1. OPERATORS IN C

C language is very rich in built-in operators. The commonly used operators include

1. Arithmetic operators
2. Unary operators
3. Relational and Equality operators
4. Logical operators
5. Assignment operators
6. Conditional operator
7. Comma operator
8. Bitwise operators

1.1. ARITHMETIC OPERATORS

There are five arithmetic operators in C. They are

+ Addition
- Subtraction
* Multiplication
/ Division
% Remainder after integer division (Modulus Operator)

The operands of the arithmetic operators must represent numerical values – integer quantities, floating-point quantities, or characters. The modulus operator requires that both the operands be integers and the second operand be nonzero. Similarly, the division operator requires that the second operand be nonzero.

Division of one integer quantity by another is referred to as integer division. This operation always results in a truncated quotient (i.e., the decimal portion of the quotient will be dropped). On the other hand, if the division operation is carried out with two floating-point numbers, or with one floating-point number and one integer, the result will be a floating-point quotient.

Suppose that *a* and *b* are integer variables whose values are 10 and 3, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

Expression	Value
a + b	13
a - b	7
a * b	30
a / b	3
a % b	1

Notice that the truncated quotient resulting from the division operation, since both

the operands represent integer quantities.

Now suppose that *a* and *b* are floating-point variables whose values are 12.5 and 2.0, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

Expression	Value
a + b	14.5
a - b	10.5
a * b	25.0
a / b	6.25

Finally suppose that *a* and *b* are character variables that represent the characters *P* and *T*, respectively. Several arithmetic expressions that make use of these variables are shown below, together with their resulting values.

Expression	Value
a	80
a + b	164
a + b + 5	169
a + b + '5'	217

Note that '*P*' is encoded as 80 in decimal, '*T*' is encoded as 84, '5' is encoded as 53 in the ASCII character set.

If one or both operands represent negative values, then the addition, subtraction, multiplication and division operations will result in values whose signs are determined by the usual rules of algebra. Integer division will result in truncation towards zero; i.e., the resultant will always be smaller in magnitude than the true quotient. The interpretation of the remainder operation is unclear when one of the operand is negative. Most versions of C assign the sign of the first operand to the remainder.

Suppose that that *a* and *b* are integer variables whose values are 11 and -3, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

Expression	Value
a + b	8
a - b	14
a * b	-33
a / b	-3
a % b	2

If *a* had been assigned value of –11 and *b* had been assigned 3, then the value of *a*/*b* would be –3 but the value of *a*%*b* would be –2. Similarly, if *a* and *b* had both been assigned negative values (-11 and –3, respectively), the value of *a*/*b* would be 3

and the value of *a%b* would be –2.

1.2. UNARY OPERATORS

The class of operators that act upon a single operand to produce a new value are known as unary operators. The frequently used unary operators in C are (the others will be discussed later in the book):

- Unary Minus
- Increment and Decrement operator
- *sizeof* operator
- Cast operator

UNARY MINUS (-)

The operand of unary – operator must have arithmetic type, and the result is the negative of its operand. The operand can be numerical constant, variable, or expression. Note that the unary – operator is distinctly different from the arithmetic binary operator which denotes subtraction (-). The subtraction operator requires two separate operands. Here are several examples which illustrate the use of the unary minus operation.

-123 -(x+y) z = w+x* -y flip = -net -3*(x+y)

INCREMENT (++) AND DECREMENT (–) OPERATORS

Increment operator causes its operand to be increased by 1, whereas the decrement operator causes its operand to be decreased by 1. The operand used with each of these operations must be a single variable.

The ++ and -- operators can each be utilized in two different ways, depending on whether the operator is written before or after the operand. If the operator precedes the operand (e.g., *++i*), then the operand will be altered in value before it is utilized for its intended purpose within the program. If, however, the operator follows the operand (e.g., *i++*), then the value of the operand will be altered after it is utilized.

As an example, suppose that *n* is an integer variable that has been assigned a value of 7. The statement

```
a = ++n;
```

first increments and then assign the value 8 to a. But the statement

```
a = n++;
```

first assigns the value 7 to *a* and then increments *n* to 8. In both cases, though, the end result is that *n* is assigned the value 8.

Now consider another example. Suppose that a C program includes an integer

variable *i* whose initial value is 1. Suppose the program includes the following three *printf* statements.

```
printf("i=%d\n",i);
printf("i=%d\n",++i);
printf("i=%d\n",i);
```

These *printf* statements will generate the following three lines of output

```
i=1
i=2
i=2
```

The first statement causes the original value of *i* to be displayed. The second statement increments i and then displays its value. The final value of *i* is displayed by the last statement.

Now suppose that the program includes the following three *printf* statements, rather than the three statements given above.

```
printf("i=%d\n",i);
printf("i=%d\n",i++);
printf("i=%d\n",i);
```

The first and third statements are identical to those shown above. In the second statement, however, the unary operator follows the integer variable rather than precedes it. These statements will generate the following three lines of outputs.

```
i=1
i=1
i=2
```

The first statement causes the original value of *i* to be displayed, as before. The second statement causes the current value of *i* (1) to be displayed and then incremented to 2. The final value of *i* (2) is displayed by the last statement.

Rules for ++ and – operators are summarized below:

- 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 other unary operators like unary -.

THE *sizeof* OPERATOR

The *sizeof* operator returns the size of its operand in bytes. The *sizeof* operator always precedes its operand. The operand may be an expression, or it may be a cast. This operator allows the determination of the number bytes allocated to various types of data items. This information can be very useful when transferring a program to a different computer or to a new version of C. It also used for dynamic memory allocation.

Suppose that *i* is an integer variable, *x* is a floating-point variable, *d* is a double precision variable and *c* is character-type variable. The statements

```
printf("Size of Integer          : %d",sizeof i);
printf("Size of Floating-point   : %d",sizeof x);
printf("Size of Double           : %d",sizeof d);
printf("Size of Character        : %d",sizeof c);
```

might generate the following output

```
Size of Integer          : 2
Size of Floating-point   : 4
Size of Double           : 8
Size of Character        : 1
```

Thus, this version of C allocates 2 bytes to each integer quantity, 4 bytes to each floating-point quantity, 8 bytes to each double-precision quantity, and 1 byte to each character.

