustrated in Fig. 4.11.

```
Program
    main()
    {
      char x = 'A';
      char name[20] = "ANIL KUMAR GUPTA";
```

```
        printf("OUTPUT OF CHARACTERS\n\n");
        printf("%c\n%3c\n%5c\n", x,x,x);
        printf("%3c\n%c\n", x,x);
        printf("\n");
        printf("OUTPUT OF STRINGS\n\n");
        printf("%s\n", name);
        printf("%20s\n", name);
        printf("%20.10s\n", name);
        printf("%.5s\n", name);
        printf("%-20.10s\n", name);
        printf("%5s\n", name);
    }
Output
        OUTPUT OF CHARACTERS
        A
          A
            A
          A
        A
        OUTPUT OF STRINGS
        ANIL KUMAR GUPTA
          ANIL KUMAR GUPTA
                ANIL KUMAR
        ANIL
        ANIL KUMAR
        ANIL KUMAR GUPTA
```

Fig. 4.11 *Printing of characters and strings*

Mixed Data Output

It is permitted to mix data types in one **printf** statement. For example, the statement of the type

```
printf("%d %f %s %c", a, b, c, d);
```

is valid. As pointed out earlier, **printf** uses its control string to decide how many variables to be printed and what their types are. Therefore, the format specifications should match the variables in number, order, and type. If there are not enough variables or if they are of the wrong type, the output results will be incorrect.

Table 4.3 *Commonly used printf Format Codes*

Code	*Meaning*
%c	print a single character
%d	print a decimal integer
%e	print a floating point value in exponent form