UNIT 3 OPERATORS AND EXPRESSIONS IN C

Structure **Page No.**

3.1 INTRODUCTION

This Unit introduces some basic concepts of the C language. If we have an expression like `2+7%3`, will the C program first add 2 and 7 and then find the reminder on division by 3 or will it add the reminder on division of 7 by 3 to 2? In Sec. 3.2, we will discuss rules that govern the evaluation in such situations. In Sec. 3.3, we discuss the concept of C expressions. In Sec. 3.4, we discuss **lvalues** and **rvalues**. These concepts are important pre-requisites for understanding arrays and pointers. When variables of two different types like **float** and **long int** occur in a computation, what will be the type of the result? We discuss such questions in Sec. 3.5. In Sec. 3.6, as we promised in the previous unit, we will discuss how to control the output and input in `printf()` and `scanf()` functions, respectively.

Objectives

After studying this unit, you should be able to

- write and evaluate complex C expressions, built with the arithmetic operators of C;
- explain the order of precedence among operators, and direction in which each associates;
- explain the concept of lvalue and rvalue of a variable;
- explain how these properties help decide the sequence in which the various parts of a C expression are evaluated; and
- explain how to use format control statements in `printf()` and `scanf()` to control input and output in C programs.

3.2 ELEMENTARY ARITHMETIC OPERATIONS AND OPERATORS

Let us start our discussion of arithmetic operators by the discussing a simple statement. Let us look at the statement. 3 times 4 is 12. The "times" or multiplication operation is represented in C, as it is in most computer languages, by the asterisk, *. In other words, the **operator** for **multiplication** is the *. It's a **binary operator** because it takes **two operands**, the multiplier and the multiplicand, namely 4 and 3 respectively in the present case. Now consider the C statement:

```
x=3*4;
```

There are **two** operators at work here. This may seem a little strange to you: the only visible operator, you might think, is the operator for multiplication, the asterisk. Not quite true. The second operator in the statement above is the **assignment operator**, the =. It assigns the value of the quantity on its right hand side to the variable named on its left, overwriting its previous value. The point to note is that **the assignment of a value to a variable is an operation**, the execution of a machine instruction, just as much as multiplication is an operation. The assignment operation causes the CPU to seek out the RAM location of the variable, and to deposit therein the value assigned to the variable.

The Random Access Memory in a computer may be thought of as a linear collection of words, each word being a basic unit of storage. Every word of memory has a unique "address", which is a positive integer that helps identify the word. The addresses of successive words increase uniformly from the lowest to the highest value. This is not unlike the way houses along a road have numbers by means of which they are located

Once the CPU is given the address of a word, selected at random, it can rapidly gain access to its contents, in a fixed small amount of time (of the order of a hundred millionth of a second), no matter what random value (between the lowest and the highest) the address of the word may have. Hence the name: Random Access Memory.

After a program has been compiled, each variable is assigned to a word of RAM. This association between a program variable and its memory address remains unchanged throughout the execution of the program. When you refer to a variable by its identifier, the computer translates the reference to the address of the word in which the variable is stored. The contents of a word may change as the program executes, just as much as the residents of a house may change with the passage of time. But for the duration of time that a family lives in a house, the association of the family's name with the address of the house is fixed.

Similarly, for as long as a program module executes, the association of a variable with its storage location remain fixed.

Now you can see why statements such as:

```
n = n + 1;
```

make sense to a computer. To a person unfamiliar with programming the reflex reaction on encountering such a statement is, "How can `n` possibly equal `n + 1`?" The fact is that in such a statement one is not asserting the equality of `n` to `n + 1`. One is actually instructing the CPU: "Replace the value in the storage location marked `n`, by whatever it was plus 1." To emphasis the difference between the act of assignment of a value to a variable, commonly done in computer programs, and the assertion of equality of the left and right hand side quantities in an algebraic equation, commonly done in high school problems, Pascal uses the compound symbol: = for assignment. In C the operator for assignment is the = symbol, and the statement:

```
n = n + 1;
```

has the meaning:

"Make the new contents of `n` its old contents + 1"

C operators have two properties with which you must become familiar: these are **priority** and **associativity**. When there is more than one operator occurring in an expression, it is the relative priorities of the operators with respect to each other that will determine the order in which the expression will be evaluated. The associativity defines the direction, left-to-right or right-to-left, in which the operator acts upon its operands. You may have noticed that we quietly slipped in a new word, "expression", without defining it first: here's the definition: an **expression** is a syntactically valid combination of operators and operators, that computes to a value. The number 7, or any

other number by itself, is an expression with no operands. It's a valid C expression. It's value is the value of the number, in the present case, 7. 3 + 8 is another valid expression, in which the operands 3 and 8 are syntactically connected by +, the operator for addition. Though we hate to be so obvious, it must be stated that this last expression computes to the value 11.

The assignment operator assigns the quantity on its right to the variable named on its left. In other words, **it groups, or associates from right to left**. In contrast, **the multiplication operator groups from left to right**, as indeed do most C operators. Basically, this means that if C sees a statement like:

```
w = x * y * z;
```

it will first compute `x*y`, (grouping from left to right), and will then multiply this value by z. Then, and only then, will it assign the value of the product `x*y*z` or `w`.

It is important to realise that the action of assignment is performed **after** the computations have been done. This is the natural order of arithmetic, and this, clearly, is what one should expect—first compute a value, then assign it to the variable on the left. Here then is the reason for the built in priority of C operators, and for the direction of their association.

It makes sense for the priority of the assignment operator to be lower than the priorities of an the arithmetic operators, and for it to group from right to left. Naturally it is very important for you to become adept at the precedence and grouping properties of all of C's operators. But if you are not sure of the order in which operators will be evaluated in a computation, you may use the **parentheses operator**, the (), to override default priorities. (Yes, even the parentheses is an operator in C!) The parentheses operator has a priority higher than any binary operator, such as that for multiplication; it groups from left to right. Thus in the statements:

```
w = x * (y * z);
```

the product `y * z` will be computed first; the value obtained will then be multiplied by `x`; lastly the assignment of the result will be made to `w`. Had the parentheses been absent, the order of the computation would have been:

1) The multiplication of `x` by `y`, with the result stored as an intermediate quantity.
2) The multiplication of this quantity by `z`.
3) The assignment of the result to `w`.

Generally we will state the direction of association and the relative priority of each operator of C with respect to the others when we introduce it. But for convenience a table (in order of decreasing operator priority) stating their direction of grouping is given in Table. 1 on the following page. [Not all of these operators may be available in non-ANSI implementations.] The parenthesis is an example of a **primary** operator; C has in addition three other primary operators: the **array operator** (`[]`), and the **dot** (`.`) and **arrow**`->` operators which we will encounter in later Units. All these operators have the same priority, higher than that of any other operator. They all group from left to right.

Aside from the primary operators, C operators are arranged in priority categories depending on the number of their operands. Thus a **unary** operator has but a single operand, and a higher priority than any **binary** operator, which has two operands. Binary operators have a higher priority than the **ternary** operator, with three operands. The **comma** operator may have any number of operands, and has the lowest priority of all C operators. Table. 1 on the next page reflects this rule.

One readily available example of a unary operator is the **operator for negation**, the `-`. It changes the sign of the quantity stated on its right. Since the unary operators have a higher priority than the assignment operator, in the statement:

Table 1: Precedence and Associativity of Operators.