Another way to generate the same information is to use a cast rather than a variable within each *printf* statement. Thus, the *printf* statement could have been written as

```
printf("Size of Integer          : %d",sizeof (integer));
printf("Size of Floating-point   : %d",sizeof (float));
printf("Size of Double           : %d",sizeof (double));
printf("Size of Character        : %d",sizeof (char));
```

These *printf* statements will generate the same output as that shown above.

THE CAST OPERATOR

Cast operator can be used to explicitly convert the value of an expression to a different data type. To do so, the name of the data type to which the conversion is to be made (such as *int* or *float*) is enclosed in parentheses and placed directly to the left of the expression whose value to be converted. The word *cast* never is actually used. Its general format is

(data type) expression

Suppose that *f* is a floating-point variable and i is an integer variable. Expression *f* % *i* is invalid, because the first operand is a floating-point constant rather than an integer. However, it could be written as (int)*f* % *i*. The data type associated with the expression itself is not changed by a cast. Rather, it is the value of the expression that undergoes type conversion wherever the cast appears.

Other examples are

```
(int)3.1415
(int)n
(float)(4*a)/b
```

C includes several other unary operators. They will be discussed later in this book.

1.3. RELATIONAL AND EQUALITY OPERATORS

Relational and equality operators are symbols that are used to test the relationship between two quantities such as variables, constants or expressions. There are four relational operators in C. They are

Operator	Meaning
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

Closely associated with the relational operators are the following two equality operators.

Operator	Meaning
==	Equal to
!=	Not equal to

These six relational operators are used to form logical expressions, which represent conditions that are either *true* or *false*. The resulting expression will be of type integer, since *true* is represented by integer value 1 and false by the value of 0.

Suppose that *i*, *j* and *k* are integer variables whose values are 1, 2, and 3, respectively. Several logical expressions involving these variables are shown below.

Expression	Interpretation	Value
i<j	true	1
(i+j)>=k	true	1
(j+k)>(i+5)	false	0
k!=3	false	0
j==1	false	0
j==2	true	1

When carrying out relational and equality operations, operands that differ in type

will be converted in accordance with the rules discussed earlier.

Suppose that *i* is an integer variable whose value is 7, *f* is a floating-point variable whose value is 5.5, and *c* is a character variable that represents the character 'w'. Several logical expressions that make use of these variables are shown below.

Expression	Interpretation	Value
f>5	true	1
(i+f)<=10	false	0
c==119	true	1
c!='p'	true	1
c>= 10*(i+f)	false	0

1.4. LOGICAL OPERATORS

Logical operators are the symbols that are used to combine or negate expressions containing relational operators. There are three logical operators in C

Operator	Meaning
&&	logical and
||	logical or
!	logical not

The logical operators act upon operands that are themselves logical expressions. The net effect is to combine the individual logical expressions into more complex conditions that are either true or false. The result of the logical *and* operation will be true only if both operands are true, whereas the result of a logical *or* operation will be false only if both operands are false. The unary operator logical *not* is used to negate the value of a logical expression, i.e., it causes an expression that is originally true to become false, and vice versa.

Suppose that *i* is an integer variable whose value is 7, *f* is floating-point variable whose value is 5.5 and *c* is character-type variable that represents the character '*w*'. Several complex logical expressions that make use of these variables are shown below.

Expression	Interpretation	Value
(i>=6)&&(c=='w')	true	1
(i>=6)||(c==119)	true	1
(f<11)&&(i>100)	false	0
c!='p')||((i+f)<=10)	false	0
!(f>5)	false	0
!(i>(f+1))	false	0

Whenever logical/relational expressions are connected together using the logical

operators && (AND) or || (OR), only as many expressions as are needed to determine the overall logical value will be evaluated. This is known as *short-circuit behaviour*. For example, if two expressions are connected with && and the first expression is false, then it is guaranteed that the second expression will not be evaluated. The same is true of expressions connected with || where the first expression is true. For example:

```
if (b != 0 && a / b > 1)
    statement;
```

will prevent the evaluation of *a* / b if *b* is zero.

1.5. ASSIGNMENT OPERATORS

An assignment operator is used to form an assignment expression, which assigns the value of an expression to an identifier. The most commonly used assignment operator is =. Assignment expressions that make use of this operator are written in the form

identifier = expression

where *identifier* generally represents a variable, and *expression* represents a constant, a variable or a more complex expression.

Here are some typical assignment expressions that make use of = operator.

```
a=3
x=y
delta=0.0001
area=length*breadth
```

Remember that the assignment operator = and the equality operator == are distinctly different. The assignment operator is used to assign a value to an identifier, whereas the equality operator is used to determine if two expressions have the same value. Assignment expressions are often referred to as *assignment statements*, since they are usually written as complete statements. If two operands in an assignment expression are of different data types, then the value of the expression on the right will automatically be converted to the type of the identifier on the left. The entire assignment expression will then be of this data type. Under some circumstances, this automatic type conversion can result in an alteration of the data being assigned. For example,

- A floating-point value may be truncated if assigned to an integer identifier.
- A double precision may be rounded if assigned to floating-point identifier.
- An integer quantity may be altered if assigned to a short integer identifier or a character identifier.

In the following assignment expressions, suppose that *i* is an integer-type variable.

Expression	Value
i = 3.3	3
i = 3.9	3
i = -3.9	-3

Now suppose that *i* and *j* are both integer-type variables and *j* has been assigned a value of 5. Several assignment expressions that make use of these two variables are shown below.

Expression	Value
i = j	5
i = j/2	2
i = 2*j/2	5
i = 2*(j/2)	4

Finally assume that *i* is integer type variable, and the ASCII character set applies.

Expression	Value
i = 'x'	120
i = '0'	48
i = ('x'-'0')/3	24
i = ('y'-'0')/3	24

Multiple assignments of the form

identifier1 = identifier2 = ... = expression

are permissible in C. In such situations, the assignments are carried out from right to left. Thus the multiple assignments

identifier1 = identifier2 = expression

is equivalent to

identifier1 = (identifier2 = expression)

and so on, with right-to-left nesting for additional multiple assignments.

