UNIT 3 OPERATORS AND EXPRESSIONS IN C

Structure

3.0 INTRODUCTION

This Unit introduces some basic concepts of the C language: the priorities and associative properties of several operators, and their use in the evaluation of expressions; Ivalues and rvalues, and the promotion and demotion of variable types. This Unit also explains the distinction between a memory address and its contents (Ivalues and rvalues); these concepts., are important pre-requisites to understanding arrays and pointers.

3.1 OBJECTIVES

After working through this Unit you should be able to

- write and evaluate complex C expressions, built with die arithmetic operators of C
- learn about the order of precedence among operators, and the direction in which each associates, and
-appreciate how these properties help decide the sequence in which the various parts of a C expression are
evaluated.

3.2 ELEMENTARY ARITHMETIC OPERATIONS AND OPERATORS

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 liner 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.

Similary, 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 emphasise 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 to 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 die assignment operator to be lower than the priori1ties of all 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 wil 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 statement:

```
    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:

first, the multiplication of x by y, with the result stored as an intermediate quantity;

second, the multiplication of this quantity by z;

third, 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 below. [Not all of these operators may be available in non-ANSI implementations.]

Table 3.1
Precedence and Associativity of Operators

Operators	Associativity
( ) [ ] -- .	L to R
! ~ ++ -- + - * & (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
? :	L to R
= += -= *= /= %= &= ^= |= <<= =	R to L
,	L to R

The parentheses 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 3.1 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 it, right. Since the unary operators have a higher priority than the assignment operator, in the statement:

```
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 3.1.

As we learned 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 one integer by another). They are respectively +, -, / and % (refer to Programs 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 / operator when used with int like variables, and the % operator _ which can only be used with int like variables_ correspond precisely to the div and mod operators of Pascal.

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 thin 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?

Examples:

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; die 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 assigned the value 1.

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

The parentheses ensure that 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, ie. 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.

Check Your Progress 1

Question 1: 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.

Question 2: Find the value that is assigned to the variables x, y and z when the following program is executed:

```
/* Program 3.1 */
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;
    }
```

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

Question 3: Give the output of the following program:

```
/* Program 3.2 */
#include <stdio.h
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 + (x + y) / z;
     /* contd. */
     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);
    }
```

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 semicolon 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. Consequently, 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 gets 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))

Each parenthetical expression is 3.thus x is 3.

printf () prints expressions just as easily as it prints variables. This is illustrated in Program 3.3 below:

```
/*                    Program                    3.3                    */
#include                                                          <stdio.h
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))))))));
             }
```

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 gel., the value z % y, which is 1, finally x becomes 2 * 1, and this is also the value output.

3.3.1 Abbreviated Assignment Expressions:

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

n = n + 5;

C allows a shorter form for such statements:
n += 5;

Assignment expressions 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 +=, -=, *=, 1= and %= is the same as that of the assignment operator.

Check Your Progress 2

Question 1: What does the following program print:

```
/*                    Program                    3.4                    */
#include                                                          <stdio.h
main ( )
         {
          printf   ("%d\n",   -   1   +   2   -   12   *   -   13   /   -   4);
          printf   ("%d\n",   -   1   %   -   2   +   12   %   -   13   %   -   4);
          /*                              contd                              */
          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));
   }
```

Question 2: State the output of the following programs: (Hint: A single printf () may be used to print the values of several variables or expressions. 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 %ds) and the variables in the argument list. Note the commas after the control string, and between identifiers.)

```
/* Program 3.5 */
#include<stdio.h
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);
   }
/* Program 3.6 */
#include <stdio.h
main ( )
   {
    int x = 7, y = -7, z = 1 1, 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);
   }
```

Question 3: Given that x, y, z and w are ints 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 ("\t%td\n\t%d\n\t%d\n\t%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 */
```

Insofar as their operand n is concerned, the pre- and post- incrementation or decrenientation 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 a "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 unincremented 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 inremented 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-incremented 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 of the assignment statements in the program below, to see how x, y, z and w get their values:

```
/*              Program              3.7              */
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++;
        }
```

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.

Check Your Progress 3

Question l: Determine the values given to x, y, z, and w as a consequence 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.

Question 2: Give the output of the following program:

```
/* Program 3.8 */
#include <stdio.h
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);
     w = ++ x % ++ y % ++ z % w- -;
     printf ('w = %d, x = %d, y = %d, z = %dn", w, x, y, z");
     w = ++ w / ++ x / y- -.,
     printf ('w = %d, x = %d, y = %d\n", w. x, y);
    }
```

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 ++;? lie 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 DECSYSTEhl-10 and DECSYSTEM-20 machines from Digital Equipment Corporation, U. S. A.)

In machines with large word sizes the the gradation between shorts, ints, or longs, or between floats and doubles may he 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 die 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, die 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 is in C called its Ivalue; 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 Ivalue, never. The distinction between Ivalues 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. alptia is an Ivalue. The assignment operation deposits beta's rvalue at alpha's Ivalue. Think of an Ivalu c 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 Ivalue, 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 */
```

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 Ivalues 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 assignments; 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 zeros?

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

(ii) 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 of magnitude greater than the limit of int is assigned to an int variable. The program below was executed in VAX C (where ints occupy four bytes, while shorts are two byte quantities): observe that its result., are in accordance with these rules. Observe also that the, results of a demotion cannot be predicted.

```
/*                    Program                    3.9                    */
#include                                                          <stdio.h
main ( )
        {
         char                       one                          byte,
         short                      int                      two-bytes;
         int                        small_                            box;
         double        large_       value         =          12.34e+24;
         float            neg_yalue            =            -2.78964e+8;
         float             small_neg_val            =            -6.78;
         small_box    =    largp_value;    /*   double   demoted   to   int   */
         printf ("large_value %e in small_box (4 byte int): %d\n", large-value, small_box);
         two-bytes    =    neg-value;    /*    negative    float    to    short    */
         printf   ("neg-value   %e   in   two-bytes:   %ft",   neg_value,   two_bytes);
         one_byte    =    small_neg_val;    /*    small    neg.    float    to    char    */
         prientf  ("small_neg_val  %f  in  one_byte:  %d\n",  small-neg_val,  one_byte);
        }
```

```
/* 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 Program 3.9 should warn you that the results of demotion of type will in general be utterly unpredictable, and should you use such demotions in your programs, do so with the full consciousness that you're doing so!

Exercise: Experiment with Program 3.9 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. [Note on 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 computations; a floating point value followed by an 1 or L is treated as long double.]

```
/* Program 3. 10 */
#include <stdio.h
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);
    }
```

The output of program 3.10 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 single_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 (hexFFFF)
```

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

```
FFFFFFFF
```

This is illustrated in the output of Program 3.11 below.

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

When an 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_past) 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 short int, (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            */
#include                                   <stdio.h
main ( )
        {
         short   alpha   =   -5,   beta   =   5;
         long   lambda   =   12345678L,   mu   =   -12345678L;
         printf ("alpha cast as a long = %ld, in hex = %1x\,n", (long) alpha, (long) alpha);
         printf   ("\n   Note   appropriate   sign   extension   ...   \n");
         printf ("\nbeta cast as a long = %ld, in hex = %1x\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);
        }
        Here's the output of Program 3.11:
                /* 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 a 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 expressions involving more than one operand data type conversions obey the following rules:

i. float operands are converted to double;
ii. chars or shorts (signed or unsigned) are converted to ints (signed or unsigned);
iii. if any one operand is double, the other operand is also converted to double, and that is the type of the result; otherwise
iv. if any one operand is long, the other operand is treated as long, and that is the type of the result; otherwise
v. if any one operand is of type unsigned, the other operand is converted to unsigned, and that is the type of the result; otherwise, the only remaining possibility is that
vi. both operands must be ints, and that is also the type of the result.

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

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

The variables to be printed are listed after the control string. Commas are required to separate the control string and the variables listed. A printf () 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:

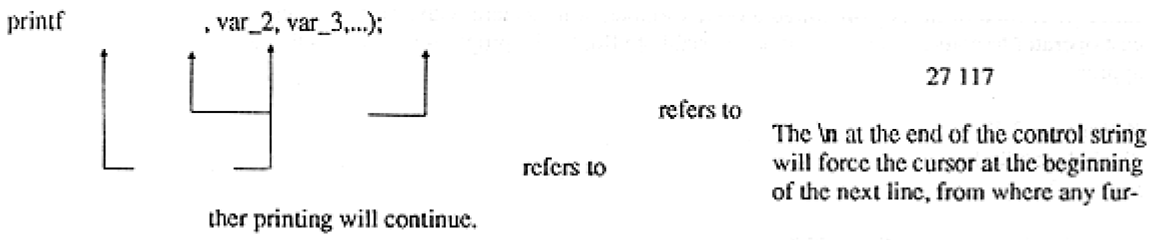
printf (" %d %d\n", x, y);

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. Because x is a two-digit 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

The \n at the end of the control string will force the cursor at the beginning of the next line, from where any further printing will continue.