Operators	Associativity
`(), [], ->,.,`	L to R
`!, ~, ++, -,+,-, *`derefencing operator`, &, (type cast), sizeof,` (all unary)	R to L
`*,/,%,`	L to R
`+ , -`	L to R
`<<,>>`	L to R
`<, <=, >, >=`	L to R
`==,!=`	L to R
`&`	L to R
`^`	L to R
`|`	L to R
`&&`	L to R
`||`	L to R
`?  :` (Ternary if-then-else operator)	L to R
`=, +=, -=, *=, /=, %=, &=, ^=, |=, «=, »=`	R to L
`,` (comma operator)	L to R

```
x = -3;
```

the 3 is **first** negated, and only **then** is this value assigned to `x`. The negation operator has a priority just below that of the parentheses operator; it groups from right to left. (Right to left association is a property the operator for negation shares in common with all unary operators.) In the following statement:

```
x = -(3*4);
```

the presence of the parentheses ensures that the expression `3*4` is evaluated first. It is then negated. Finally `x` is assigned the value -12.

A question that you might have is: Does C have a unary plus operator, +? In other words can one make an assignment of the form `a = + 5`? Not in compilers conforming to the **K** & **R** standard, though ANSI C does provide a unary plus operator. See Table. 1.

As we learnt in the last unit, C provides operators for other elementary arithmetic operations, such as addition, subtraction, division and residue-modulo (the operation that yields the remainder after division of any integer by another). They are respectively `+, -, /` and `%` (Refer to Program 2.3 and 2.4). Here we quickly recapitulate their properties. Each of these operators requires two operands. (The binary operator for subtraction must be distinguished from the unary operator for negation.)

If `int` variables `a, b, c` and `d` are given the values 3, 4, 5 and 6 respectively, then:

```
a+b  is 7,   /*  +,  the operator for addition */
b-c  is -1,  /*  -,  the operator for subtraction */
a*b  is 12,  /*  *,  the operator for multiplication */
c/b  is 1,   /*  /,  the operator for division: the remainder is discarded
                     when one int is divided by another. */
a%d  is 3,   /*  %   the residue modulo operator; it yields the remainder
                     after division of a by d */
```

Important: In the division of one integer by another the remainder is discarded. **Thus 7/3 is 2, and 9/11 is 0.**

The multiplication, division and residue-modulo operators have each the same priority. The addition and subtraction operators also have equal priority, but this is lower than that of the former three operators. *, / and %. All these operators group from left to right.

In a C program, is the value of 3/5+2/5 the same as (3+2)/5? Is 3*(7/5) the same as 3*7/5? Let us look at some examples to understand how the priorities work in the case of arithmetic operators.

Example 1: In the examples below let `x` be an `int` variable:

i) `x = 2 * 3 + 4 * 5;`

The products `2*3` and `4*5` are evaluated first; the sum `6 + 20` is computed next; finally the assignment of 26 is made to `x`.

ii) `x = 2 * (3 + 4) * 5;`

The parentheses guarantee that `3 + 4` will be evaluated first. Since multiplication groups from left to right, the intermediate result 7 will be multiplied by 2, and then by 5, and the assignment of 70 will finally be made to `x`.

iii) `x = 7 * 6 % 15 / 9;`

Each of the operators above has equal priority; each groups from left to right. Therefore, the multiplication 7 * 6 (= 42) is done first, then the residue modulo with respect to 15 (42 % 15 = 12), and finally the division (of 12) by 9. Since the division of one integer by another yields the integer part of the quotient, and truncates the remainder, 12 / 9 gets the value 1. `x` is therefore asigned the value 1.

iv) `x = 7 * (6 % 15)/ 9;`

The parentheses ensure than 6 % 15 is evaluated first. The remainder when 6 is divided by 15 is 6. In the second step this result is multiplied into 7, yielding 42. Integer division of 42 by 9 gives 4 as the quotient, which is the value assigned to `x`.

v) `x = 7 * 6 % (15 / 9);`

15 / 9 is performed first, yielding 1; the next computation in order is 7 * 6 % 1, i.e. the remainder on division of 42 by 1, which is 0. `x` gets the value 0.

vi) `x = 7 * ((6 % 15)/ 9);`

The innermost parentheses are evaluated first: 6 % 15 is 6. The outer parentheses are evaluated next, 6/9 is 0. `x` gets the value 7 * 0 = 0.

* * *

Try some exercises now to check your understanding of priorities of operators.

E1) Verify by creating and executing a C program that for `int` variables `a` and `b`, `a % b` equals `a - (a / b) * b`, regardless of whether `a` or `b` is negative or positive.

E2) Find the value that is assigned to the variables `x, y` and `z` when the following program is executed:

```
/* Program3.1; File name:unit3-prog1.c*/
#include<stdio.h>
int main()
{
    int x, y, z;
    x = 2 + 3 - 4 + 5 - (6 - 7);
    y = 2 * 33 + 4 * (5 - 6);
    z = 2 * 3 * 4 / 15 % 13;
    x = 2 * 3 * 4 / (15 % 13);
```

```
    y = 2 * 3 * (4 / 15 % 13);
    z = 2 + 33 % 5 / 4;
    x = 2 + 33 % -5 / 4;
    y = 2 - 33 % -5 / -4;
    z = -2 * -3 / -4 % -5;
    x = 50 % (5 * (16 % 12 * (17 / 3)));
    y = -2 * -3 % -4 / -5 - 6 + -7;
    z = 8 / 4 / 2 * 2 * 4 * 8 % 13 % 7 % 3;
return (0);
}
```

By inserting appropriate calls to `printf()`, verify that the answers you obtained are correct.

E3) Give the output of the following program:

```
/* Program 3.2; File name:unit3-prog2.c */
#include <stdio.h>
int main()
{
    int x = 3, y = 5, z = 7, w;
    w = x % y + y % x - z % x - x % z;
    printf("%d\n", w);
    w = x / z + y / z + (z + y) / z;
    printf("%d\n", w);
    w = x / z * y / z + x * y / z;
    printf("%d\n", w);
    w = x % y % z + z % y % (y % x);
    printf("%d\n", w);
    w = z / y / y / x + z / y / (y / x);
    printf("%d\n", w);
    return (0);
}
```

We have reached the end of this section. In the next section, we will discuss the concept of expressions in C.

3.3 EXPRESSIONS

An `expression` in C consists of a syntactically valid combination of operators and operands, that computes to a value. 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 constituent elements of a statement, the "building blocks" from which statements may be constructed. The important thing to note is that **every C expression has a value**. The number 7, as we said a while ago, or any other number by itself, is also an expression, the value of the number being the value of the expression.

`3 * 4 % 5`

is an expression with value 2.

`x = 3 * 4`

is an example of an **assignment expression**. (Note the absence of the semicolons in the assignment above. The terminating semicolon would have converted the expression into a statement.) Like any other C expression, **an assignment expression also has a value. It's value is the value of the quantity on the right hand side of the assignment operator**. So, in the present instance the value of the expression `(x = 3 * 4)` is 12. Therefore it is, that in C, statements such as:

`z = (x = 3 * 4)/ 5;`

are meaningful. Here the parentheses ensure that x is assigned the value 12 first. **12 is also the value of the parenthetical expression (x = 3 * 4)**, from the property that every expression has a value. Thus, the entire expression reduces to:

```
z = 12/5
```

Next in order of evaluation is the integer division of 12 by 5, yielding 2. The leftmost assignment operator finally bestows the value 2 to z. x continues to have the value 12.