Suppose that *i* and *j* are integer variables. The multiple assignment expression

i = j = 5

will cause the integer value 5 to be assigned to both *i* and *j*. (To be more precise, 5 is first assigned to *j*, the value of *j* is then assigned to *i*)

Similarly the multiple assignment expression

i = j = 5.9

will cause the integer value 5 to be assigned to both *i* and *j*. Remember that the truncation occurs when the floating-point value 5.9 is assigned to *i*.

C contains the following five additional assignment operators

+= , -= , *= , /= , %

To see how they are used, consider the first operator, +=. The assignment expression

expression1 += expression2

is equivalent to

expression1 = expression1 + expression2

Similarly, the assignment expression

expression1 -= expression2

is equivalent to

expression1 = expression1 - expression2

and so on for all five operators.

Usually *expression1* is an identifier, such as variable or an array element.

Suppose that *i* and *j* are integer variables whose values are 5 and 7, and *f* and g are floating-point variables whose values are 5.5 and -3.25. Several assignment expressions that make use of these variables are shown below.

Expression	Equivalent expression	Final value
i += 5	i = i+5	10
f -= g	f = f-g	8.75
j *= (i-3)	j = j*(i-3)	14
f /= 3	f = f/3	1.8333
i % = (j-2)	i = i%(j-2)	0

1.6. THE CONDITIONAL OPERATOR

The conditional operator is represented by ? and : symbols. The conditional operator is a ternary operator, meaning it operates upon three values. Simple conditional operations can be carried out with conditional operator (?:). An expression that makes use of the conditional operator is called conditional expression. A conditional expression is written in the form

expression1 ? expression2 : expression3

The *expression1* is interpreted as a Boolean expression. It is a sort of switch, in that it determines which of the two expressions, *expression2* or *expression3*, is to be evaluated.

When evaluating a conditional expression, *expression1* is evaluated first. If *expression1* is true (i.e., if its value is non zero), then *expression2* is evaluated and

this becomes the value of the conditional expression. However, if *expression1* is false (i.e., if its value is zero), the *expression3* is evaluated and this becomes the value of the conditional expression. Note that only one of the embedded expressions (either *expression2* or *expression3*) is evaluated when determining the value of a conditional expression.

For example, in the conditional expression shown below, assume that *i* is an integer variable.

```
(i<0) ? 0 : 100
```

The expression (*i*<0) is evaluated first. If it is true, the entire conditional expression takes on the value 0. Otherwise, the entire conditional expression takes on the value 100.

In the following conditional expression, assume that *f* and *g* are floating-point variables.

```
(f < g) ? f : g
```

This conditional expression takes on the value of *f* if *f* is less than *g*; otherwise the conditional expression takes on the value of *g*. In other words, the conditional expression returns the value of the smaller of the two variables.

If the operands (i.e., *expression2* and *expression3*) differ in type, then the resulting data type of the conditional expression will be determined by the rules given earlier. Now suppose that *i* is an integer variable, and *f* and *g* are floating-point variables. The conditional expression

```
(f < g) ? i : g
```

involves both integer and floating-point operands. Thus the resulting expression will be floating-point, even if the value of *i* is selected as the value of the expression.

Conditional expressions frequently appear on the right-hand side of a simple assignment statement. The resulting value of the conditional expression will be assigned to the identifier on the left.

Here is an assignment statement that contains an assignment expression on the right-hand side.

```
min = (f < g) ? f : g
```

this assignment expression causes the value of the smaller of *f* and *g* to be assigned to min.

1.7. THE COMMA OPERATOR

The comma is frequently used in C as a simple punctuation mark, to separate variable declarations, function arguments, etc. In certain situations, the comma acts as an

operator. When used as an operator, the comma operator (represented by the token ,) takes two operands (*a*, *b*), evaluates the first, discards its value, evaluates the second and returns it's value as the result of the expression. Comma separated operands can be chained together, evaluated in left-to-right sequence with the right-most value yielding the result of the expression. The comma operator has the lowest precedence of any C operator. Some examples of the use of comma operator are shown below:

```
/* not an operator here*/
int a=1, b=2, c=3, i;

/* stores b into i  */
i = (a, b);

/* stores a into i */
i = a, b;

/* increases a by 2, then stores a+b, i.e. 3+2 into i */
i = (a += 2, a + b);

/* increases a by 2, then stores a i.e. 5 into i (as a is 3 from adding 2 in previous line) */
i = a += 2, a + b;
/* stores a into

i */i = a, b, c;
/* stores c into i */
i = (a, b, c);
```

Because the comma operator discards its first operand, it is generally only useful where the first operand has desirable side effects, such as in the initialiser or increment statement of a for loop.

The use of the comma token as an operator is distinct from its use in function calls and definitions, variable declarations, *enum* declarations, and similar constructs, where it acts as a separator.

The following table gives some more examples of the uses of the comma operator.

Statement	Effects
i= (1,2,3,4,5,6,7,8,9);	9 is assigned to *i*
j= (j=3, j+2);	5 is assigned to *j*
t=x, x=y, y=t;	Exchange *x* and *y*
for (i=0; i<2; ++i, f());	A *for* statement in which *i* is incremented and *f()* is called at each iteration.

Statement	Effects
if (f(), ++i, i>1) { /* ... */ }	An *if* statement in which function *f()* is called, variable *i* is incremented, and variable *i* is tested against a value. The first two expressions within this comma expression are evaluated before the expression *i>1*. Regardless of the results of the first two expressions, the third is evaluated and its result determines whether the *if* statement is processed.
func((++a, f(a)));	A function call to *func()* in which *a* is incremented, the resulting value is passed to a function *f()*, and the return value of *f()* is passed to *func()*. The function *func()* is passed only a single argument, because the comma expression is enclosed in parentheses within the function argument list.