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 choose the following format specification, which includes the escape sequence for a tab between the %d's:

printf (" %d\t%d\n", 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\twhile 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 for- mat conversion sperifier 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 specirier within the control string, the first associated with die 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 occurren- ces % % are required in the string in order to print it as a literal in the output. Let's look at an example:

```
/*                    Program                    3.13                    */
#include                          <stdio                                 .h
main ( )
                {
                          int          chance-of-rain          =          70;
                 printf ("There Y sa %d%% chance of rain today. \n", chance of rain):
                }
```

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 riglit 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:

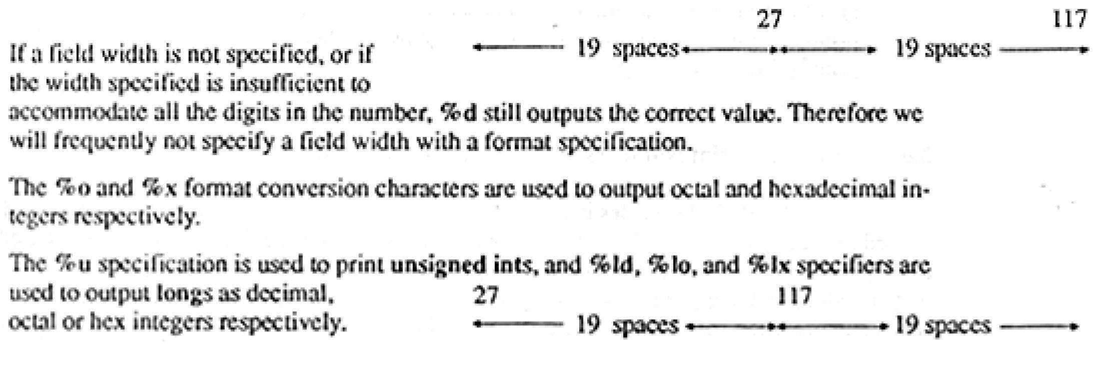
printf ("The values of x and y are:\n\n%-19d%-19d\n". x. y);

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

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.



27
117
19 spaces
19 spaces
27
117
19 spaces
19 spaces

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 %o and %x format conversion characters are used to output octal and hexadecimal in- tegers respectively.

The %u specification is used to print **unsigned ints**, and **%ld**, **%lo**, and **%lx** specifiers are used to output longs as decimal, octal or hex integers respectively.

Check Your Progress 4

Question 1: Do you agree with the output of the program below?

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

Question 2: State the output of the following programs:

```
/* Program 3. 15 */
#include <stdio.h
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);
         printf ("at %02d:%02d hours\n",birth_hour, birth_minute);
         printf ("in the city of Jodhpur, Rajastban.\n");
        }
/* Program 3.16 */
#include <stdio.h ,
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);
        }
/* Program 3.17 */
#include <stdio.h
main ( )
        {
         int x = 27, y = 117;
         printf ("y %% x = %0 in octal and %x in hex", y % x, y % x);
        }
/* Program 3. 1 8 */
#include <stdio.h
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");
        }
```

3.6.1 Floating point Numbers and Character Strings with printf ( )

The output of floats and doubles 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 0; 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.

Check Your Progress 5

Question l:. Execute the following program to verify the rules stated above for the output of floating point variables:

```
/*            Program            3.19            */
#include                                  <stdio.h
main ( )
     {
      double      pi      =      3.14159265;
      printf      ("%      15f\n",      pi);
      printf      ("%      15.12f\n',      pi);
      printf      ("%-15.12M",      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);
     }
```

Question 2: What does the following program print?

```
/*            Program            3.20            */
#include                                  <stdio.h
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      prinfing.\n");
      printf      ("%-40.0s",      "      Left      justified      printing.'\n');
      printf      ("%40.25s",      "Right      justified      printing.\n");
      printf      ("%40.20s",      "Right      justified      prinfing.\n");
      printf      ("%40.15s",      "Right      justified      prinfing.\n');
      printf      ("%40.10s",      "Right      justified      printing.\n");
      printf      ("%40.5s",      "Right      justified      printing.\n");
      printf      ("%40.0s",      "Right      jusdfied      printing.\,n"),,
      printf      ("%40.0s","      Right      justified      printing.\n");
     }
```

3.7 SUMMARY

C expressions are syntactically valid combinations of operators and operands that compute to a value determined by the priority and associativity of the operators. The type of an expression is generally the type of its "widest" operand. In an assignment expression the final result has the type of the variable to which it is assigned. This may result in a promotion or a demotion, by which values are converted to wider or less wide types; a demotion may cause significance to be lost unpredictably.

The control string of a printf (can be made to print numeric or character data according to a variety of available options.

UNIT 4: DECISION STRUCTURES IN 'C'

Structure

4.0 INTRODUCTION

This Unit introduces the if () and if () -else statements of C. In the if () statement the parentheses which follow the keyword if contain a special type of C expression, called a Boolean expression, which evaluates to one of two values _ true or false, but nothing in between. Booleans are black or white, there are no shades of grey! Typically Booleans compare the values of two variables or expressions, through relational operators; these are used to construct expressions that evaluate to true or false. The result of a Boolean expression can thus be used to guide the flow of control in a program to alternative courses of action: do such if the Boolean is true, such other if false.

Consider, for example, a program that a small bank might use for its book-keeping, to permit a customer to encash a cheque if the amount she wishes to withdraw is less than her balance:

```
if (amount_requested < current_balance)
  printf ("issue cash, Rs.%-d\n", amount_requested);
else
  printf ("Sorry, amount requested exceeds balance.\n").,
```

The expression:

```
amount_requested < current_balance
```

is an example of a Boolean expression. Obviously, it can only be true or false. Either the value of amount_requested is less than current_balance, or it's not. If it's less, issue cash, else not.

Suppose x and y are int variables that have the respective values 3 and 5: then the Boolean expression:

```
x > y        /* Is x greater than y ? */
```

has the value false, while

```
x < y                  /* Is x
```

has the value true, and

```
x == y       /* Is x equal to y ? */
```

is false. , and == are relational operators. Boolean expressions such as

amount__requested < current_balance

are generally (but in C not necessarily, as we shall find), constructed from the relational operators. C has three other relational operators:

```
>=    /* greater than or equal to */
>=    /* less than or equal to /
! =   /* not equal to */
```

In addition there is the unary negation operator, ! , which negates the expression on its right. The priorities and associativides of the relational operators are listed in Table 3. 1.

In an if ( ) statement the parentheses containing the Boolean are followed by a statement called the object statement of the if (); the object statement is executed if the parenthetical condition is true, and is ignored otherwise. In the last example, the statement:

```
printf ("issue cash, Rs.%-d\n", amount_requested);
```

is the object statement of the associated if ().

The if () statement imparts decision-making power to a program. The object statement is executed if and only if the value of the Boolean expression so warrants:

```
if (this condition is true)

execute this object statement, otherwise ignore it;
```

Example:

```
if ( X Y )

    printf ("x is greater than y.\n");
```

A variant of the if () statement is the if ()-else statement, which has an object statement with the if () part, and another with the else part: if the parenthetical condition evaluates to true, the object of the if () part is executed; otherwise the object of the else part is executed: if there's money in the account, allow a withdrawal; else not.

Example:

```
if (x + 3
  printf ("x plus 3 is less than z times w.\n");
else
  printf ("x plus 3 is not less than z times w.\,n");
```

Note from Table 3.1 that the priority of

```
x + 3 < z *w
```

is evaluated in the order **z * w, x + 3, x + 3**

We use if () and if ()-else statements in our daily lives. Here are two examples from mine:

Example:

```
if (it's a pleasant day)
        printf ("I think I\'11 relax today.\n");
```

Example:

```
if (it's a pleasant day)
  printf ("I think I\'ll relax today.\n");
else          /* because it's not a pleasant day */
  printf ("Sorry, I won\'t be able to work today.\n");
```

(You'll be unlikely to catch me working most days of the week, whether the days are pleasant or not!)

Such statements provide the computer, as they do us, with the almost magical power to make decisions; and bring into sharp focus the chiefest difference between a hand held calculator and a computer: a calculator cannot execute an if () statement; it cannot decide between alternative courses of action, depending on the result of a computation: of course, neither can it store a program in its memory.

4.1 OBJECTIVES

In this Unit you will learn to

- use the six Boolean operators of C
- use the operator for negation
- use the logical connectives for and and or
- create programs with branches
- use the (disreputable) goto statement

4.2 BOOLEAN OPERATORS AND EXPRESSIONS

Computers can demonstrate amazing feats of what appears at first sight to be "reasoning behaviour." (Whether this apparently intelligent behaviour is in fact akin to a form of human being-like reasoning is the subject of intense debate among Artificial Intelligence experts). In substantial measure the "intelligence" that computers seem to exhibit is due to the fact that the internal two-valued logic of the CPU is isomorphic to the Boolean logic used almost instinctively by intelligent human beings.

For a first glimpse of Boolean logic, consider the following problem:

Amal will join the Good Men's Club if either Bimal is chosen its President or Saran its Vice-President.

Bimal will accept Presidentship of the Club only if Ashok is made Vice-President and Rajni not made the Club Secretary.

Rajni will not join the Club if Suman opts for membership.

Saran decides not to join the club, Ashok is made Vice- president, and Suman becomes a club member.

What will Amal do?

Logic problems such as this one are most easily solved by the techniques of an algebra called Boolean Algebra, in honour of its inventor, George Boole. The "variables" of Boolean Algebra are expressions which can have one of precisely two values: they can be either true or false. (For this discussion we will use the words expression and statement interchangeably.)

Thus, it's either true or false that " Amal will join the Good Men's Club". There is no third alternative. The problem is to determine the truth value of this statement, given the truth values of the statements with which it is connected.

From the information supplied, the statement acquires the value true if Bimal is made President of the Club or if Saran becomes its Vice-President. In either case, Amal will join the Club. The or here is an example of a logical connective. It connects two Boolean variables (or constants).

Each of the statements:

Bimal is President
Saran is Vice-President

has a value that is true or false.