Consider now the expression:

```
x = y = z = 3
```

The assignment operator groups from right to left. Therefore the rightmost assignment:

```
z = 3
```

is made first. z get the value 3; this is also the value of the rightmost assignment expression, `z = 3`. In the next assignment towards the left the expression is:

```
y = z = 3.
```

since the sub-expression `z = 3` has the value 3, we are saying in effect:

```
y = (z = 3)
```

i.e.

```
y = 3
```

the assignment to y is again of the value 3. Equally,the entire expression:

```
y = z = 3
```

gets the value 3. In the final assignment towards the left x gets the value of this latter expression:

```
x = (y = (z = 3))
```

Value of each parenthetical expression is 3. Thus x is 3. `printf()` prints expressions just as easily as it prints variables. This is illustrated in the program in Listing 1 below:

```
/* Program 3.3; File name:unit3-prog3.c */
#include <stdio.h>
int main()
{
    int x, y, z;
    printf("x = %d\n", x = (y = 2 * (z = 12 * 13 / 14)) % 5);
    printf("x = %d\n", x = 3 * (y = x % (z = 2 * (x = 5))));
    printf("x = %d\n", x = 2 * (x = z % (y = 10 -
          (x = 3 * (z = 13 % (x = 3 * (y = 12 / (z = 5)))))));
    return (0);
}
```

Listing 1: Values of expression.

In the first `printf`(), z gets the value 156 / 14 i.e. 11, y gets the value 22, x is y % 5, i.e. 2, which is the value of the expression output.

In the second `printf`(), x is first assigned the value 5, then z becomes 10, then y becomes 5 % 10, i.e. 5, finally x becomes 15; this is also the value of the expression, which is printed.

In the third `printf`(), z is 5, y is 2, and x is 6, to begin with; then z becomes 13 % x, i.e. 1, x is 3 * z, i.e. 3, and y is 7; then x gets the value z % y, which is 1, finally x becomes 2 * 1, and this also the value output.

3.3.1 Abbreviated Assignment Expressions

It is frequently necessary in computer programs to make assignment such as:

```
n = n + 5;
```

C allows a shorter form for such statements:

```
n += 5;
```

Assignment expression for the other arithmetic operations may be similarly abbreviated:

```
n -=5;   /* is equivalent to n = n - 5; */
n *=5;   /* is equivalent to n = n * 5; */
n /=5;   /* is equivalent to n = n / 5; */
n %=5;  /* is equivalent to n = n % 5;*/
```

The priority and direction of association of each of the operators +=, -+, *=, /= and %= is the same as that of the assignment operator.

E4) Guess the output of the following program.

```
/* Program 3.4; File name:unit3-prog4.c */
#include <stdio.h>
int main()
{
    printf("%d\n", -1 + 2 - 12 * -13 / 14);
    printf("%d\n", -1 % -2 + 12 % -13 % -4);
    printf("%d\n", -4 / 2 - 12 / 4 - 13 % -4);
    printf("%d\n", (-1 + 2 - 12) * (-13 / -4));
    printf("%d\n", (-1 % -2 + 12) % (-13 % -4));
    printf("%d\n", (-4 / 2 - 12) / (4 - 13 % -4));
    return (0);
}
```

Check your answer by compiling the program.

E5) State the output of the following programs:
(Hint: A single `printf()` may be used to print the values of several variables or expression. See Section 3.6 for details. In the `printf()` functions below the first `%d` corresponds to `x`, the second to `y`, the third to `z`, and so on. There must be a one-one correspondence between the format conversion characters (the `%d`s) and the variables in the argument list. Note the commas after the control string, and between identifiers.)

```
/* Program 3.5; File name:unit3-prog5.c */
#include <stdio.h>
int main()
{
    int x = 3, y = 5, z = 7, w = 9;
    w += x;
    printf("w = %d\n", w);
    w -= y;
    printf("w = %d\n", w);
    x *= z;
    printf("x = %d\n", x);
    w += x + y - (z -= w);
    printf("w = %d, z = %d\n", w, z);
    w += x -= y %= z;
    printf("w = %d, x = %d, y = %d\n", w, x, y);
    w *= x / (y += (z += y));
    printf("w = %d, y = %d, z = %d\n", w, y, z);
    w /= 2 + (w %= (x += y - (z -= -w)));
    printf("w = %d, x = %d, z = %d\n", w, x, z);
    return (0);
}
```

```
/* Program 3.6; File name:unit3-prog6.c */
#include <stdio.h>
int main()
{
    int x = 7, y = -7, z = 11, w = -11, s = 9, t = 10;
    x += (y -= (z *= (w /= (s %= t))));
    printf("x=%d, y=%d, z=%d, w=%d, s=%d, t=%d\n",
x, y, z, w, s, t);
    t += s -= w *= z *= y %= x;
    printf("x=%d, y=%d, z=%d, w=%d, s=%d, t=%d\n",
x, y, z, w, s, t);
    return (0);
}
```

E6) Given that `x,y, z` and `w` are `int`s with the respective values 100, 20, 300 and 40, find the outputs from the `printf`()s below:

```
printf("%d\n%d\n%d\n%d", x, y, z, w);
printf("%d\n%d\n%d\n%d", x, y, z, w);
printf("%d%d%d%d%d%d%d%d", x, y, w, z, y, w, z,
x);
printf("%d %d", x + z - y * y, (y - z % w) *
x);
```

3.3.2 Incrementation and Decrementation Operators

Probably the commonest form of assignment to be found in computer programs is of the form:

```
n = n + 1;
```

or

```
n = n - 1;
```

in which a variable `n` is incremented or decremented. Typically `n` may be the index of a loop which is changed by unity in each execution of the loop. From the point of view of program efficiency it is important that such assignments be executed as rapidly as possible, (in fact many modern assembly languages provide for this by including opcodes such as INC or DEC in their instruction sets) and C has special operators, called the **incrementation** and **decrementation** operators for these operations.

The incrementation and decrementation operators (there are two, a **pre-** and a **post-** operator for each operation) are unary operators. They have a priority as high as that of the unary negation operator, and, after the fashion of unary operators, these operators too group from right to left.

The **post-incrementation** operator ++ is written to the right of its operand.

```
n ++;
```

The effect of the statement above is to add one to the current value of `n`; its action is equivalent to:

```
n = n + 1;
```

Besides being more concise, the post-incrementation operation on some computers may be executed faster than this latter form of assignment.

The **post-decrementation** operation is represented by `--` and is written:

```
n --;
```

It decrements `n` by unity, producing a result identical to that of:

```
n = n - 1;
```

The **pre-incrementation** and **pre-decrementation** operators are placed to the left of the operand on which they are to act, and the result of their action is, as before, to increment or decrement the operand:

```
-- n;  /* assigns n - 1 to n */
++ n;  /* assigns n + 1 to n */
```

As far as their operand n is concerned, the pre- and post- incrementation or decrementation operators are equivalent. Each adds or subtracts one to its operand.