1.8. BITWISE OPERATORS

Some applications require the manipulation of individual bits of an integer operand within a word of memory. The bitwise operators supported by C are given below:

Operator	Function	Usage
~	Bitwise NOT	~exp
<<	Left shift	exp1 << exp2
>>	Right shift	exp1 >> exp2
&	Bitwise AND	exp1 & exp2
^	Bitwise XOR	exp1 ^ exp2
|	Bitwise OR	exp1 | exp2
&=	Bitwise AND assign	exp1 &= exp2
^=	Bitwise XOR	exp1 ^= exp2
|=	Bitwise OR assign	exp1 |= exp2

A bitwise operator interprets its operand(s) as an ordered collection of bits, either individually or grouped as fields. The bitwise *NOT* operator flips the bits of its operands. Each 1 bit is set to zero; each 0 bit is set to 1. The bitwise *shift* operators (<< and >>) shift the bits of the left operand (*exp1*) some number of positions (*exp2*) either left or right. The operand's excess bits are discarded. The *left shift* operator inserts 0-valued bits in from the right. The *right shift* operator inserts 0-valued bits in from the left if the operator is unsigned. If the operand is signed, it can either insert copies of the sign bit or insert 0-valued bits, which is implementation defined.

The bitwise *AND* operator (&) takes two integral operands. For each bit position, the result is 1 if both operands contain 1; otherwise, the result is 0. The bitwise *XOR* operator (^) takes two integral operands. For each bit position, the result is 1 if either

but not operands contain 1; otherwise, the result is 0. The bitwise *OR* operator (|) takes two integral operands. For each bit position, the result is 1 if either or both operands contain 1; otherwise, the result is 0.

The usage of bitwise operators is illustrated below:

```
#include<stdio.h>
main()
{   int c1 = 0x8,c2 = 0x3,c3;
    char* printbinary(int);
    c3 = c1 & c2;
    printf("\n Results are in binary\n");
    printf("\n Given\n c1 = %s or 0X%x and \n c2 = %s or 0X%x",
                printbinary(c1),c1,printbinary(c2),c2);
    printf("\n Bitwise AND i.e. c1 & c2 = %s",printbinary(c3),c3);
    c3 = c1 | c2;
    printf("\n Bitwise OR i.e. c1 | c2 = %s or 0X%x",printbinary(c3),c3);
    c3 = c1 ^ c2;
    printf("\n Bitwise XOR i.e. c1 ^ c2 = %s or 0X%x",printbinary(c3),c3);
    c3 = ~c1;
    printf("\n Ones complement of c1 = %s or 0X%x",printbinary(c3),c3);
    c3 = c1<<2;
    printf("\n Left shift by 2 bits c1 << 2 = %s or 0X%x",printbinary(c3),c3);
    c3 = c1>>2;
    printf("\n Right shift by 2 bits c1 >> 2 = %s or 0X%x",printbinary(c3),c3);
}

char* printbinary(int b)
{   int i;
    char *binarystring = (char *) malloc(sizeof(char) * 33);
    binarystring[32]='\0';
    for (i = 31; i >=0 ; i--)
       binarystring[31-i]=((b>>i)&1)?'1':'0';
    return(binarystring);
}
```

Note that no conversion character is available in C for displaying an integer value in binary, though the conversion characters *o* and *x* are available for displaying the integer value in octal and hexadecimal forms. In the above program, a user-defined function *printbinary()* is used for the purpose.

The program outputs the following results:

```
Results are in binary
Given
c1 = 00000000000000000000000000001000 or 0X8 and
c2 = 00000000000000000000000000000011 or 0X3
Bitwise AND i.e. c1 & c2 = 00000000000000000000000000000000
Bitwise OR i.e. c1 | c2 = 00000000000000000000000000001011 or 0Xb
```

Bitwise XOR i.e. c1 ^ c2 = 00000000000000000000000000001011 or 0Xb
Ones complement of c1 = 11111111111111111111111111110111 or 0Xfffffff7
Left shift by 2 bits c1 << 2 = 00000000000000000000000000100000 or 0X20
Right shift by 2 bits c1 >> 2 = 00000000000000000000000000000010 or 0X2

1.9. OPERATOR PRECEDENCE AND ASSOCIATIVITY

Operator precedence determines the order in which the operators are to be evaluated in a complex expression. The operators in C are grouped in hierarchically according to their precedence. Operations with a higher precedence are carried out before operation having the lower precedence. However the natural order of precedence can be altered through the use of parentheses.

The *associativity* determines the order in which consecutive operations within the same precedence group are carried out. The precedence and associativity of all operators in C are summarized below.

	Precedence Group	Operators	Associativity
1	Function, array, structure member, pointer to structure member	() [] . ->	L→R
2	Unary operations	- ++ -- ! ~ * & sizeof (type)	R→L
3	Arithmetic multiply, divide and modulus	* / %	L→R
4	Arithmetic add and subtract	+ -	L→R
5	Bitwise shift operators	<< >>	L→R
6	Relational operators	< <= > >=	L→R
7	Equality operators	== !=	L→R
8	Bitwise AND	&	L→R
9	Bitwise Exclusive OR	^	L→R
10	Bitwise OR	|	L→R
11	Logical AND	&&	L→R
12	Logical OR	||	L→R
13	Conditional operator	?:	R→L
14	Assignment operators	= += -= *= /= %= &= ^= |= <<= >>=	R→L
15	Comma operator	,	L→R

2. EXPRESSIONS

An expression is composed of a single *operand* or more operands glued together by *operators*. An operand may be a constant, symbolic constant, name for a variable,

array element, e.g., structure/union element, function reference or another expression surrounded by parenthesis. Operators may be unary, binary or any other operators supported by C.

All expressions in C compute to a value of a specific data type such as integer, floating-point, etc. Expressions can also represent logical conditions that are either *true* or *false*. However, in C, the conditions *true* and *false* are represented by the integer values 1 and 0, respectively. Hence logical-type expressions really represent numerical quantities.