For Amal to join the Club, it's enough that one of the statements be true; it's immaterial then that the other statement is true or false. Amal will join the Club either if it's true that Bimal is President, or that Saran is Vice-President, or if both statements are true. The only case in which Amal will not join the Club is when both statements are false. Therefore, the or logical connective between Boolean expressions satisfies the following truth table:

true	or	true	=	true
true	or	false	=	true
false	or	true	=	true
false	or	false	=	false

The OR truth table

Further we are given that Bimal accepts Presidentship only if Ashok is made Vice-President and Rajni not made the Club Secretary. The statement:

Bimal accepts Presidentship

is true if the statement:

Ashok is made Vice President

and simultaneously the statement:

Rajni is not the Club Secretary

is true. For Bimal to accept Club Presidentship it's necessary that both the connected statements be true. It is not sufficient that Ashok is made Vice President. Nor is it enough that Rajni is not the Club Secretary. If either statement is false, Bimal refuses Presidentship. It is clear therefore that the and logical connective between Boolean values satisfies the following truth table:

true	and	true	=	true
true	and	false	=	false
false	and	true	=	false
false	and	false	=	false

The AND truth table

Again, the statement:

Rajni is not the Club Secretary

is the negation of :

Rajni is the Club Secretary

If the truth value of the latter is true, the truth value of the former is false. Thus not, which negates the truth value of the expression on which it operates, has the following truth table:

not	true	=	false
not	false	=	true

The NOT truth table

The problem further specifies that Rajni will not join the Club, (and so, incidentally, cannot become it, Secretary) if Suman becomes a member.

That is:

```
      if (Suman is a member)
            then
                  (Rajni is not the Secretary);
```

or

```
if (Suman is a member)
    then not
        (Rajni is the Secretary);
```

or

if((Suman is a member)= **true**)
then
((Rajni is the Secretary) = **false**).

The following additional data are provided:

Suman is a member = **true**
Ashok is Club Vice-President = **true**

thus:

Rajni is the Secretary = **not (true) = false.**

Now, the statement of the problem may be written:

if ((Ashok is Club Vice-President)
and not (Rajni is the Secretary))
then (Bimal accepts Presidentship);

i.e.

if **((true) and not (false))**
then (Bimal accepts Presidentship)

i.e.

if **((true) and (true))**
then (Bimal accepts Presidentship).

From the truth table for the and connective we find that the parenthetical condition in the last if evaluates to true. Bimal is President of the Club. Amal joins it.

You might be curious about the connection between such problems of logic and digital computers. The fact is that all of the hardware of computers and hand-held calculators_their digital circuitry_ is based entirely on the laws of Boolean algebra. Every single computation that's done on a computer or calculator translates at its most basic to the evaluation of complex Boolean functions. In fact it was from the need for the exploration of the principles of computer design, that a fresh lease of life has been given to the study of Boolean Algebra. When it was invented more than a hundred and fifty years ago, and for many years afterwards, Boolean Algebra was no more than a mathematical curiosity, and there were absolutely no practical applications of it! But here's one:

Suppose we were to design an intelligent robot that would have the capability to cross a road in the midst of traffic. At first glance this might seem to be an example of intelligent behaviour, yet the process of crossing a road is purely algorithmic, and involves the computation of several Booleans, as we'll see below.

Obviously the robot has to be equipped with cameras for input devices, that can receive traffic flow data. Its output devices would be wheels driven by motors whose speed is controlled by the CPU.

Clearly the CPU must be programmed to process the images it receives from the input. One quantity that it should be able to estimate from its input is the width of an approaching vehicle, whether it may be car, scooter-rickshaw, bus or motorcycle. This is important: it takes a longer time to cross the path of a bus, than of a motorcycle.

Apart from estimating size, the CPU should be able to gauge not only the distance of the approaching vehicle, but also its speed. This information will help the robot answer such a question as: will I be able to cross clear of the oncoming vehicle, at my maximum speed, in the time that it will take to reach me? The technology and the calculation involved are not too difficult: the robot could throw an acoustic pulse at the vehicle, and by determining the time elapsed between sending the pulse and receiving its echo compute the distance to the vehicle. By sending another pulse a short interval later, the CPU can deduce the distance travelled by the vehicle over the interval, and thus its speed.

Given its speed and distance, the robot's CPU can estimate the time an approaching vehicle will take to reach it. Then, with knowledge of the width of the approaching vehicle, and the width of the road, the CPU must evaluate Booleans such as:

```
if      (the time the vehicle will take to reach me, say t1
        is greater than the time I will take to
        cross its path, say t2, at my maximum speed)

then

        I'll move forward,

else

        I'll wait until the road is clear.
```

The Boolean here is:

t l t2.

If it returns the value true, the robot goes ahead to cross the road, else it waits:

```
if (tl t2)
        I'll cross the road;
else
        I'll wait until the road is clear.
```

Obviously more complex Booleans must be evaluated if the robot is to be able to cope with more than one vehicle at a time, and if vehicles are allowed to approach from either direction. Suppose there are two vehicles approaching the robot, one from the left and one from its right.

The Booleans it must evaluate before beginning to cross are:

```
if ((the time taken by the vehicle on my left to reach me, tl,
is greater than the time I will take to cross its path, t2)
            and
        (the time taken by the vehicle on my right to reach me, t3
        is greater than the time 1 will take to cross the road, t4))
                I'll cross the full width of the road;
else
        if ((I can cross past the vehicle on my left)
                    and not
                (I can cross past the vehicle on my right))
                        I'll cross past the vehicle on my left and pause in the
                        middle of the road for the vehicle on my right to
                        pass;
        else
        I'll wait for the vehicle on my left to pass me by;
```

In a programming language these Booleans may be written:

```
if ((tl t2) and (t3 t4))
    I'll cross the road;
else
    if ((tl t2) and not (t3 t4))
            I'll cross over to the middle and pause
            for the vehicle on my right to pass me by;
else
            I'll wait for the vehicle on my left to pass me by;
```

The next time you cross a road, contemplate with wonder on the enormous complexity of the computations your brain instinctively performs, when you use it to carry out so common an undertaking!

Realise that each else statement above can be associated only with the if physically nearest it This is also true of analogous statements in C programs.

4.2.1 Boolean Operators

To see how Boolean relationships are written in C. suppose x and y are program variables The following Boolean Operators and logical connectives are available:

operator	**Usage**	**Meaning**
!	! (Boolean)	negates Boolean expression
	x y	x grater than y
=	x = y	x greater than or equal to y
<	x < y	x leer than y
<=	x <= y	x less than or equal to y
==	x == y	x equal to y
!=	x ! = y	x not equal to y
&&	x y &&x !=0	logical and of two Booleans
||	x 5 || x= =Y	logical or Of two Booleans

The unary negation Operator ! has a Priority just below the parentheses operator: it negates the Boolean expression which follows it. Like all unary operators, it groups from right to left

The four binary operators , =, and <= each have equal priority. The remaining two operators == and !=, which also require two operands, have equal priority, but lower than that of the first four.

&&, the logical and connective has a priority lower than the six relational operators above, but exceeding that of the logical or connective, ll. All these operators group from left to right. Each of the arithmetic operators has a higher priority than the Boolean operators; but the assignrnent operator has a lower precedence.

We've seen that in contrast to Pascal, C does not have variables Of type Boolean. However, it uses two rules to get by without such variables:

i. Every expression, **including one involving Boolean operators**, has a value.
ii. An expression that has a non-zero value is true. while one that evaluates to zero is false.

There's a third rule that you should know, because it is often the source of puzzling exam questions (though it's not of much practical use):

iii. The value of a Boolean expression that is true is 1, while the value of an expression that is false is 0.

Thus the value of 5 3 is 1, of 5 .

Check Your Progress 1

Question 1: What does the following program print?

```
/*                    Program                    4.1                    */
#include                                                          <stdio.h
main ( )
         {
          printf              ("%d\n",              5              3);
          printf          ("%d\n",          3          <          5);
          printf  ("%d  %d\n",  (5  3)  &&  (3  <  5),  (5  3)  ll  (3  <  5));
          printf  ("%d  %d\n",  (1  0)  *  (1  <  0),  1  0  *  1  <  0);
         }
```

Question 2: Give the output of the following program:

```
/*                Program                4.2                */
#include                                            <stdio.h
main ( )
     {
     int x = 5, y = 0;
     if (X y)
          printf ("x y\n");
     if (X < y)
          printf ("x < y\n");
     if (! (X == Y))
          printf ("x != y");
     }
```

Question 3: In Program 4.2 above are the parentheses around x == y in the last if 0 statement necessary? Remove them, compile and execute your program again, and explain your result

Question 4: In Program 4. 1, are the parentheses around the expressions 5 3 and 3

In Program 4.3 below, the Boolean of the first if (), x y, is false; so z does not get the value 1. The second Boolean, x < y, is true. z becomes 2. The Boolean of the next if () x == y, is false, so its object statement, x = z = 3 is not executed. Because x is still 2, and y is 3, the Boolean in the next if (), x != y, is true, and y becomes 0. Finally, since x is 2 (or true) !x is false (or 0), so !x == y is true, and z gets the value 0.

```
/*                Program                4.3                */
#include                                            <stdio.h
main ( )
       {
         int    x    =    2,    y    =    3,    z    =    0;
         if (x y)
           z = 1;
         if (x < y)
           z = 2;
         if (x == y)
           x = z = 3;
         printf   ("x   =   %d,   y=%d,   z=%d\n",   x,   y,   z);
         if (x != y)
           y = 0;
         if (! x == y)
           z=0;
         printf   ("x   =   %d,   y=%d,   z   =   %d\n",   x,   y,   z);
         }
              /* Program 4.3: Output: */
              x    =    2,    y    =    3,    z    =    2
              x = 2, y = 0, z = 0
```

4.3 THE goto STATEMENT

The next program illustrates the use of the goto statement, which transfers control unconditionally to another place in the program that is marked by a label:

```
printf        ("Whoopee        ...        Holidays!!!\n"),
goto                                                 Goa;
.
.    /*    intervening    code    is    skipped    */
.
Goa: printf ("See me at the beach !!%");
```

Label names are subject to the same rules as are identifiers: they are built from the alphabetical characters, digits and the underscore, with the proviso that the first character must not be a digit. In a program a label name is followed by a colon.