Where the pre- and post-operators differ is in the value used for the operand `n` when it is embedded inside expressions.

If it's a "pre" operator the value of the operand is incremented (or decremented) **before** it is fetched for the computation. The altered value is used for the computation of the expression in which it occurs.

To clarify, suppose that an `int` variable `a` has the value 5. Consider the assignment:

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

Pre-incrementation implies:

```
step 1:   increment a;              /* a becomes 6 */
step 2:   assign this value to b;   /* b becomes 6 */
result:   a is 6, b is 6
```

If it's "post" operator the value of the operand is altered **after** it is fetched for the computation. The unaltered value is used in the computation of the expression in which it occurs. Suppose again that `a` has the value 5 and consider the assignment:

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

Post-incrementation implies

```
step 1:   assign the uni-increment a to b ;   /* b becomes 5 */
step 2:   increment a;                        /* a becomes 6 */
result:   a is 6, b is 5
```

The placement of the operator, before or after the operand, directly affects the value of the operand that is used in the computation. When the operator is positioned **before** the operand, the value of the operand is altered **before** it is used. When the operator is placed **after** the operand, the value of the operand is changed **after** it is used. Note in the examples above that the variable `a` has been incremented in each case.

Suppose that the `int` variable `n` has the value 5. Now consider a statement such as:

```
x = n ++ / 2;
```

The post-incrementation operator, possessing a higher priority than all other operators in the statement, is evaluated first. But the value of `n` that is used in the computation of **x is still 5**! Post-incrementation implies: use the current value of `n` in the computation; increment it immediately **afterwards**.

So `x` gets the value 5 /2 = 2, even though `n` becomes 6. We repeat the rule: In an expression in which a post-incremented or post-decremented operand occurs, the current (unaltered) value of the operand is used; then, and only then, is it changed. Accordingly, in the present instance, 5 is the value of `n` that's used in the computation. `n` itself becomes 6.

Now consider:

```
x = ++ n / 2;
```

where n is initially 5.

Pre-incrementation or pre-decrementation first alters the operand n; it is this new value which is used in the evaluation of x. In the example, n becomes 6, as before; but this new value is the value used in the computation, not 5. So x gets the value 6 / 2 = 3.

Let's work through some the assignment statements in the program in Listing 2, to see how x, y, z and w get their values:

```
/* Program 3.7; File name:unit3-prog7.c */
#include<stdio.h>
int main()
{
    int x = 1, y = 2, z = 3, w;
    w = x++ + ++y - --z;
    w = --x - y-- + z++;
    w = x++ * ++y % ++z;
    w = --x / --y * --z;
    w = --x - z -- % --y;
    w = - --x - --y - --z;
    w = ++x * y-- - ++y * x--;
    w = x++ * ++y - x * y * z++;
    return (0);
}
```

Listing 2: Incrementation and decrementation operators.

initially:

x = 1, y = 2, z = 3 and w is undefined.

Consider the first assignment to w:

```
w = x ++ + ++ y - --z;
```

x is post-incremented, so its current value, 1, is used.

y is pre-incremented, so its new value, 3, is used.

z is pre-decremented, so its new value, 2, is used.

Therefore,

$w = 1 + 3 - 2 = 2;$

$x = 2;$

$y = 3;$

$z = 2;$

Let's now look at the second assignment to w:

```
w = --x - y-- + z++;
```

x is pre-decremented, so its new value, 1, is used.

y is post-decremented, so its current value, 3, is used.

z is post-incremented, so its current value, 2, is used.

Therefore

$w = 1 - 3 + 2 = 0;$

$x = 1;$

$y = 2;$

$z = 3;$

To check if you have understood our discussion of increment and decrement operators, try the following exercises.

E7) Determine the values given to `x, y, z,` and `w` as a consequences of each of the remaining assignment statements in Program 3.7. Verify your answers by inserting appropriate `printf`()s in the program, and executing it.

E8) Give the output of the following program:

```
/* Program 3.8; File name:unit3-prog8.c*/
#include <stdio.h>
int main()
{
    int x = 10, y = 11, z = 12, w;
    w = ++x - y++;
    printf("w = %d, x = %d, y = %d\n", w, x, y);
    w = ++z % - --y;
    printf("w = %d, z = %d, y = %d\n", w, z, y);
    w = ++y + x++ * z--;
    printf("w = %d, y = %d, x = %d, Z = %d\n",
w, y, x, z);/*Continued*/
    w = ++x % ++y % ++z % w--;
    printf("w = %d, x = %d, y = %d, Z = %d\n",
w, y, x, z);/*Continued*/
    w = ++w / ++x / y--;
    printf("w = %d, x = %d, y = %d\n", w, x, y);
    return (0);
}
```

In the next section, we will explain the concept of lvalue and rvalue of a variable. This will give you better insight into the way that expressions are evaluated in computer.

3.4 LVALUES AND RVALUES

One question that might have occurred to you after reading the foregoing is: what meaning, if any, is to be associated with a statement such as `-n++;`? The answer is, none whatever; in fact such statements will elicit from the compiler vehement protest, and it's instructive to see why. But before that we'll need to take a quick look at the way a computer stores instructions and data, and executes programs.

As you know, the **core** or **main** memory of a computer, also called its **RAM**, (the abbreviation for Random Access Memory) is divided into units known as **words**. We've also seen in the foregoing that an **int** variable, and in many cases even a **short**, usually occupies a word of memory, a **long int** may occupy two words, a **double** may occupy four, and so on.

Depending on the computer a word of memory may be two, four or even eight bytes big. (For the sake of accuracy, however, it must be stated that there have been computers, in fact very popular computers, whose word lengths were not integral multiples of 8 bits: for example, the 36-bit DECSYSTEM-10 and DECSYSTEM-20 machines from Digital Equipment corporation, U.S.A.) In machines with large word sizes the gradation between **short**s, **int**s or **long**s or between **float**s and **double**s may be different than indicated above. As we've seen, each word has associated with it a unique address, which is a positive integer that helps the CPU to access the word. Addresses increase consecutively from the top of memory to its bottom. When a program is compiled and linked, each instruction and each item of data is assigned an address. At execution time instructions and data are found by the **CPU** from these addresses.

The **PC**, or **Program Counter**, is a CPU **register** which holds turn by turn the addresses of successive instructions in a program. In the beginning the PC holds the address of the zeroth instruction of the program. The CPU fetches and then executes the instruction to be found at this address. The PC is meanwhile incremented to the address of the next instruction in the program. In computer jargon, that is where the PC now **points**. Having executed one instruction, the CPU goes back to look up the PC. Therein, in its updated value, it finds the address of the next instruction in the program. This instruction may not necessarily be in the next memory location. It could be at a quite different address, a random distance away from the one just executed (For example, the last statement could have been a **goto** statement, which unconditionally transfers control to a different point in the program; or there may have been a branch to a function subprogram.). The CPU fetches the contents of words addressed by the PC in the same amount of time, whatever their physical locations. That, we recall, is why core memory is called "random access memory". The CPU has random access capability to any and all of the words of memory, no matter what their addresses may be.

note

In time sharing computers with virtual memory this may not be a quite accurate picture. But that's another story.

Program execution proceeds in this way until the last instruction has been processed by the CPU.