An expression by itself is not a statement. Remember, a statement is terminated by a semicolon; an expression is not. Expressions may be thought of as the building blocks of a statement, from which statements may be constructed. The important thing to note is that every C expression has a value

Since expressions are built from combinations of operators and operands, C support different types of expressions, including

- Constant expressions
- Integer expressions
- Float expressions
- Unary expression
- String expressions
- Pointer expressions
- Relational expressions
- Logical expressions
- Conditional expression
- Bitwise expressions
- Assignment expressions

An expression may also use combinations of above expressions. Such expressions are known as *compound or complex expressions*.

A *constant expression* consists of only constant values. For example, 15, 20+5/2.0, 'x' are constant expressions.

Integer expressions are those which produce integer results after implementing all the automatic and explicit type conversions. For example, *m*, *m*n*-5, *m**'x' and *m*int*(2.5) (where *m* and *n* are integers) are integral expressions.

Float expressions are those which, after all conversions, produce floating-point results. For example, *x+y*, *x*y*/10, and 10.15 (where *x* and *y* are floating-point variables) are float expressions.

Unary expressions are expressions involving one or more unary operators. Examples include: *-a*, sizeof(int), (int) *a*, *++i*, i--.

Any string constants or expressions involving strings and string functions can be considered as *string expression*.

Pointer expressions produce address values. For example, *&m*, *ptr*, and *ptr+1* (where *m* is a variable and *ptr* is a pointer variable) are pointer expressions.

Relational expressions yield values that are either true or false. For example, *x<=y*, *a+b == c+d*, *m+n > 100* are *relational expressions*. However, in C, the conditions *true* and *false* are represented by the integer values 1 and 0, respectively. Hence relational- or logical-type expressions really represent numerical quantities.

Logical expressions combine two or more *relational expressions* and produce the value that is either *true* or *false*. For example, *a>b && x == 10*, and *x==20 || y ==5* are logical expressions.

A *conditional expression* is an expression involving the conditional operator (?:). Example: *(a>b)?max=a:max=b*.

Bitwise expressions are used to manipulate data at bit level. They are basically used for testing and shifting bits. For example, x<<3, and y>>1 are bitwise expressions.

An *assignment expressions* has the form

variable = expression;

The assignment expression evaluates the expression on the right and assigns the resulting value to the variable on the left. Assignment expressions are often referred to as assignment statements, since they are usually written as complete statements.

2.1. ARITHMETIC EXPRESSIONS

Of the above types of expressions, the expressions that yield a numeric value are normally referred to as arithmetic expression. Basically, an *arithmetic expression* will be a combination of numeric operands (such as numeric variables, constants, array and function references that re and arithmetic operators arranged as per the syntax of the language and it returns a numeric value. C can handle any complex mathematical expressions. Some of the examples of C expressions are:

```
circumference= 2.0*3.14*radius;
area = 3.14*radius*radius
a+ 'f'
a+b-c*d/e%f
(x+(b*b-4*a*c))/(2*a)
```

Remember that C does not have an operator for exponentiation.

Expressions are evaluated using an assignment statement of the form:

variable = expression;

where *variable* is any valid entity that has a *lvalue*. 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.

```
root=(x + (b * b – 4 * a * c)) / (2 * a);
```

The blank space around an operator is optional and adds only to improve readability. When the above statement is used in a program, the variables *a*, *b*, *c* and *x* must be defined before they are used in the expression, i.e., all the entities on the right hand side of an arithmetic expression should have *rvalues* associated with them.

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.

For example, the following arithmetic expression with values $i = 8$ and $j = 5$ evaluates to 23, as illustrated below:

```
z=2*i/5+4*j-3%i+j-2;
```

When values are substituted for variables

```
2*8/5+4*5-3%8+5-2
```

First pass:

```
Step 1: 16/5+4*5-3%8+5-2
Step 2: 3+4*5-3%8+5-2
Step 3: 3+20-3%8+5-2
Step 4: 3+20-3+5-2
```

Second pass:

```
Step 5: 23-3+5-2
Step 6: 20+5-2
Step 7: 25-2
Step 8: 23
```

Rules for evaluation of arithmetic expression can be summarized as follows:

- 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.

2.2. TYPE CONVERSIONS

C allows the use of mixed mode expressions, in which the operands are of different data types. In such cases, the operands are converted before evaluation, to maintain compatibility between data types. It can be carried out by compiler automatically or by programmer explicitly.

IMPLICIT TYPE CONVERSION

Wherever possible and permissible by the rules of the language, the compiler performs type conversion of data items when an expression consists of data items of different types using the rule that the smaller type is promoted to the wiser type. This is called *implicit conversion* or *coercion*. The following rules apply to arithmetic operations between two operators with dissimilar data types.

1. If one of the operands is *long double*, the other will be converted to *long double* and the result will be in *long double*.
2. Otherwise, if one of the operands is *double*, the other will be converted to *double* and the result will be *double*.
3. Otherwise, one of the operand is *float*, the other will be converted to *float* and the result will be *float*.
4. Otherwise, if one of the operand is *unsigned long int*, the other will be converted to *unsigned long int* and the result will be *unsigned long int*.
5. Otherwise, one of the operand is *long int* and the other is *unsigned int*, then:
   a. If *unsigned int* can be converted to *long int*, the *unsigned long int* operand will be converted as such and the result will be *long int*.
   b. Otherwise, both operands will be converted to *unsigned long int* and the result will be *unsigned long int*.
6. Otherwise, if one of the operand is *long int*, the other will be converted to *long int* and the result will be *long int*.
7. Otherwise, if one of the operand is *unsigned int*, the other will be converted to *unsigned int* and the result will be *unsigned int*.
8. If none of the above conditions applies, then both the operands will be converted to *int*, and the result will be *int*.