The goto statement is held in deepest contempt by the votaries of structured programming, primarily because programs with gotos are extremely difficult to verify for correctness, and to debug. A program with a sprinkling of gotos quickly begins to resemble a plateful of noodles, and to determine the general flow of control in such a program can lead to a great deal of frustration.

Program 4.4, which uses if ()s and gotos, is concerned with a popular (and still unsolved) problem variously known as the 3*N + 1 problem or the Collatz problem, after Lothar Collatz who is credited with having invented it. The problem is easy to state: think of a whole number greater than zero; if it's odd, multiply it by 3 and add 1.; if it's even, divide it by 2. Repeat these steps for the resulting number; the series of numbers generated eventually terminates in ... 4, 2, 1, no matter what number you may have begun with! For example, let's choose 7; because it's odd, we multiply it by 3 and add 1, to obtain 22; because 22 is even, we divide by 2 and arrive at the next number, 11. The progression continues until 4, 2, 1 are reached:

7,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1.

A Proof that every number will generate a series terminating in 4, 2, 1, is still not available so the next best thing to do is to test the conjecture for as many numbers as possible: computers make this easy to do, and by the end of the 'eighties the Collatz conjecture had been verified for all values upto 1000000000000.

Program 4.4 computes a quantity called cycle_ length, which stores the number of iterations needed for a given int input, before the sequence terminates at 1. Its f-irst goto statement transfers control right back to a label named begin_again if the initial input is less than or equal to zero.

However, if the initial input was 1, there is nothing to do, and control is transferred directly to the label Finished, without changing the value of cycle-length.

Any other value of input is successively transformed until the last iteration yields 1, at which time the program terminates.

```
/* Program 4.4 */
#include <stdio.h
main ( )
       {
        int input, cycle-length = 0;
        begin_again:
        printf ("Enter a positive trial number.");
        if (input <= 0) /* only a positive input permitted */
              goto begin_again;
        iterate:
        if (input == 1)
              goto finished;
        /* contd. */


         if (input % 2 == 1) /* input was odd */
              goto odd_input;
         if (input % 2 == 0) /* input was even */
              goto even_input;
        odd_input:
        input = input * 3 + 1;
        cycle_length ++;
        goto iterate; /* Is this statement necessary ? */
        even_input:
        input /= 2;
        cycle_length ++;
        goto iterate;
```

```
  finished:
  printf ("1appeared as the terminating digit after %d iterations", cycle_length);
}
```

Question 1: In Program 4.4 above, is the statement preceding the comment:

```
/* Is this statement necessary? */
```

necessary?

Question 2: Follow through the steps of Program 4.4 to compute manually the value of cycle_length for input = 27. Do you agree that the goto should be banned?

Question 3: One major defect with Program 4.4 is the presence of two if () statements where one would suffice:

```
if (input % 2 == 1) etc;
```

and

```
if (input % 2 == 0) etc;
```

Obviously, input can be only either even or odd. If it's odd, the second if () is redundant; if it's even, the second if () is not required. Rewrite the program using the first if () alone.

Question 4: Write a program to read three positive integers from the keyboard, into the ints first, second and third, and to print their values in descending order. You are not allowed to define any other variables in your program.

Though the use of the goto statement is justifiably considered reprehensible, particularly when it is used as a "quick fix" to solve a problem (because it most likely will have repercussions on other parts of the program), there are instances in which a non-linear flow of control is called for, and the use of the goto may then be appropriate. Error handling constitutes just such a case. One of the qualities of a robust program is that it should check for errors in its input and be able to cope with any, by for example, prompting for fresh input or by terminating with a relevant error message. When it encounter any error, it should invoke a diagnostics routine that outputs information about the problem, and should then terminate. But this error reporting function, like all functions, can only return control to the point from which it was called, which may be nested deeply inside loops, or other program structures. The best solution may then be to transfer control via a goto to a label preceding statements for methodical program termination:

```
    if (error)
      goto report_error;
            .
            .
            .
report_error:
    print relevant data;
    terminate program;
```

The goto can also be used with advantage in search procedures that may involve several nested loops, in which the value sought is determined in the innermost loop; then the statement:

```
if (found)
        goto print_value;
```

is preferable to working outwards one by one through the enveloping loops. In the unit on loops we shall see just such a use of the goto.

4.4 THE if ( ) STATEMENT

The if () statement is perhaps the most powerful statement of any programming language. It's powerful because its Boolean argument helps the computer make choices between alternative courses of action:

```
if (true)
        this statement will be executed;
```

```
if (false)
        this statement will be skipped;
```

As we saw in the last program, two consecutive equal signs == test for equality of the value of the expressions which occur on either side:

```
if (input== 1)
        Printf ("input equals %d\n", input);
```

But this calls for caution: for probably the commonest mistake that beginning C programmers make is to write a single "equals sign". =, when they mean to test one value for equality with another:

```
if (input = 1) etc.;
```

The problem with this statement is that the parenthetical expression:

input = 1

happens to be an assignment expression, with value 1. Now a non- zero value inside the parentheses of an if () is considered true (by rule (ii)), so the statement that depends on such an if () is automatically executed; a side effect is that this assignment sets the value of input to 1, no matter what the value that may earlier have been assigned to it! There's no question, then, of comparing input with 1: input gets the value 1!

Program 4.5 brings this out clearly, and deserves to be studied with some care. It depends upon the fact that an expression has a value, which may be zero or different from zero. All non-zero values, positive or negative, are true. Therefore, any expression, even an assignment expression such as x = 0, x = 5, etc. may be regarded as a Boolean in an if () statement.

```
/*                    Program                    4.5                    */
#include                                                          <stdio.h
main ( )
        {
         int                    x                    =                    0;
         if (x= 0)
                 Printf ("x is 0, x == 0 is true, but this statement will not be output.\n-);
        if (x != 0)
                 prientf (,x is 0, x !=0 is false, so this statement will not be output.\n,,);
        if (x = 5)
                 Printf (" Can you believe it, x is indeed %d!\n", x);
        }
```

Look at the first if () statement in the program.. it's Boolean argument has the value (), since () is the value of the assignment expression:

x = 0.

Since a Boolean that evaluates to 0 has the value false, the object statement:

```
printf ("x is 0, x == 0 is true, but this statement will not be output.\n");
```

will not be executed.

Attend now to the second if () statement. Its Boolean x != 0 is clearly false, since the value of x is 0 (observe the argument of the preceding ir (). The object statement is again skipped.

The third if () is interesting:

```
if                    (x                    =                    5)
   printf ("Can you believe it, x is indeed %d!\n", x);
```

The Boolean expression:

x = 5

happens also to be an assignment expression, with value 5. Now, because, a Boolean with a non-zero value is true, the printf () is executed, and prints 5 for x, which is its current value. This if () statement accomplishes two goals: it assigns a value to a variable; and it tests whether its Boolean argument is true or raise. The simplest expression is a variable name by itself, the value of the variable being the value of the expression. Suppose that a statement must be executed only if the value of the variable is non-zero. The following ir ()s are equivalent:

```
if (x != 0) statement;
if (x) statement;
```

Another mistake frequently encountered in C programs is the inadvertent placement of a semicolon immediately after the parenthetical Boolean in an if () statement as shown below:

```
if (input== 5);
        prinif ("The value of input is %d\n", input);
```

There are now two statements here instead of the single one that was actually intended: the first is an if 0 statement with a null object statement__the semicolon immediately after the Boolean; the second is the printf () statement which stands by itself, and which does not depend for its execution on the preceding if () statement; it will be executed whether or not the parenthetical Boolean is true. There are no syntactical errors, a program containing these statements will compile perfectly, but will execute the printf () no matter what value is assigned to input , 5 or any other. It is the null statement, the semi-colon, that is executed (or not executed) depending upon the truth value of the Boolean. The object statement of an if is the single statement that follows it, and it may quite possibly be a null statement if the logic so warrants.

If there is a series of statements that must be executed when the Boolean of an if () is true, they must be enclosed in braces:

```
if (Boolean condition is true)
    {
            all the statements upto
            the terminating right
            brace will be executed;
    }
```

A set of statements enclosed in braces is called a compound statement; compound statements may occur wherever the syntax allows a single statement. (Recall, Pascal encloses compound statements in a begin-end pair.) Compound statements help make program logic clearer; they also serve to dispense with gotos. The program below, a revised version of Program 4.4, has fewer gotos because it uses compound statements.

```
/* Program 4.6., Revised Version of Iltogram 4.4 */
#include <stdio.h
main ()
        {
         int input, cycle_length = 0;
         begin_again:
         printf ("Enter a positive trial number");
         scanf ("%d", &input);
         if   (input < = 0)
                goto begin _ again;
         iterate:
         if    (input == 1)
                goto finished;
         if    (input % 2 == 1 )
                {
                  input = input* 3 + 1;
                  cycle_length ++;
                }
         input /= 2;
         cycle_length ++;
         goto iterate;
         finished:
         printf(" 1 appeared as the terminating digit after %d iterations", cycle-length);
        }
```

The goal of structured programming is to create goto-less programs. While compound object statements help to an extent in achieving this objective, it is the loop structures of C_he for (;;) loop, the while () loop and the do-while () loop _ that are indispensable tools for accomplishing this goal. Loops will be studied in depth in the next unit