Now, the names of program variables, too, are associated with memory addresses; one of the functions of the compiler is to prepare a **symbol table** which contains the memory address of each program variable. Whenever there's a reference to a variable in the program, the CPU is directed to the address from where it can fetch its value. While HLL programmers refer to variables in their programs, the CPU knows only the addresses of memory at which it can find them; in other words when we refer to a variable `n` in a program, the CPU looks at the memory address where it is stored.

The address associated with a program variable in C called its **lvalue**; the contents of that location is its **rvalue**, the quantity which we think of as the value of the variable. The notation corresponds with our picture of memory, a table with two columns in which the left hand column contains memory addresses, and the right hand column the contents of those addresses. The rvalue of a variable may change as program execution proceeds; its lvalue, never. The distinction between lvalues and rvalues becomes sharper if you consider the assignment operation with variables `alpha` and `beta`:

```
alpha = beta;
```

`beta`, on the right hand side of the assignment operator, is the quantity to be found at the address associated with `beta`, i.e. is an rvalue, the contents of the variable `beta`, `alpha`, on the left hand side, is the address at which the contents are altered as a result of the assignment. `alpha` is an lvalue. The assignment operation deposits `beta`'s rvalue at `alpha`'s lvalue. Think of an lvalue as something to which an assignment can be made.

The incrementation and decrementation operators operate on lvalues, and increment or decrement their rvalues. It should now be clear why `-- n ++`, which may be interpreted as `--(n = n + 1)`, is not an allowed operation: `n ++` is an expression, not an lvalue, not an object to which an assignment can be made. For the same reason `(-n) ++` is not correct, but

```
m = -n ++
```

is syntactically valid, since the right to left associativity of the unary operators involved ensures that the expression is evaluated in the order

```
m = - (n ++),
```

i.e.

```
m = -(n), /* use the unincremented n */
n = n + 1./* then increment it */
```

What happens when there are two variable of different sizes in expression? What if the size of a variable is too small to hold the result of an expression? These questions are answered in the next section.

3.5 PROMOTION AND DEMOTION OF VARIABLE TYPES: THE CAST OPERATOR

We've seen that the fundamental types of C variables fall broadly into two categories, integer and floating point, with each category further classed according to size: **char**, **short**, **int**, **long** and **float**, **double** and (in ANSI C) **long double**. C is fairly permissive that it allows the assignment of rvalues of one type to lvalues of a different type: Thus, so far as the syntax of the language is concerned, it's okay to assign a **char** value to an **int** variable, and vice versa. But it's important to be aware of the ramifications of such assignment; for when a "wider" type such as **float** is assigned to a less wide type such as **int** or **char**, one is really trying to do the impossible: Squeeze a big object into a small and in this case a fundamentally "differently shaped" hole: because floating point numbers are stored and operated upon quite differently from integer types. This, in C parlance, is called a **demotion**. Some bits are going to get knocked out. Which will they be? How can the resulting assignment be interpreted? Contrarily, when a less wide type is assigned to a type of larger width, such as in the assignment of a **char** to an **int** (this is called a **promotion**, and it's certainly a less traumatic operation than demotion), how are the extra bits filled: By ones or by zero?

The conversion of data types is subject to a few rules, which we'll explore through the programs below.

i) **The Conversion of Floating point Numbers to Integers.** Let's first look at the conversion of a floating point value to one of an integer type, say **int**. The operative rule is this: If the value is small enough for it to fit into an **int**, the digits after its decimal point are discarded, and the **int** variable is assigned the integer part of the floating point number. But if the value of the **float** variable is negative, then the truncation may be towards zero on some machines, and away from zero on others. For example, if -6.78 is assigned to an **int** variable, the value of the latter may be -6 on some machines, -7 on others. It is not defined in the language as to what may happen if a negative floating value is assigned to an **unsigned int**, or if a floating number or magnitude greater than the limit of **int** is assigned to an **int** variable. The program in Listing 3 was executed in VAX C (where **int**s occupy four bytes, while **short**s are two byte quantities): observe that its results are in accordance with these rules. Observe also that the results of a demotion cannot be predicted.

```
/* Program 3.9; File name: unit3-prog9.c */
#include <stdio.h>
int main()
{
    char one_byte;
    short int two_bytes;
    int small_box;
    double large_value = 12.34e+24;
    float neg_value = -2.78964e+8;
    float small_neg_val = -6.78;
    small_box = large_value;    /* double demoted to int */
```

```
    printf("large_value %e in small_box (4 byte int); %d\n",
large_value, small_box);/*Continued.*/
    two_bytes = neg_value;      /* negative float to short */
    printf("neg_value %e in two_bytes: %d\n",
neg_value, two_bytes);/*Continued*/
    one_byte = small_neg_val;   /*small neg. float to char */
    printf("small_neg_val %f in one_byte: %d\n",
small_neg_val, one_byte);/*Continued*/
    return (0);
}
```

Listing 3: Conversion of floats to ints.

```
/* Program 3.9 :  Output :  */
large_value 1.234000e+25 in small_box (4 byte int):
1073741824
neg_value -2.789640e+8 in two_bytes:  22752
small_neg_val -6.780000 in one_byte:  -6.
```

On a PC with an ANSI C compiler execution was aborted with the error message:

```
Floating point error:  Overflow.
Abnormal program termination
```

The output of the program in Listing 3 should warn you that the results of demotion of type will in general by utterly unpredictable, and should you use such demotions in your programs, do so with the full consciousness that you're doing so!

E9) Experiment with the program in Listing 3 on your system, and study its output using variables of differing widths, types and values.

---

ii) **The conversion of doubles to floats, and vice-versa.**

When a **double** value is converted to a **float**, the value is rounded to the precision of **float**, before truncation occurs. If the result is out of range, the behaviour is undefined. In general, floating point arithmetic is carried out in double precision. When a variable of type **float** is converted to **double**, its fractional part is padded with zeros.

ANSI C

If a **float** value is followed by an `f` or `F`, it's treated as a single precision number, and is NOT converted to **double** in computation; a floating point value followed by an `l` or `L` is treated as **long double**.

```
/* Program 3.10; File name: unit3-prog10.c */
#include <stdio.h>
int main()
{
    double double_width = 12345.0123456789;
    float single_width;
    single_width = double_width;
    printf("double_width = %16.10f, single_width = %f\n",
double_width, single_width);    /*Continued */
    return (0);
}
```

Listing 4: Conversion of doubles to floats and vice-versa.

The output of the program in Listing 4 on our system was:

```
double_width = 12345.0123456789
single_width = 12345.012695
```

The `%16.10f` format conversion specifier for `double_width` ensured that its value was output in a minimum of 16 columns, precise to 10 decimal places. The value of `sinlge_width` was rounded to the precision of **float**. Note that the value is correct to only eight significant figures. See Section 3.6

iii) **The Promotion and Demotion of Integer Types**

When a signed integer type is promoted to an integer type of larger width, the sign is extended to the left. For example, the representation of the two-byte **int** −1 on a two's complement machine is:

1111111111111111 (hex FFFF)

If this value is promoted to **long**, the sign is left-extend, so that its representation as a four byte quantity becomes a collection of 32 one bits:

FFFFFFFF

This is illustrated in the output of the program in Listing 5 below.

Sign extension with one bits does not occur when an **unsigned int** or **char** is assigned to a wider type.