The rules followed by the compiler for implicit conversion is can be summarized as:

Operand1	Operand2	Result
char	int	int
int	long	long
int	float	float
int	double	double
int	unsigned	unsigned
long	double	double
double	float	double

Consider the following statement to illustrate the automatic conversion:

```
int ival = 0;
ival = 3.541+3;
```

The final result is to assign *ival* the value of 6. The actual steps in accomplishing this are the following: Since 3.541 is a literal of type *double* and 3 is a literal of type *int*, the *int* type is promoted to *double* type prior carrying out the addition. These conversions are carried out automatically by the compiler. The next step is to assign the result 6.541 which is of *double* type to *int* type *ival*. The conversion of *double* type to *int* is accomplished automatically by truncation. Thus 6.541 is truncated to 6, which is the value assigned to ival. This type of conversion, where the variable of higher type is converted to a lower type is called a *demotion*.

EXPLICIT TYPE CONVERSION

The implicit conversion can lead to errors creeping into the program, if adequate care is not taken. Therefore, the use of explicit type conversion is recommended in mixed mode expressions. It is achieved by typecasting a value of a particular type, into the desired type using the cast operator. To type cast, the name of the data type to which the conversion is to be made (such as *int* or *float*) is enclosed in parentheses and placed directly to the left of the expression whose value to be converted. The word *cast* never is actually used. Its general format is

(datatype) expression

Suppose that *f* is a floating-point variable and *i* is an integer variable. Expression

```
f % i
```

is invalid, because the first operand is a floating-point constant rather than an integer. However, it could be written as

(int)f % i

The data type associated with the expression itself is not changed by a cast. Rather, it is the value of the expression that undergoes type conversion wherever the cast appears.

Other examples are

```
(int)3.1415
(int)n
(float)(4*a)/b
```

3. STATEMENTS

A statement specifies an action. Thus a statement causes the computer to carry out some action. There are three different classes of statements in C. They are

- Expression statements
- Compound statements
- Control Statements

An *expression statement* consists of a valid expression followed by a semicolon. The execution of an expression statement causes the expression to be evaluated.

Several expression statements are shown below

```
func();  /* a function call */
c=a+b;  /* an assignment statement */
b+f();  /* a valid, but strange statement */
;       /* an empty or null statement */
```

The first expression statement executes a function call. The second is an assignment statement. The third expression, though strange, is still evaluated by the compiler because the function *f*() may perform some necessary task. The final expression shows that a statement can be empty (sometimes called a *null statement*)

A compound statement consists of several individual statements enclosed within a pair of braces { }. The individual statements may themselves be expression statements, compound statements or control statements. Unlike expression statements, compound statement does not end with a semicolon.

A typical compound statement is shown below.

```
{ pi=3.14;
  circumference= 2.0 □ pi □ radius;
  area = pi □ radius □ radius;
}
```

This particular compound statement consists of three assignment-type expression statements.

Control statements are used to create special program features, such as logical tests, loops and branches. Control statements will be discussed later in this book.

4. LIBRARY FUNCTIONS

Technically speaking, one can create a useful, functional C program consist of solely of statements involving only C keywords. However, this is quite rare because C does not provide keywords that perform such things as input/output operations, high-level mathematical computations, or character handling. As a result, most programs include calls to various functions contained in C's standard library. All C compilers come with a standard library of functions that carry out various commonly used operations or calculations. Library functions that are functionally similar are usually grouped together as object programs in separate library files. These library files are supplied as a part of each C compiler. A library function is accessed by simply writing the function name, followed by a list of arguments that represent information being passed to the function. The arguments must be enclosed in parentheses and separated by commas. The arguments can be constants, variable names or more complex expressions. The parentheses must be present, even if there are no arguments. Some commonly used library functions are

Function	Type	Purpose
		<ctype.h>
isalnum(int c)	*int*	*isalpha(c)* or *isdigit(c)*
isalpha(int c)	*int*	*isupper(c)* or *islower(c)*
iscntrl(int c)	*int*	Is control character. In ASCII, control characters are 0x00 (NUL) to 0x1F (US), and 0x7F (DEL)
isdigit(int c)	*int*	Is decimal digit
isgraph(int c)	*int*	Is printing character other than space
islower(int c)	*int*	Is lower-case letter
isprint(int c)	*int*	Is printing character (including space). In ASCII, printing characters are 0x20 (' ') to 0x7e ('~')
ispunct(int c)	*int*	Is printing character other than space, letter, digit
isspace(int c)	*int*	Is space, formfeed, newline, carriage return, tab, vertical tab
isupper(int c)	*int*	Is upper-case letter
isxdigit(int c)	*int*	Is hexadecimal digit
tolower(int c)	*int*	Return lower-case equivalent
toupper(int c)	*int*	Return upper-case equivalent

const char ct)*		Compares *cs* with *ct*, returning negative value if *cs<ct*, zero if *cs==ct*, positive value if *cs>ct*
strncmp(const char cs, const char* ct, size_t n)*	*int*	Compares at most (the first) *n* characters of *cs* and *ct*, returning negative value if *cs<ct*, zero if *cs==ct*, positive value if *cs>ct*
strchr(const char cs, int c)*	*char**	Returns pointer to first occurrence of *c* in *cs*, or *NULL* if not found
strrchr(const char cs, int c)*	*char**	Returns pointer to last occurrence of *c* in *cs*, or *NULL* if not found.
strstr(const char cs, const char* ct)*	*char**	Returns pointer to first occurrence of *ct* within *cs*, or *NULL* if none is found
strlen(const char cs);*	*int*	Returns length of *cs*.

In order to use a library function it may be necessary to include certain files that contain information about the library functions. Such information is normally kept in header files, such as *stdio.h*, *stdlib.h*, *math.h*, *string.h*, etc. The header files can be accessed by the pre-processor statement *#include*.

#include<filename>

where the filename represents the name of the header file.

5. USE OF COMMENTS

Comments should always be included within a C program. If written properly, comments can provide a useful overview of the general programming logic. They also delineate major segments of a program, identify certain key items within the

program and provide other useful information about the program. Generally, the comments need not be extensive; a few well-placed comments can shed a great deal of light on an otherwise obscure program. Such comments can be of great use to the original programmer as well as to other persons trying to read and understand a program, since most programmers do not remember the details of their own programs over a period of time. This is especially true of programs that are long and complicated.

A comment is normally placed within the delimiters /* and */. A comment can appear anywhere within a program, and they may be of any length. A comment may start on the same line as another statement or on a line of its own. It may also extend over any number of lines if the terminating delimiter is placed at the appropriate point. Comments cannot be nested in C.