The object statement of an if () statement may be another if () statement. The nested if () is reached only if the Boolean of the covering if () is true. In the program below, the condition inside the first if (), x = 2 is true; its object statement will therefore be executed. As it happens, that is another if () of which the Boolean is also true. The third if (), the object of the second, will consequently be reached, and since its Boolean is also true, z is assigned the value 4. It's a simple matter now to predict the output from the succeeding statements.

```
/*                  Program                 4.7                 */
#include                                                  <stdio.h
main ( )
       {
        int    x    =    2,    y    =    3,    z    =    100;
        if                                                 (x=2)
          if          (y          <          =              3)
            if                    (y                        x)
              z                   =                         4;
        printf         ("z         =         %d\n",          z);
        x              =                     z=            100;
        if          (x          (y          =              99))
          if                                        (y(x=x-2))
            z                    =                         99;
        printf ('x = %d, y = %d, z = %d\n", x, y, z);
        if                        (y                        X)
         if (2*zx+y)
              {
                x                    =                      2:
                y                    =                      3;
                z                    =                      4;
              }
        printf ("x = %d, y = %d, z = %d\n", x, y, z);
                        if         (x          <             0)
          if                      (y                         0)
            printf   ("Will   this   statement   be   executed   ?\n");
       }
```

Question 1: State the Output of Program 4.7

4.5 THE if ( ) - else STATEMENT

The if () else statement provides for two-way branching:

```
if (condition is true)
   this statement is executed;
else
   this statement is executed;
```

In other words, if the Boolean in an if () - else statement evaluates to true, the object statement of the if () is executed; the object statement of the else is skipped. Contrarily, if the Boolean evaluates to false, the object statement of the if () is ignored, that of the else is executed. Where there are only two choices provided in a program the else statement is redundant : it's useful only when the number of choices is larger than two, as in Program 4.10 below.

The following simple program illustrates the usage of the if () - else statement. It prints two numbers it scans from the keyboard in descending order.

```
/*                    Program                    4.8                    */
#include                                                           <stdio.h
main ( )
         {
          int          num          I,          num2,          bigger;
          printf    ("Enter    two    numbers.\n    The    first    is    ?");
          scanf              ("%d",              &num              1);
          printf          ("The          second          is          ?");
          scanf                    ("%d",                    &num2);
          printf   ("In   descending   order   the   numbers   are:   ");
          if          (num          1          =          num2)
           printf          ("%d          %d\n",          numl,          num2);
          else
           printf("%d%d\n",num2,numl);
         }
```

Caution: Pay attention to the semicolon preceding the keyword else: a semicolon must terrninate the object statement of an if () whether or not it is followed by an else statement. (Contrast this with Pascal, where the if-then-else is a continuous statement, unbroken by a semicolon.) An else statement must be preceded by, and be associated with, an if () statement; it may not occur by itself. But an if () statement doesn't necessarily require an else statement

Program 4.9 below distinguishes between people who have fun programming in C, and those who don't (inexplicably, there are some who don't).

```
/*                              Progrim                              4.9
*/#include                                                         <stdio.h
main ( )
         {
          char                                                 response;
              printf    ("Do    you    like    programming    in    C?\n");
             printf   ("You\'re   supposed   to   type   y,   then   press   :   ");
                                   getchar                    (response);
                         if  (response          ==          'y')
           printf    ("Thank    you,    thank    you,    for    your    kindness!\n");
          if              (response              ==              'n')
           printf          (""Be          off          with          you          !!\n");
         }
```

But what, you might ask, if the person who executes the program types in neither a ' y ', nor an 'n', but a quite different and irrelevant character? Users are entitled to their foibles, you know! Then neither of the printf ()s is executed. To handle such a situation we use again if ()-else statement so that it can cover a larger number of possibilities _ **the object statement of an else may itself be an if () - else statement, and so on for as many times as may be needed.**

```
            if              (response              ==              'y')
              printf    ("Thank    you,    thank    you,    for    your    kindness!\n");
            else

                  if   (response == 'n')
                        printf ("Be off with you!!\n");
                    else
                        printf ("We assume you don\t want to commit yourself !");
/*               Program               4.               10               */
#include                                                           <stdio.h
main ( )
         {
          char                                                 response;
          printf    ("Do    you    like    Programming    in    C    ?\n");
          printf ("You\'re supposed to type y, then press :");
          getchar (response);
          if (response == 'y')
           printf ("Thank you, thank you, for your kindness!\n");
          else
           if (response == 'n')
            printf ("Be off with you!!\n");
           else
            printf("We assume you don\'twant to commit yourself!");
         }
```

With these improvements Program 4. 10 is almost OK as it stands, but not quite: for what if the user types an uppercase 'Y' when the means to say "Yep, C's great!" or an uppercase 'N' when she means to say, "No thank you, C isn't quite my cup of tea!" The uppercase 'Y' will fail the test of equality against the lowercase 'y', as wil I the uppercase 'N' against the lowercase 'n' (these chars are different because their ASCII decimal equivalents are different). Our program should be able to accept both 'Y' and 'y' for yes, both 'N' and 'n' for no. Here's one way of solving the problem:

```
/* Program 4. 11 */
#include <stdio.h
main ( )
         {
          char response;
          printf ("Do you like Programming in C.?\n");
          printf ("You\'re supposed to type y, then press :")
          scanf ("%c", &response);
          if ((response == 'y') ll (response == ' Y '))
           printf ("Thank you, thank you, for your kindness!\n");
          else
           if ((response == 'n') ll (response == 'N'))
            printf ("Be off with you!!\n");
           else
            printf(" We assume you don\'t want to commit yourself!");
         }
```

The two vertical bars ll represent the logical or operator. Since the result of an or is true if any of the connected Booleans is true, the program gives the appropriate output if it scans 'y' or'Y', or 'n'or'N'in the input.

C stops evaluating a Boolean expression as soon as it determines its truth value. Therefore in the expression:

```
      response == 'y' || response == 'Y'
```

if it is true that response == 'y', the entire expression becomes true, and its latter half, response == 'Y' is not scanned.

The if () - else statement is an aid to structured programming: it helps make programs easier to understand and debug. Consider yet another version of our program for the Collatz problem:

```
/* Program 4.12 */
#include <stdio.h
main ( )
         {
          int input, cycle_length = 0;
          begin _again:
          printf ("Enter a positive trial number
          scanf ("%d", &input);
          if (input < = 0)
            goto begin _again;
          iterate:
          if (input = 1)
            goto finished;
          else
            if (input % 2 == 1) /* input was odd*/
          input = input * 3 + 1;
            else
              input /= 2 ;
          cycle_length ++;
          goto iterate;
          finished:
          printf ("1 appeared as the terminatingdigitafter%d iterations", cycle_length);
         }
```

Question 1: In Program 4.1 1, are the inner parentheses in: if ((response == 'y') ll (response == 'Y')) required? Why or why not?

Question 2: State the output of program 4.13: Hint: An if () associates with the else physically nearest it.

```
/* Program 4.13 */
#include <stdio.h
main ( )
          {
          int x = 2, y 3, z = 100;
          if (x < 2)
            if (y = 3)
              if (!(y X))
                   z = 4;
               else z = 5;
               else z = 6;
               else z = 7;
               printf ("z = %d\n', z);
               x = z = 100;
               if (x (y = 99))
                    if (y(x=x+2))
                       z = 99;
               else z = 101;
               else z = 102;
               printf ("x = %d, y = %d, z = %d\n", x, y, z);
       /* contd. */
          if (y X)
            if (2*z x + y )
                   {
                      x = 2;
                      y = 3;
                      z = 4;
                   }
                else
                   {
                      x = 4;
                      y = 3;
                      z = 2;
                   }
                else
                  x = y = z = 5;
                printf ("x = %d, y = %d, z = %d\n', x, y, z);
               }
```

The last program sheds some light on the importance of proper indentation: see how difficult it is to follow the logic of statements written thus:

```
if (x, < 2) if (y = 3) if (!(y x))
z = 4; else z = 5; else z = 6; else z = 7;
```

Let's look at Program 4.14 to explore the precedence relationshops of the Boolean and arithmetic operators.

```
/* Program 4.14 */
#include <stdio.h
main ( )
        {
         int alpha = 10,
         beta = 5, gamma = 1, delta;
         delta alpha beta gamma;
         printf ("%d\n", delta);
         delta = alpha / beta == ++ gamma;
         printf ("%d\n", delta);
         delta = alpha ll beta ll gamma ++ - 2;
         printf ("%d\n", delta);
         delta = alpha && beta && gamma ++ - 2;
         printf ("%d\,n", delta);
         delta = alpha / beta = beta / gamma;
         printf ("%d\n", delta);
         delta == alpha/2 == beta;
         printf ("%d\n", delta);
         delta = ++ delta ll (delta = 5);
         printf ("%d\n", delta);
         delta = 5 ll ++ delta;
         printf ("%d\n", delta);
         (delta = 5) ll ++ delta;
         printf ("%d\,n", delta);
         delta == delta ++;
         printf ("%d\n", delta);
        }
```

In the first assignment to delta:

delta = alpha> beta> gamma;

the assignment operator has the lowest precedence. Since groups from left to right, alpha beta is evaluated first, and has the value true, which by rule (iii) is 1. The next expression to be computed is therefore:

1 >gamma

which evaluates to false, and has the value (). delta gets the value ().

Consider the second assignment to delta:

delta = alpha / beta == ++ gamma;

The pre-incremented value of gamma (2) is to be compared against the quotient alpha / beta (also 2). Note that the quotient is available before the comparison is done. The Boolean is true, so delta is 1.

In the assignment:

delta = alpha || beta || gamma ++ - 2;

it is important to realise that a Boolean is evaluated only to the extent necessary to determine its truth value. Here alpha is non-zero and is therefore true, so the entire expression is true; gamma is not post-incremented; delta is 1. This deserves comparison with the next assignment to delta, where the ORs are replaced by ANDS:

delta = alpha && beta && gamma ++ - 2;

Post-incrementation of gamma implies that its last value (2) must he used in the evaluation of delta. alpha is 10 and beta is 5, so alpha && beta

is true, but

gamma ++ - 2

is false, the entire expression is false, and delta is zero. Note that gamma is post-incremented when the expression for delta involves &&s; each sub-expression must be evaluated to determine the truth value of delta.