When a integral type is coerced to a less wide **int** type variable the leftmost bits of the wider type are truncated, provided both source and destination variables are **unsigned**. But if the source is a signed quantity, it is accommodated without change in the destination variable if its magnitude is within the limit of the destination's type, otherwise the result is undefined.

The unary **cast** operator (type_cast) of C, introduced in Program 3.11 below is very useful when you need to convert from a variable or expression of one type to one of another. It works like this: suppose `alpha` is an **int** variable, then (**long**) `alpha` is an expression of type **long**, (**float**) `alpha` casts the value of `alpha` to the type **float**, (**short**) `alpha` demotes its value to a `shortint`, (**double**) `alpha` promotes it to type **double**, and so on. The operand of a cast operator may be an rvalue or an expression. The cast operator **does not** alter the rvalue or the type of its operand. The expression (type_cast) rvalue acquires the type enclosed in the parentheses only for the current computation. The cast operator has a priority just below the parentheses operator, and groups from right to left.

```
/* Program 3.11; File name: unit3-prog11.c */
#include <stdio.h>
int main()
{
    short alpha = -5, beta = 5;
    long lambda = 12345678L, mu = -12345678L;
    printf("alpha cast as a long = %ld, \
in hex = %lx\n", (long) alpha, (long) alpha);
    printf("\nNote appropriate sign extension...\n");
    printf("\nbeta cast as a long = %ld, \
in hex = %lx\n", (long) beta, (long) beta);
    printf("\nlambda cast as an \
int = %d, as a char = %d\n", (int) lambda, (char) lambda);
    printf("\nmu cast as an \
int = %d, as a char = %d\n", (int) mu, (char) mu);
    return (0);
}
```

Listing 5: Promotion and demotion of integer types.

Here's the output of program in Listing 5:

```
/* Program 3.11 :  Output:  */
alpha cast as a long = -5, in hex = fffffffb
Note appropriate sign extension...
beta cast as a long = 5, in hex = 5
lambda cast as an int = 24910, as a char = 78
mu cast as an int = -24910, as char = -78
```

Verify that (**int**) `lambda` and (**char**) `lambda` are indeed the contents of the leftmost two bytes, and the leftmost byte, respectively, of the **long int** variable `lambda`.

In the evaluation of an expression involving more than one operand data type conversions obey the following rules in ANSI C:

a) All **char** and **short int** values are elevated to **int**. After this, the following conversions are done operation by operation as follows:

b) If one of the operands in an operation is **long double**, the second operand is also converted to **long double**; otherwise

c) if one of the operands is a **double**, the other operand is converted to **double**; otherwise

d) if one of the operands is a **float**, the second is also converted to **float**; otherwise

e) if any one operand is **unsigned long**, the other operand is also converted to **unsigned long**; otherwise

f) if any one operand is **long**, the other operand is converted to **long**; otherwise

g) if any one operand is of type **unsigned**, the other operand is converted to **unsigned**. Only remaining possibility is that

h) both operands must be **int**s, and that is also the type of the result.

i) If one operand is **long** and the other is an **unsigned int**, and if the value of the **unsigned int** cannot be represented by a **long**, both operands are converted to **unsigned long**.(For example, if you have solved exercise 6 of unit 2, you would have most probably found out that for the gcc compiler on a 32-bit machine, the maximum size of long is 2147483647 and the the maximum size of **unsigned int** is 4294967295, the same as **unsigned long**. Confusing? Remember that the limits in Table 1 of Unit 2 prescribe the **minimum magnitudes** and compilers can allow larger values! So, on a 32-bit machine, the size of **unsigned int** can be larger than **long**! It is for handling this situation that rule viii) is prescribed.)

Let's apply these rules to the evaluation of an expression involving a **float** variable `floatex`, and **int** variable `intex`, a long variable `longex` and a **short** variable `shortex`:

```
floatex * shortex + longex % intex
```

By rule i) `shortex` is promoted to **int**. Then, in the evaluation of

```
floatex * shortex
```

`shortex` is promoted to float by rule iv). In the evaluation of :

```
longex % intex
```

`intex` is promoted to **long** by rule vi), and the expression itself though of magnitude less than `intex`, is of type **long**.

In the computation of the entire expression, which consists of an expression that is **float** and one that is **long**, rule iii) makes the **long** expression a **float**, and the result is also **float**.

Example 2: Write a C programme that solves the following problem: Pradeep's car travels 238 kilometres on 15 litres of petrol. How many kilometres does that average to per litre?

It wouldn't do simply to declare the variables `distance_travelled` and `litres_consumed` simply as **int**s, and perform integer division; for the quotient would be truncated to its integer part, and the value would be off by quite a **bit**, causing avoidable disquietude in Pradeep's mind. But we have now at lest two ways to solve this problem: either to declare one `distance_travelled` or `litres_consumed` a **float** variable, or to declare both as **int**s, and use the cast operator to convert the value of any variable to **float**. The program in Listing 6 on the next page below uses the latter approach.

```
/* Program 3.12; File name:unit3-prog12.c */
#include <stdio.h>
int main()
{
    int distance_travelled = 228, litres_consumed = 15;
    printf("Pradeep's car travels %f kilometers per litre.\n",
(float) distance_travelled / litres_consumed);
    return (0);
}
```

Listing 6: Average petrol consumed per litre.

Here's the output of our little program, a little more satisfying for Pradeep than the value that would have been obtained without using the cast operator:

Pradeep's car travels 15.866667 kilometre per litre.

* * *

Try the following exercise now.

E10) Modify the above program so that it's a little more useful for vehicle owners: the program should prompt for the initial and final readings on the milometer, and the petrol consumed for the distance travelled (which is the difference of these readings), and should then output the average obtained per litre.

We close the section here. In the next section, we will discuss format control in `printf()` and `scanf()` functions in detail.

3.6 FORMAT CONTROL IN THE `printf()` AND `scanf()` FUNCTIONS

We have seen that the `printf()` function can be used to output strings of characters as well as numbers; the fact is that `printf()` is a versatile function which can be used to deliver the formatted output of any numbers of variables or expression of any type. Some of the capabilities of `printf()` are illustrated in the examples below.

When `printf()` is used to output the values of variables or of expressions formed from them, the formatting information is supplied in its first argument, which is a string called the **control string**. The variables to be printed are listed after the control string. Commas are required to separate the control string and the variables listed. A `print()` that's used to output the values of variables looks typically like this

`printf("control string", var_1, var_2, ..., var_n)`

Format specification, i.e. how the variables to be output should be displayed on the screen, is provided in the control string; var_1, var_2,..., var_n are variables or expressions. If there are no variables to be printed, as in our early examples of `printf()`, the control string must not contain any format specifiers. In this case the string itself is output:

```
printf("This string does not refer to a variable list.");
```

Let's now suppose that you wish to print the values of two **int** variables, `x` and `y`, which are respectively 27 and 117. Then the following statement will do:

```
printf("%d %d \n", x, y);
```

The first `%d`—called a **format conversion specifier**—in the control string instructs that the variable `x` will be printed as a **decimal integer** (The `d` stands for decimal.) in a **field**

(i.e. consecutive spaces on a line) as wide as may be necessary to accommodate it. Becasue x is a two-digits number, it will be printed in field of width two units.

The second %d refers to the variable y, which, being a number of three digits, will be printed in a field three units wide.