6. I/O FUNCTIONS

One of the primary advantages of the modern computers is their ability to communicate with the user during program execution. This feature enables the programmer to enter the values into variables as the occasion demands, without having to change the program itself. The advantage of this option is that execution stops each time the program needs to have a new value entered. Once the required value has been entered and the return key is pressed, execution resumes form the point at which it stopped. Thus a program written in C may requires data to be read into variables and data stored in variables need to be displayed. As a programming language, C does not provide any keyword or statement that performs input/output. Instead, input and output are accomplished through library functions provided by the C compilers. C's input/output system is quite large and consists of several different functions. The header for I/O functions is *<stdio.h>*. There are both console and file I/O functions. This chapter examines the console I/O functions that read input from the keyboard and provide output to the screen. The file I/O system will be discussed later in Chapter 5.

6.1. SINGLE CHARACTER INPUT FUNCTIONS

Single characters can be entered into the computer using the C library function *getchar()*. The *getchar()* function waits until a key is pressed and then returns its value. The key press is also echoed to the screen. The prototype of *getchar()* is

```
int getchar(void);
```

As its prototype shows, the *getchar()* function is declared as returning an integer even though a character is read. However, one can assign this value to a char variable, as is usually done, because the character is contained in the low-order byte (the high-order byte is usually zero). In general terms, a function reference would be written as


```
character variable = getchar();
```

where character variable refers to some previously declared character variable.

For example, a C program contains the following statements.

```
char c;
....
c = getchar();
```

The first statement declares that *c* is character type variable. The second statement causes a single character to be entered from the keyboard and then assigned to *c*.

The *getchar()* returns an *EOF* if an error occurs. The symbolic constant *EOF* is defined in *<stdio.h>* and is often equal to –1.

The *getchar()* function can also be used to read multi-character string, by reading one character at a time within a multi-pass loop.

There are some potential problems with getchar() function. For many compilers, *getchar()* is implemented in such a way that it buffers input until *Enter Key* is pressed. This is called a *line buffered input*. One has to press *Enter* before any character is returned. Also, since *getchar()* inputs only one character each time it is called, the line buffering may leave one or more characters in waiting in the input queue, which is annoying in interactive environments. Even though it is permissible for *getchar()* to be implemented as an interactive function, it seldom is.

For example, the following program is supposed to input characters from the keyboard and display it in reverse case. The program should terminate only when a period has entered.

```
#include<stdio.h>
#include<ctype.h>
main()
{   char c;
    do
    {   c=getchar();
        if(islower(c))
            c=toupper(c);
        else
            c=tolower(c);
        putchar(c);
    }while(c!='.');
}
```

This program may not behave as you expected because of the reasons discussed above.

Two of the most common alternatives to *getchar()* function are *getch()* and *getche()*. They have the following prototypes

```
int getch(void);
int getche(void);
```

For most compilers, the prototypes for these functions are found in the console header file <conio.h>.

(Note that these console functions may not work with GCC compilers.)

The *getch()* function waits for a key press after which it returns immediately. It does not echo each character to the screen. The *getche()* is same as *getch()*, but the key is echoed. These two functions are commonly used instead of *getchar()* when a character needs to read from the keyboard in an interactive program. For example, the previous program is shown below using *getche()* instead of *getchar()*.

```
#include<stdio.h>
#include<ctype.h>
main()
{   char c;
    do
    {   c=getche();
        if(islower(c))
            c=toupper(c);
        else
            c=tolower(c);
        putchar(c);
    }   while(c!='.');
}
```

When you run this version of the program, each time you press a key, it is echoed on the screen (not echoed if *getch()* is used) and also immediately transmitted to the program and displayed in reverse case. Input is no longer line buffered.

6.2. THE *putchar(c)* FUNCTION

Single characters can be displayed on the screen using C library function *putchar(c)*. The prototype for *putchar()* function is

int putchar(int c);

In putchar function, even though it is declared as taking an integer type parameter, one can call it using a character assignment. Only the low-order byte of its parameter is actually output to the screen (Remember that an integer has two bytes of memory associated with it and the character is contained in the low-order byte). The *putchar()* function returns the character written or *EOF* if an error occurs.

The *putchar()* function transmits a single character to the monitor. The character being transmitted will normally be represented as a character type variable. In general, a function reference would be written as

putchar(character variable);

where character variable refers to some previously declared character variable.

For example, a C program contains the following statements.

```
char c;
.....
putchar(c);
```

The first statement declares that *c* is a character type variable. The second statement causes the current value of *c* to be displayed on the screen.

The *putchar()* function can be used to output multiple character strings, by displaying one character at a time within a multi-pass loop.

6.3. THE *scanf(...)* FUNCTION

The scanf function can be used to enter any combination of numerical values, single characters and strings. The function returns the number of data items that have been entered successfully. The *f* in *scanf* stands for *formatted*.

In general terms, the scanf function is written as

scanf(format string, arg1,arg2,...,argn)

where the *format string* refers to a string containing some required formatting information, and *arg1*, *arg2*,..., *argn* are arguments that represent the individual data items. The arguments represent pointers that indicate the address of the data items within the computer's memory.

The format string consists of individual group of characters, with one character group for each input data item. Each character group must begin with a percent (%) sign. In its simplest form, a single character group will consist of the % sign followed by a conversion character, which indicates the type of the corresponding data item.

Within the format string, multiple character groups can be contiguous, or they can be separated by white space characters (i.e., blank spaces, tab, or newline character). If white space characters are used to separate multiple character groups in the format string, all the consecutive white space characters in the data will be read but ignored.