In the assignment:

delta = alpha / beta = beta / gamma;

the quotients alpha / beta and beta / gamma are computed before the, inequality is tested. With the current values of alpha, beta and gamma, the inequality is true and delta is 1.

The next statement:

delta == alpha / 2 = beta.,

is not an assignment statement. No rvalue is modified by it, and this may cause a warning to be issued at compile time. delta is still 1.

Check Your Progress 5

Question 1: Calculate the values that delta gets in the remaining assignment statements, and verify your results by executing the program.

Question 2: Write a program that accepts three numbers and decides:

1. whether these can be the lengths of the sides of a triangle;
2. if they form an equilateral, isosceles or scalene triangle
3. if they form a right-angled triangle

Question 3: Write a program which accepts two numbers, and determines the following:

1. the first is positive, negative or zero
2. the second is positive, negative or zero
3. the first is even or odd
4. the second is even or odd
5. the first is larger than the second, in absolute value
6. if the first is exactly divisible by the second

Hint: Divisibility of the first by the second is determinable only if the second number is different from zero.

Question 4: Write a program that accepts the coefficients a, b and c of a quadratic equation, and determines whether its roots are real, complex or equal.

Question 5: Write a Program to determine whether the value of a year input to it, such as 1900, or 1996, or 2000, represents a leap year or not. Your program should prompt the user to enter a value, and it should then output one of. "Leap year' or "Not a leap year". Hint: A leap year value is evenly divisible by 4. However, a value that is divisible by 100 is not a leap year, unless it is also divisible by 400.

Question 6: Give the output of the following programs:

```
/*                  Program                  4.15                  */
#include                                                    <stdio.h
main ( )
        {
         int       x=       10,y       =       5,z       =       0;
         x                %=              y-=z==x                  <y;
         printf   ("x   =   %d,   y   =   %d,   z   =   %d\n",   x,   y,   z);
         x *= y += z = x == (y /= 2);
         printf ("x = %d, y = %d, z = %d\n", x, y, z);
         x = y- - && - -z ll (X + y - z);
         printf ("x = %d, y = %d, z = %d\n", x, y, z);
         x =y++ z ++ ++ x /12;
         x =--x <=-y z ll x && y;
         printf "x = %d, y = %d, z = %d\n", x, y, z);
        }
  /* Program 4.16 */
  #define RICH savings_deposit 1000000L
  #define MOBILE cars 1
  #define GOOD_JOB working_hours 4 && salary 5000
  #define VERY_HAPPY RICH && MOBILE && GOOD_JOB
  #define HAPPY MOBILE && GOOD_ JOB
  #define FAIRLY_HAPPY RICH ll GOOD_JOB
  #include <stdio.h
  main ( )
        {
         long savings,_deposit = 1234567L;
         int cars = 2, working_hours = 5, salary = 6000;
         if (VERY_HAPPY)
           printf ("Thank you very much, God!\n");
         if (HAPPY)
           printf ("Thank you, God!\n');
         if (FAIRLY_HAPPY)
           prinif ("Things could be better, God\n");
        }
```

Question 7: Is!(VERY_HAPPY) ==!VERY _ HAPPY?

Give reasons for your answer.

Question 8: Write a program that accepts a digit from the keyboard, and displays it in English. So, if the user types 7, the program responds with "seven", if she types 4, the program responds with "four", etc.

Question 9: Write a program that gets a character such as +, -, * or /, then scans two numbers, and then yields the result of the operation denoted by the character. In the division of one value by another, your program should behave intelligently if the denominator == 0.

4.6 SUMMARY

The truth tables for AND, OR and NOT, the six relational operators of C, and the logical connectives && and ll can be used with the if () and the if () - else statements to create programs with branches. Boolean expressions have values which are considered true if they are different from zero, false otherwise. For this reason C does not have a datum type akin to Pascal's Boolean. A Boolean is evaluated only to the extent necessary to ascertain its truth value. The object statements of an if () or if () - else may be simple or compound statements. Programs with branches enable the computer to choose between alternative courses of action, giving it the power to make decisions. The goto statement allows immediate transfer of control to a statement marked by a label; its use can lead to undisciplined programming.

UNIT 5: CONTROL STRUCTURES - I

Structure

5.0 INTRODUCTION

The rationale of structured programming is to create goto-less programs. While compound Object statements help to an extent in achieving this objective, it is the loop structures of C_the for (;;) loop, the while () loop and the do - while () loop _ that are indispensable tools for accomplishing this goal. Loops are required wherever a set of statements must be executed a number of times. Consider, for example, the method you employ for looking up a word in the dictionary, let's say the word ringent. It's unlikely that you will be able to open the dictionary at the right page and locate the word in its line the first time; so you open it towards the middle, where let's assume you find the word nexus.

Since n comes before r, the latter half of the book must now be searched. Thus the process is repeated: you open the book towards the middle of the second half, at approximately the three-quarters point, where again assume you find scrub. s comes after r, so-ringent must occur, if at all it does, between the pages which contain nexus and scrub; you therefore bisect this interval, and suppose you now find the the dictionary open at the page containing puddle. p comes before r, so ringent must be between puddle and scrub. You therefore divide this interval, and thus continue to bisect the ensuing sub-intervals, until you find the word, or discover that it's not defined in the dictionary. Note that, beginning with the entire dictionary as the original search interval, you have executed the interval creation and search instructions repetitively: in computer jargon, you've executed a loop. Here's the algorithm:

```
interval = entire dictionary;
while (word has not been found)
        {
         bisect interval;
         of the two intervals created, determine likely_interval;
         interval = likely_interval;
         if (interval 0)
           continue;
         else
           break out from the loop;
        }
```

This search procedure is appropriately enough called the **bisection method**; note that it can work only if the entries are sorted in ascending order, such as words in a dictionary, or names in a telephone directory. Each execution of the loop instructions _ the statements inside the curly braces _ is an **iteration**, and is Performed only if the Boolean at the beginning of the loop is **true**: that is, if the word is still not found. Like Pascal, C has three loops, two of which, the **while** () and the **do - while** () are introduced in this until. In the **for** (;;) and the while () loop, the Boolean is evaluated **before** the loop is entered; the loop is skipped if the Boolean is **false**. In the **do - while** () loop, the Boolean sits at the end of the loop statements which are executed anyway the first time around oven before the Boolean has been examined. If the Boolean is then found to be **true**, the loop is iterated, else not. So a **do- while** () loop differs from the other two in this important particular.. it is invariably executed at least once.

The break statement forces an immediate exit from the loop. The continue statement forces control to be transferred back to a re-evaluation of the Boolean controlling the loop. If it is true, the loop is again executed, else the loop is exited. The break and the continue statements may he used in each of the loop structures of C; the continue statement cannot be used outside a loop; the only other place, besides loops, where the break statement may be used is in the switch - case - default statement, discussed later in this unit; break and continue are keywords, and may not be used in any contexts other than those stated.

Sometimes it is desirable to be able to transfer control to one of several possible execution paths in a program, to the exclusion of all the others. In driving your car in New Delhi, you may find yourself at a roundabout of several roads radiating outwards, of which only one will (optimally) take you where you want to go. You have to decide from the possibilities available, and choose the appropriate path. In doing so, you will be executing the analogue of a C decision structure called the switch - case - default statement; this directs the flow of control to one of the several cases listed in the statement, depending upon the value taken by an integer variable, called the switch variable. Just as a driver at a roundabout chooses one and only one radial road, control flows in a switch statement along a unique path (among several that may be listed) determined by the switch variable.

This Unit introduces loops and the switch - case - default decision structure.

5.1 OBJECTIVES

After reading this unit you will be able to

- write programs using the **while ( )** and **do - while** ( )loops
- use the ternary or **if - then - else** operator
- use the **switch - case - default** decision structure for multiple-way branchings.

5.2 The do - while ( ) and while ( ) Loops

For a quick look at the while ( ) and do - while ( ) loops, let's rewrite Program 4.12 _ our program for the Collatz problem __ without using the goto statement. This will clear the way for us to see how Booleans control loops. And you will learn how loops make the goto unnecessary. For your convenience, we reproduce that program (with its many faults) below:

```
/ * Program 4.12 */
# include <stdio.h
main ( )
     {
          int input, cycle_length = 0;
          begin _again:
          printf ("Enter a positive trial number.");
          scanf ("%d", &input);
          if (input <= 0)
            goto begin_again;
          iterate:
          if (input== 1)
            goto finished;
          else if (input % 2 = 1) /* input was odd */
                   input = input * 3 + 1;
               else
                 input/= 2;
               cycle_length ++ 1
               goto iterate;
               finished:
               printf (" 1 appeared as the terminating digit after %d iterations",
               cycle_length);
```

```
                    }
```

Note that the first goto in Program 4.12 transfers control over and over again to the label **begin_again** while the value scanned for the variable named input is negative or zero. But if the value entered for input is positive, the preceding if () statement ensures that the goto is not executed.

```
begin_again :
          printf          ("Enter          a          positive          trial          number:          ");
          scanf                                   ("%d",                                   &input);
          if  (input < = 0)
                    goto begin_again;
```

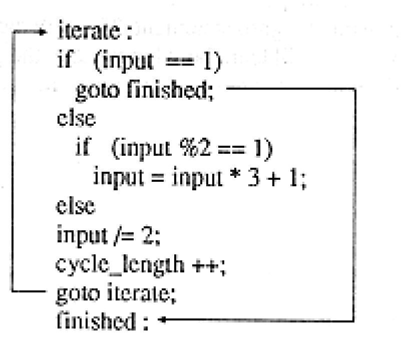
In effect we have the following situation:

```
                    do
                              {
                                        printf     ("Enter     a     positive     trial     number     ");
                                        scanf ("%d", &input);
                              }
                    while (input <= 0);.
```

In the **do - while** () loop, the parentheses following the keyword **while** contain a Boolean expression. For so long as the Boolean expression is **true**, (yet at least once) the statement sandwiched between the keyword **do** and the **while** () (which may be a compound statement if need be), is repeatedly executed. But if, at the end of any iteration, the Boolean is found be **false**, the loop is exited. Because the truth or falsity of the Boolean is established in the last statement of the loop, and therefore only after the loop has been entered, the intervening statement is necessarily executed at least once. In the present instance, if the value scanned for input exceeds zero, the loop is exited after the first execution of its statements. But if input is less than or equal to zero, the terminating Boolean remains **true**, and the program asks for another value for **input**. One acceptable value for input must be got; if it's right the first time, the loop is exited immediately; if it's not, the loop is executed over and over until a usable value for input has been entered. In using the do - while () loop, we've rid ourselves of the first goto. See Program 5. 1.