Note that there are two variables to be printed, x and y, and there are two %d's in the control string, one for each variable to be output.

The output in the present case will be :

```
27 117
```

Observe that there is but a single space separating the values of x and y in the output. This is because we had placed a single space between the %d's in the control string. Suppose instead that we choos the following format specification, which included the escape sequence for a tab between the %d's:

```
printf("\%d\t\%d\m", x, y);
```

The values of x and y will then be separated by a tab in the output:

```
27 117
```

Any characters other than the format conversion characters such as %d, etc. within the control string are reproduced directly in the output. So spaces, tabs, escape sequences and text in the control string appear without change on the monitor. Therefore the output of:

```
printf("\t The value of x = \%d,\n\t while y = \%d.", x, y);
```

will be

```
The value of x = 27,
while y = 117.
```

printf() may similarly be used to output expressions; suppose that x and y have the values 27 and 117 as before. Then the statement:

```
printf("\%d \%d\n", y / x, y, \% x);
```

gives the following output:

```
4 9
```

It is very important that there should be a one-one ordered correspondence between each format conversion specifier in the control string, and the list of variables or expressions which follows it. If you wish to print twenty **int** variables: var_1, var_2,..., var_20, there should be twenty occurrences of the %d format conversion specifier within the control string, the first assoicated with the first value to be output, the second with the second, and so on.

Because the % character has a syntactic value in the control string, two consecutive occurrence % % are required in the string in order to print it as a literal in the output. Let's look at the example in Listing 7:

```
/* Program 3.13; File name: unit3-prog13.c */
#include <stdio.h>
int main()
{
    int chance_of_rain = 70;
    printf("There\'sa %d%% chance of rain today.\n",
chance_of_rain);/*Continued*/
```

```
    return (0);
}
```

Listing 7: Printing % character in `printf()`

The two % %'s after the %d are output as a single % character. Verify this by executing the program.

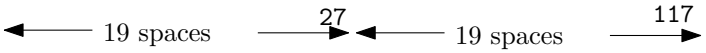
It is often desirable to print a value right or left justified within a specified field width, say w. (For example, when a Rupee amount is to be printed on a cheque, common sense dictates it should be printed left justified in the designated field.) %wd right justifies the value to be output within a field of width w, %-wd left justifies within the field.

To print the values of x and y (which we assume as before are 27 and 117 respectively) right justified in fields of width 19, use:

```
printf("The values of x and y are:\n\n\%19d\%19d\n", x, y);
```

Output:

```
The values of x and y are:
```

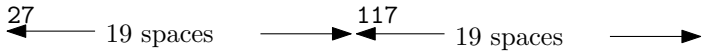


To print the values of x and y left justified in fields of width 19, we write:

```
printf("The values of x and y are:\n\n\%-19d\%-19d\n", x, y);
```

Output:

```
The values of x and y are:
```



If the first digit of the field width is zero, the field is stuffed with leading or trailing zeroes, depending on whether the justification is towards the right or towards the left.

If a field width is not specified, or if the width specified is insufficient to accommodate all the digits in the number, %d still outputs the correct value. Therefore we will frequently not specify a field width with a format specification.

The %0 and %x format conversion characters are used to output octal and hexadecimal integers repectively.

The %u specification is used to print unsigend **int**s, and %d, %l0, and %lx specifiers are used to output **long**s as decimal, octal or hex integers respectively.

E11) Do you agree with the output of the program below?

```
/* Program 3.14; File name: unit3-prog14.c */
#include <stdio.h>
int main()
{
    typedef int debt;
    debt amount = 7;
    printf("If I give you Rs. %05d\n", amount);
    printf("you will owe me Rs. %-05\n", amount);
    return (0);
}
```

E12) State the output of the following programs:

```
/* Program 3.15; File name: unit3-prog15.c */
#include <stdio.h>
int main()
{
    int birth_hour = 23, birth_minute = 47;
    int birth_day = 17, birth_month = 3, birth_year = 1981;
    printf("\t Abhishek was born on %02d-%02d-%4d\n",
birth_day,birth_month, birth_year);/*Continued*/
    printf("at %02d:%02d hours\n", birth_hour, birth_minute);
    printf("in the city of Jodhpur, Rajasthan.\n");
    return (0);
}

/* Program 3.16; File name: unit3-prog16.c */
#include <stdio.h>
int main()
{
    int amount = 64;
    /* the 64,000 Dollar question */
        printf("Q: How can you make 64000 dollars from 64?\n");
    printf("A: By clever printing: %-05d", amount);
    return (0);
}
/* Program 3.17; File name unit3-prog17.c */
#include <stdio.h>
int main()
{
    int x = 27, y = 117;
    printf("y %% x = %0 in octal \
and %x in hex", y % x, y % x);
    return (0);
}
/* Program 3.18; File name: unit3-prog18.c */
#include <stdio.h>
int main()
{
    unsigned cheque = 54321;
    long time = 1234567L;       /* seconds */
    printf("I\'ve waited a long time (%1d seconds)\n", time);
    printf("for my cheque (for Rs. %u/-), and now\n", cheque);
    printf("I find it\'s unsigned!\n");
    return (0);
}
```

3.6.1 Floating point Numbers and Character Strings with printf()

The output of **float**s and **double**s is accomplished by the `%f` specifier where an optional `w.d` is used to specify a field width `w` and number of digits, `d`, after the decimal point. `%e` is used to output single or double precision floating point numbers in scientific notation (e.g. 1.234e5) and `%g` prints a float value either as `%e` or as `%f`, whichever is shorter. The width and precision specification are optional, as before. A precision specification rounds off the value that is output to the number of places stated.

C strings are output directly via the `printf()`; however the `%s` conversion character with an optional `w.d` specification can be used to format the output. In this case `d` number of characters from the beginning of the string will be printed in a field of width `w`.

Example 3: Let us look at an example program. The program in Listing 8 on the next page prints the values of the Sine function for 0°. 10°, . . ., 90°. Another thing you

would notice is that we have converted degrees to radians; This is because the Sin() function takes input in terms of radians. Here we have used the **for** loop. You may be able to figure out how it works by compiling the program, running it and looking at the output. You may also look at Unit 6 for a description of the **for** loop.

```
/*Program to print the values of the Sine Function.*/
/*File name:unit-3-sinetable.c*/
#include<stdio.h>
#include<math.h>
#define PI 3.1416
int main()
{
    int i;
    printf("%-7.5s\t%-14.12s\n\n", "THETA", "SIN(THETA)");
    for (i = 0; i <= 90; i += 10)
        printf("%-7d\t%-14.4f\n", i, sin((PI * i) / 180.0));
    return (0);
}
```

Listing 8: A program to print the values of the Sine function.

You may experiment with different format control statements and look at the output.

* * *

Try the following exercises to check your understanding of format control of floating point and string variables.