The more frequently used conversion characters for data input are

Conversion Character	Meaning
c	Data item is a single character
d	Data item is a decimal integer
e	Data item is a floating-point value
f	Data item is a floating-point value

%d → int. %s → str.
%f → float - %c → char

Conversion Character	Meaning
g	Data item is a floating-point value
h	Data item is a short integer
i	Data item is a decimal, octal, or hexadecimal integer
o	Data item is an octal integer
s	Data item is a string followed by a null character
u	Data item is an unsigned decimal integer
x	Data item is a hexadecimal integer
[...]	Data item is a string, which may include white space characters (scan list)

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

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

The arguments are written as variables or arrays, whose types match the corresponding character groups in the format string. The arguments are actually pointers that indicate where the data items are stored in computer's memory. Thus each variable name must be preceded by the *address operator* & (ampersand). However, array names should not begin with an ampersand, since the array name is really pointer to the first element in the array.

An example of scanf function is given below.

```
#include <stdio.h>
main()
{   char item_name[15];
    int item_code;
    float unit_price;
    .....
    scanf("%s %d %f", item_name,&item_code,&unit_price);
    .....
```

format specifies

Within the *scanf* function, the format string is "%s %d %f". It contains three character groups. The first character group, %s, indicates that the first argument (*item_name*) represents a string. The second character group, %d, indicates that the second argument (*&item_code*) represents a decimal integer value, and the third character group, %f, indicates that the third argument (*&unit_price*) represents a floating-point value.

The above *scanf* function could have been written as

```
scanf("%s%d%f", item_name, &item_code, &unit_price);
```

third data item will have a maximum field width of 15 characters and it will be assigned to a double precision variable.

It is possible to skip over a data item, without assigning it to the designated variable, by placing an asterisk (*) that follows the % sign within the appropriate format group. This feature is called *assignment suppression*. Assignment suppression is especially useful when one needs to process only a part of what is being entered. For example, given

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

One could enter the data 10,10. The comma would be correctly read, but not assigned to anything.

The following points must be remembered while using the *scanf* statement:

1. All the function arguments, except the control string, must be pointers to variables
2. Each variable to read must have a field specification
3. For each field specification, there must be a variable address of proper type
4. Format specification contained in the control string should match the arguments in order.
5. Any non-whitespace character used in the format string must have a matching character in the user input.
6. Input data items must be separated by white spaces and must match the variable receiving the input in the same order.
7. The reading will be terminated when the *scanf* encounters a mismatch of data or a character that is not valid for the value being read or maximum number of characters have been read or the end-of-file is reached or an error is detected.
8. When searching for a value, *scanf* ignores line boundaries and simply looks for the next appropriate character.
9. Any unread data item in a line will be considered as part of the data input line to the next *scanf* call.
10. When the field width specifier *w* is used, it should be large enough to contain the input data size.

6.4. THE *printf(...)* FUNCTION

Output data can be written from the computer onto a standard output device using the library function *printf*. The *f* in *printf* stands for *formatted*. This function can be used to output any combination of numerical values, single characters and strings. It is similar to the input function *scanf*, except that its purpose is to display data rather

than to enter it into the computer. That is, the *printf* function moves data from the computer's memory to the standard output device, whereas the *scanf* function enters data from the standard input device and store it in the computer's memory.

In general terms, the *printf* function is written as

printf(format string, arg1,arg2,...,argn)

where the *format string* refers to a string containing some required formatting information, and *arg1*, *arg2*,..., *argn* are arguments that represent the individual data items. The arguments can be written as constants, single variable, or array names, or more complex expressions. Function references may also be included. Unlike the *scanf* function, the arguments in the *printf* function do not represent memory addresses and therefore are not preceded by ampersands.

The format string consists of individual group of characters, with one character group for each output data item. Each character group must begin with a percent (%) sign. In its simplest form, an individual character group will consist of the % sign followed by a conversion character, which indicates the type of the corresponding data item.

Within the format string, multiple character groups can be contiguous, or they can be separated by other characters, including white space characters. These "other" characters are simply transferred directly to the output device, where they are displayed. Some of the more frequently used conversion characters for data input are

Conversion Character	Meaning
c	Data item is displayed as a single character
d	Data item is displayed as a signed decimal integer
e	Data item is displayed as a floating-point value with an exponent.
f	Data item is displayed as a floating-point value without an exponent.
g	Data item is displayed as a floating-point value using either e-type or f-type conversion, depending on the value. Trailing zeroes and trailing decimal points are not displayed.
i	Data item is displayed as a single decimal integer
o	Data item is displayed as an octal integer, without a leading zero
s	Data item is displayed as a string.
u	Data item is displayed as an unsigned decimal integer
x	Data item is displayed as a hexadecimal integer, without leading 0x

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

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

For example, the following C program illustrates the use of *printf* statement.

```
#include <math.h>
#include <stdio.h>
main()
{   float i=2.0,j=3.0;
    scanf("%f %f %f %f %f",i,j,i+j,sqrt(i),sqrt(i+j));
}
```

The first two arguments within the *printf* function are single variables, the third argument is an arithmetic expression, and the last two arguments are function references.

The following skeletal outline of a C program illustrates how several different types of data can be displayed using the *printf* function.

```
#include <stdio.h>
main()
{   char item_name[15];
    int item_code;
    float unit_price;
    .....
    printf("%s %d %f",item_name,item_code,unit_price);
    .....
}
```

```
#include <stdio.h>
main ( ) /* labelling of output */
{    int a=10, b=20, c=30;
     printf("The numbers are: a=%d, b=%d and c=%d",a,b,c);
}
```

When run, the program produces the following labelled output.

```
The numbers are: a=10, b=20 and c=30
```

6.5. THE *gets()* AND *put(s)* FUNCTIONS

The *gets()* and *puts()* functions accepts a single argument. The argument must be a data item that represents a string. The *gets()* function is used to read a string from a standard input device. The *gets()* function will be terminated by a newline character. The newline character will not be included as part of the string. The string may include white space characters. The string will be terminated by a null character.

The *puts()* function is used to display a string on a standard output device. The *puts()* function automatically inserts a newline character at the end of each string it displays, so each subsequent string displayed with *puts()* is on its own line. The usage of *gets()* and *puts()* functions are illustrated in the following program.

```
#include<stdio.h>
main()
{ char name[20];
  printf("Please enter your name : ");
  gets(name);
  printf("Hello ");
  puts(name);
}
```