Moral: Use the **do - while** ( ) loop wherever a statement (or a set of statements, in which case they must be enclosed in braces) is to be executed at least once, and possibly oftener.

Furthermore, in Program 4.12, the statements between the labels iterate and finished are repeatedly executed as long as the value of input is different from 1.



```
iterate :
if   (input == 1)
  goto finished;
else
  if   (input %2 == 1)
    input = input * 3 + 1;
else
input /= 2;
cycle_length ++;
goto iterate;
finished :
```

This just the sort of situation where the while () loop is appropriate:

```
                              while   { (/* Boolean is true, i.e. */ input != 1)
                                     {
                                       execute          these          statements;
                                     }
```

TO start with, the loop is entered only if the Boolean is true, i.e. if input != 1. If the Boolean is false, control skips the loop. That takes care of the statements:

```
          if  (input==                                                                1)
            goto finished;
```

The loop is executed over and over again as long as the Boolean remains true _ its truth value is determined before each subsequent execution of the loop. Thus, if the value entered for input is 1, the loop will not be entered. (The Collatz conjecture is in this case true; no further processing is necessary.) But if input is different from 1, the loop will be executed repeatedly, until it becomes 1 (which, we believe, will happen at sometime or the other for arbitrary positive integers treated to the Collatz algorithm).

```
          while   (input != 1)
              {
                    if   (input % 2 == 1)   /* input was odd */
                          input = input * 3 + 1;
                    else
                          input/= 2;
                    cycle_length ++;
              }
```

Note that the index of the loop_ input _ is modified inside the loop. 'Me loop is executed repeatedly as long as the value of input is different from 1; that takes care of the last goto of Program 4.12; and control exits from the loop as soon as input becomes 1; this makes the second goto statement in the program superfluous.

Because Program 5.1 uses loops instead of gotos, see how much more readable and elegant it has become than our earlier version. Of course, a competent FORTRAN programmer can write FORTRAN in any language, however structured it may be!

```
/*                    Program                      5.1                    */
#                             include                             <stdio.h
main ( )
          {
                int          input,          cycle-length          =          0;
                do
                      {
                        printf   ("Enter   a   positive   trial   number   ");
                        scanf              ("%d",              &input);
                      }
                    while          (input          <=          0);
                    while (input != 1)
                      {
                          if  (input   %   2   ==   1)   /*   input   was   odd   */
                            input     =     input     *     3     +     1;
                          else
                            input               1=               2;
                          cycle_length ++;
                       }
                    printf  ("  1  appeared  as  the  terminating  digit  after  %d  iteradons",
                    cycle_length),,
              }
```

For another example of the do - while and .while () loops, let's write a program to find the sum of digits of a number. The algorithm is straightforward: while number is different from zero extract its righttmost digit using the % operator with 10 as divisor, add it to a variable sum that is initially zero, then discard the rightmost digit of number by dividing it by 10, replacing number by this quotient; and repeat these steps for the new value of number, until it reduces to zero, at which time the Boolean controlling the while () becomes false. See Program 5.2 below:

```
/*                    Program                      5.2                    */
#include                                                          <stdio.h
main ( )
          {
            int          number,          sum-Of          digits          =          0;
```

```
        printf ("Enter a ");
        do
              {
                    printf ("positive number only: .");
           /* contd */


               scanf ("%d",& number);
              }
           while (number <= 0);
           while (number)
              {
               sum_of_ digits += number % 10;
               number /= 10;
              }
            printf ("The sum of digits is %d', sum_of_digits);
           }
```

Note that braces are required in a do - while () loop when there is more than one statement sandwiched between the keywords do and while (); contrast this with the repeat - until loop of Pascal, in which several statements may be written without an enveloping begin - end pair.

Continuing further with the last example, let's now tackle a more challenging problem: find a four digit number such that its value is increased by 75% when its rightmost digit is removed and placed before its leftmost digit.

In the Previous example the value of the input variable number was successively reduced by a factor of 10 in each iteration, until it eventually became zero, making the Boolean contained in the while () loop false. But this time we'll need to store number unchanged, in order to effect the comparison with the value created after the transposition, called new_number in Program 5.3 below.

Assume the rightrnost digit is stored in rdigit. Then new_number must be 1000 * rdigit + number / 10. Now it is quite possible that there may be no numbers at all satisfying the property sought The int variable sentinel, initialised to 0, records the number of successes obtained. If there were none, the program reports this on exit from the while () loop.

```
/* Program 5.3 */
include <stdio.h
main ( )
        {
              int number = 1000, new_number, rdigit, sentinel = 0;
              while (number< 9999)
                    {
                     rdigit = number % 10;
                     new_number = rdigit * 1000 + number/ 10;
                     if  (4 * new_number == 7 * number)
                            {
                             sentinel ++;
                             printf ('%d has the required Property.\n", number);
                            }
                         number ++;
                        }
                 if (sentinel == 0)
                   printf ("There were no 4 - digit numbers with the property.\n");
           }
```

Here is the output of the program:

```
              / * Program 5.3 - output */
```

```
1212 has the required property.
2424 has the required property.
3636 has the required property.
4848 has the required property.
```

For another example of the while () loop, let's look at Program 5.4 below to generate the Fibonacei numbers, introduced in a foregoing Unit. This famous sequence of numbers is named after a pre-Renaissance Italian mathematician named Leonardo Fibonacci (which quite literally means "son of Bonacci"), who first discussed them in his book Liber Abaci ("Book about the Abacus") written in 1202. Fibonacci posed the question: How many pairs of rabbits can be produced from a single pair in one year, if every month each pair gives rise to one new pair, and new pairs begin to produce young two months after their own birth? A little thought will convince you that, provided no mishaps such as famine or predatory at- tacks befall them, and that each pair born comprises of a male and a female, the following numbers of rabbit pairs will be born in each succeeding month:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

You can see that from the third month onwards, the number of pairs produced in any month is the sum of the numbers produced in the two preceding months. With this knowledge it's a fairly simple matter to write a program that will give the number of rabbit pairs produced in any month:

```
/* Program 5.4 */
# include <stdio.h
main ( )
        {
         int fib1, fib2, pairs, loop_index = 3, month;
         printf ("Rabbit pairs produced in which month?....:");
         scanf ("%d", &month);
         if (month == 1 ll month == 2)
          pairs = 1;
         else
             {
              fib 1 = fib2 = 1;
              while  (loop_index ++ <= month)
                  {
                   pairs = fib 1 + fib2;
                   fibl = fib2;
                   fib2 = pairs;
                  }
                }
           printf ("The number of pairs produced in month %d is %d.\n", month, pairs);
          }
```

To quickly recapitulate the concepts we've discussed so far, let's work through Program 5.5 below:,

```
/* Program 5.5 * /
#include <stdio.h
main ( )
        {
         int a, b, c;
         a = b = 5;
         while (b < 20)
             {
              c = ++ a;
              b += ++ c;
             }
           printf ("a = %d, b = %d, c = %d\n", a, b, c);
           a = b = 5;
           do
        {
          c = ++ a;
          b += ++ c;
        }
     while (b <= 30 && c <= 12);
     printf ("a = %d, b = %d, c = %d\n", a, b, c);
     a = b = 0;
     if (++ a)
       if (--- b).
         while ((c = a + b ) < 5)
            {
                 ++ a;
                 ++ b;
                 ++1
/* contd */
        }
     printf ("a = %d, b = %d, c = %d\n", a, b, c);
   }
```

In the first while (), the Boolean b < 20 is initially **true**, the loop is entered, its first statement assigns 6 to c, while the second sets c at 7 and b at 12. The Boolean controlling the while ( ) remins true, so the loop is entered again. C is reset to 7, the next statement makes it 8, and assigns 20 to b. b < 20 now becomes flase; the values printed for a, b and c respectively 7, 20 and 8.

The statements of the next loop, the do - while (), are executed unconditionally the first time around. Its first statement sets a and c at 6 each, while its second increments c to 7 and assigns 12 to b. These values for b and c are such that the loop's Boolean remains true, so the loop is re-entered. a is incremented and c is reset to 7, it then becomes 8, and b becomes 20. The Boolean continues to remain true, the loop is therefore once again executed. a and c are again reset at 8 in the first statement of the loop, then c becomes 9, b becomes 29, and the loop is re-entered. c and a are set at 9, the next statement resets c to 10 and b to 39, at which time the Boolean becomes false, and the loop is exited, with a, b and c equalling 9, 39, and 10 respectively.