E13) Execute the following program to verify the rules stated above for the output of floating point variables"'

```
/* Program 3.19; File name: unit3-prog19.c */
#include <stdio.h>
int main()
{
    double pi = 3.14159265;
    printf("%15f\n", pi);
    printf("%15.12f\n", pi);
    printf("%-15.12f\n", pi);
    printf("%15.4f\n", pi);
    printf("%15.0f\n", pi);
    printf("%15.3g\n", pi);
    printf("%15g\n", pi);
    printf("%15.4e\n", pi);
    printf("%15e\n", pi);
    return (0);
}
```

E14) What does the following program print?

```
/* Program 3.20; File name: unit3-prog20.c */
#include <stdio.h>
int main()
{
    printf("%-40.24s", "Left justified printing.\n");
    printf("%-40.20s", "Left justified printing.\n");
    printf("%-40.16s", "Left justified printing.\n");
    printf("%-40.12s", "Left justified printing.\n");
    printf("%-40.8s", "Left justified printing.\n");
    printf("%-40.4s", "Left justified printing.\n");
    printf("%-40.0s", "Left justified printing.\n");
    printf("%40.25s", "Right justified printing.\n");
    printf("%40.20s", "Right justified printing.\n");
    printf("%40.15s", "Right justified printing.\n");
```

```
    printf("%40.10s", "Right justified printing.\n");
    printf("%40.5s", "Right justified printing.\n");
    printf("%40.0s", "Right justified printing.\n");
    printf("%40.0s", "Right justified printing.\n");
    return (0);
}
```

We close this section here. In the next section, we will summarise our discussion in this unit.

3.7 SUMMARY

In this unit, we have studied the following:

1) The precedence and associativity of operators: `()`, `[]`, `->` and `.` have the highest precedence and associate left to right. Unary operators like `!`, `~`, `++` , `--`, `+` have the next highest precedence and associate right to left. The operators `*` and `/` have the next highest precedence and associate left to right.
2) C expressions are syntactically valid combinations of operators and operands that compute to a value determined by the priority and associativity of the operators.
3) Incrementation and decrementation operators and abbreviated assignment operators `+=`, `-=`, `*=` and `/=`.
4) The format modifier `%wd` prints a integer right justified in a field of width `w` and the modifier `%-wd` prints an integer left justified in a field of width `w`.
5) The format modifier `%w.d` prints a floating point number in a field of width `w` with `d` decimat digits after the decimal point.
6) The format modifier `%w.ds` prints `d` characters from the beginning of a string in a field of width `w`.

3.8 SOLUTIONS/ANSWERS

E1) We have dealt with one case, a is negative and b is positive. Please add code for the remaining 3 cases.

```
/*Prog 3.0. Filename:unit3-prog0.c*/
#include<stdio.h>
int main()
{
    int a, b, ans1, ans2;
    /* a is negative and b is positive. */
    a = -10;
    b = 5;
    ans1 = a % b;
    ans2 = a - (a / b) * b;
    printf("The difference is %d\n", ans1 - ans2);
    return 0;
}
```

E2) In the first set the values are x = 7, y = 62 and z = 1. The others are easy. In the case of z, 2*3*4 is 24 and 24%15 is 1. 1%13 is again 1. You can similarly check the other values. Here is the program with `printf()` statements added. You can check your answers by compiling and running the program.

```
/* Program3.1; File name:unit3-ansex2.c */
#include<stdio.h>
int main()
{
    int x, y, z;
    x = 2 + 3 - 4 + 5 - (6 - 7);
    y = 2 * 33 + 4 * (5 - 6);
    z = 2 * 3 * 4 / 15 % 13;
    printf("x is %d, y is %d, z is %d\n",x,y,z);
    x = 2 * 3 * 4 / (15 % 13);
    y = 2 * 3 * (4 / 15 % 13);
    z = 2 + 33 % 5 / 4;
    printf("x is %d, y is %d,  z is %d\n",x,y,z);
    x = 2 + 33 % -5 / 4;
    y = 2 - 33 % -5 / -4;
    z = -2 * -3 / -4 % -5;
    printf("x is %d, y is %d,  z is %d\n",x,y,z);
    x = 50 % (5 * (16 % 12 * (17 / 3)));
    y = -2 * -3 % -4 / -5 - 6 + -7;
    z = 8 / 4 / 2 * 2 * 4 * 8 % 13 % 7 % 3;
    printf("x is %d, y is %d,  z is %d",x,y,z);
return (0);
}
```

E3) Since x = 3, y = 5 and z = 7 the first expression

`w = x % y + y % x - z % x - x % z;`

is `3%5+5%3-7%3-3%7`. Since % has higher priority, the value is $3+2-1-3=1$. You can similarly find the value w in the other cases. Do not forget to check your answer by compiling and running the program.

E4) Since * and / have higher priority and they group left to right, first `-12*-13` is computed, which is 156. Since `156/14=11`, we get `-1+2+11=12`. You can similarly find the value of the other expressions.

E5) In program 3.5, after executing the statement `w +=x`, the value of `w` is $9+3=12$ and this is printed by the first `printf()`. After executing the statement `w -= y`, the value is $12-5=7$. You can similarly work out the other values. In program 3.6, `s %= t`, the value of `s` remains 9, because `9&10=9`. This is the value of the expression `s %= t`. Since `w` is -11, the value of the expression `w /= (s %= t)`, i.e. `-11/9` is -1. So, the value of the expression `(w /= (s %= t))` is -1. Since `z` is 11, the value of the expression `z *= (w /= (s %= t))` is $11\times-1=-11$. So, `y -= (z *= (w /= (s %= t)))` is $y-(-11)=y+11=4$. So, the value of the expression `x += (y -= (z *= (w /= (s %= t))))` is $7+4=11$. Similarly, you can find the value of the other expression.

E6) In the last `printf()`, $x+z-y*y$ is $100+300-400=0$ and `(y - z % w) * x)` is `(20-300%40)*100)` is 0 since 300%40 is 20. So, the values printed will be 0, 0.

E7) After evaluating the first 2 statements, the values are `x` = 1, `y` = 2, `z` = 3. While evaluating the expression `w = x++ * ++y % ++z`, `x` is post-incremented and the value `x` = 1 is used. The variable `y` is pre-incremented and the value `y` = 3 is used. The variable `z` is pre-incremented and `z` = 4 is used. So, `w = 1 * 3 % 4` is 3. After evaluating the expression, the value of `x`, `y` and `z` are `x` = 2, `y` = 3, `z` = 4.

In the expression `w = --x / --y * --z`, `x` is pre-decremented and the value `x` = 1 is used. The variable `y` is pre-decremented and the value `y` = 2 is

used. The variable z is pre-decremented and the value z = 3 is used. So, w = 1 / 2 * 3 = 0. After evaluating the expression, the value of x, y and z are x = 1, y = 2, z = 3. Similarly you can check the output of the remaining programs.

E8) The valus x = 11 and y = 11 are used in evaluating the expression w = ++x - y++. The value of w is 0 and the value of x and y are 11 and 12 after the statement is exectued. Similarly, you can find the other values. Check your answers by compiling and running the problem.

E10)
```
#include<stdio.h>
int main()
{
    int dist_travelled, pet_consumed, ini_reading, fin_reading;
    printf("Enter the initial reading...:\n");
    scanf("%d", &ini_reading);
    printf("\nEnter the final reading...:\n");
    scanf("%d", &fin_reading);
    printf("\nEnter the petrol consumed..:\n");
    scanf("%d", &pet_consumed);
    dist_travelled = fin_reading - ini_reading;
    printf("Pradeep's car travels %f kilometers per litre.\n",
(float) dist_travelled / pet_consumed); /*Continued */
    return (0);
}
```