Proceeding further, we find:

```
     a = b = 0;
       if ( ++ a)
         if (- b)
           while ((c = a + b) < 5)
                 {
                   ++ a;
                   ++ b;
                 }
```

The Booleans with each of the ifs () are true, since they have values different from (); so the while () can be reached. At this time a is 1, b is - 1, and c is 0. Because c = 0, (less than 5), the while () loop is entered; a is incremented to 2, b to 0, and c is assigned the value 2, so that the Boolean controlling the while () remains true. Then a becomes 3, b becomes 1, c is assigned 4, and the loop is reentered. Next a becomes 4, b becomes 2, c becomes 6 and the loop is exited.

5.3 THE COMMA OPERATOR

We'll digress for a moment from our study of loops to introduce a new C operator, used most frequently in loop control expressions: this is the humble comma. It, too, belongs to the prestigious club of operators in C; however, it suffers the ignominy of having the lowest priority of all of the members of that club. Every other operator gets precedence over the comma operator. Nonetheless, it serves a useful purpose. When expressions in a C statement are separated by commas, it is guaranteed that they will be evaluated in left to right order. As an expression is evaluated, its value is discarded, and the next expression is computed. The value of the expression is the last value that was computed.

Consider the statement:

```
        x = y= 3,z = x, w = z;
```

The expression:

```
        x = y= 3
```

will be evaluated first, and will impart the value 3 to y and to x (in that order). The expression:

```
        z = x
```

will be evaluated next, and since x is 3 by this time, z is 3, too. Finally w gets the value 3.

THE COMMA OPERATOR Commas that are used to separate variable names in a declaratory or defining statement, or to separate a list of arguments (which may be expressions) in a function's invocation (such as the printf ( )), do not constitute examples of the comma operator, and it is not guaranteed that the listed expressions will be evaluated in left to right order.

Programs 5.6 and 5.7 illustrate the role of the comma as an operator. The output of each program is appended. Consider first Program 5.6.

```
/*                    Program                    5.6                    */
#                              inclutic                          <stdio.h
main ( )
          {
           int                  x,                  y,                  z;
           x             =             1,             2,             3;
           printf          ("x          =          %d\n",          x);
           x                  =                  (1,                  2,3);
           printf          ("x          =          %d\n',          x);
           z=(x=4,5,6)     /     (x     =2,y     =3,z     =4);     ,
           printf  ("x  =  %d,  y  =  %d,  z  =  %d\n",  x,  y,  z);
          }
```

In the expression:

```
                          x = 1, 2, 3
```

the comma operator guarantees that the first expression to be evaluated is the leftmost expression, which assigns the value 1 to x. 'The assignment expression is then discarded and the next expression, 2, is evaluated in which no operation is performed. It is similarly discarded. The value of the last expression is 3, and that is the value of the entire expression. The value printed in the first line of output is the current value of x.

In the expression:

```
                          x =( 1, 2, 3)
```

the parentheses ensure that the expression:.

```
                          1, 2, 3
```

is evaluated first, which, by virtue of the comma operator, gets the value 3. This is the value that is assigned to x, the value written in the second line of output.

In the last assignment statement in the program:

```
                          z = (x = 4, 5, 6) / (x = 2, y = 3, z = 4);
```

the numerator and denominator expressions get the values 6 and 4 respectively. z is 1.

Program 5.6: Output

```
                          x                    =                    1
                          x                    =                    3
                          x = 4, y =3, z =1
```

The fact that x gets the value 4 in the third line of output is intriguing. It implies that the numerator was evaluated after the denominator, in which x was set equal to 2. But consider the two lines in the output of Program 5.7 below, the first of which seems to indicate that the numerator is evaluated after the denominator, and the second that the numerator is evaluated before the denominator! (The first time x gets the final value 4, implying that the assignment of 2 to it in the denominator must have been done earlier, and was subsequently overwritten.)

```
/* Program 5.7 */
#include <stdio.h
main ( )
        {
         int x, y, z;
         z = (x = 4, 5, 6) / ( x = 2, y = 3, z = 4);
         printf ("x = %d, y = %d, z = %d\n', x, y, z);
           (x = 4, y = 5, z = 6) 1 (x = 2, y = 3, z = 4);
         printf ("x = %d, y = %d, z = %d\n", x, y, z);
        }
```

Program 5.7: Output

```
x = 4, y = 3, z = 1
x = 2, y = 3, z = 4
```

Effects which arise from a compiler dependent (and therefore non-unique) sequence of execution of operations are called side effects. They're detrimental both to program robustness and portability, and should be avoided.

Commas in the argument list of a declaratory statement, or an invoked function such as a printf (), play the role of separator rather than operator, and it is not guaranteed that the separated expressions will be evaluated in left to right order. Program 5.8 below was executed in VAX C and on a PC with ANSI C. Note the differences in the outputs.

```
/* Program 5.8 */
#include <stdio.h
main ( )
        {
         int x= 1,y = (x-=5),z =(x =7) - (x =3);
         printf ("x = %d, y = %d, z = %d\,n", x, y, z);
         printf ("x = %d, y = %d, z = %d\n",
         x = ++ y, y = ++ x, z = ++ x + ++ y);
         printf ("x = %d, y = %d, z = %d\n", x, y, z);
        }
```

Program 5.8: Output from VAX C

```
x = 3, y = -4,z = 0
x = 6, y = 5, z = 1
x = 6, y = 6, z = 1
```

Program 5.8: Output from a PC based ANSI C compiler

```
x = 3, y = -4, z = 4
x = 6, y = 5, z =1
x = 6, y = 6, z =1
```

Moral: Avoid program statements in which the order of evaluation of embedded expressions is undefined in the language.

Check Your Progress 1

Question 1: Give the output of the following program

```
/* Program 5.9 */
#include <stdio.h
main ()
```

```
{
 int a, b, c;
 do
 printf ("How many times is this statement printed ? \n");
 while  (a =1,b = 2,c = O);
        a= 1,b=2;
    /*contd */


 while   (++a,++b, c = b - ++ a)
        Printf ("a = %d, b = %d, c = %d\n", a, b, c);
        printf ("a = %d, b = %d, c = %d\n", a, b, c);
                a = 1" b = 2;
     while (a < 10)
        b += ++ a;
                c = a + b;
        printf ("a = %d, b = %d, c = %d\n", a. b, c);
               a =1,b = 2,c = O;
    do
        c+=++ b + ++a;
                  while (c < 10);
        Printf ("a = %d, b = %d, c = %d\n", a, b, c);
                            a= 1,b=2,c=o;
    while   (++ b <10)
        {
         c = a ++;
         a += c ++;
        }
       Printf ("a = % d, b = % d, c = % d\n", a, b, c);
                    b = 2;
     do
        while   (a=++b<l0);
     while  (c = 0);
        Printf ("a = % d, b = % d, c = % d\n", a, b, c);
    b = 2, c = 0,
     do
        while  ((a = ++b)
    while  (++ c < 3);
        pnntf ("a = % d, b = % d, c = % d\n", a, b, c);
   }
```

Question2: In the loops:

```
do
        while (a = ++ b < 10);
    while (c = 0);
```

Prove that it makes no different to the result whether the three lines are interpreted as an "empty" do - while () loop follwed by another while () loop, or as a while () loop fully nested inside the do - while ().

Question 3: Suppose int x = 5; what's the value of x /++ x ?

Question 4: Modify Program 5.4 so that it continus to give the correct output after making the following changes:

(i) Replace the while () loop by a do - while () loop.
(ii) Replace loop _index ++ by ++ loop _index.

5.4 THE TRANSFER OF CONTROL FROM WITHIN LOOPS

There are several ways by which the execution of a loop may be terminated. There is of course, the goto statement but its use is frowned upon, though it can provide a useful escape route from deeply nested loops (loops within loops within loops ...); more appropriate are the break and the continue statements, and the return statement; this last however can be used only inside a function other than main (), and we shall defer its discussion to a subsequent unit.

The break statement transfers control to the statement following the do - while (), while (), or for (;;) loop body. The continue statement transfers control to the bottom of the loops and thence to a re-evaluation of the loop condition. In the do - while () and while () loops, the continue statement causes the Boolean within the while () to be re-evaluated; loop execution is continued if the Boolean there remains true. The continue statement cannot be used outside loops.

Program 5.10 illustrates how the break statement enables control to be transferred outside while () loop. It computes the month in which, beginning at month 2, rabbits reproducing ac- cording to the Fibonacci formula will exceed a limit scanned from the keyboard.

```
/* Program 5. 1 0 */
#include <stdio.h
main ( )
        {
         int fib1= 1, fib2 = 1, fib3, limit, month = 2;
         do
             {
              printf ("Upper limit for Fibonacci\'s rabbits? (must exceed 1):");
              scanf ("%d", &limit);
             }
            while (limit <= 1);
            while (fib3 = fib 1 + fib2)
             {
             month ++;
             if (fib3 =limit)
              break;
             fibl = fib2,
             fib2 = fib3;
             }
            printf ("Limit is reached or exceeded in month: %d.\n", month);
          }
```

The continue statement transfers control back to a re-evaluation of the loop condition for while () and a do - while () loops. For a simple example of the continue statement, let's look at Program 5. 1 1, which finds out whether a number input to it is a perfect square (that is, if it has an integer square root). The logic is straightforward:

```
Assign 1 to loop_index
if (loop_index * loop_index < input)
  increment loop_index and continue;
else
  break;
```

```
/* Program 5. 11 */
#include <stdio.h
main ( )
  {
       int input, loop_index = 1;
       do
            {
             printf ("Enter a number greater than 1.....");
```

```
                    scanf ("% d", &input);
                   }
               while (input <= 1);
               while  (loop_index ++)
                   if  (loop_index * loop-index < input)
                           continue;
                       else
                           break;
               if (loop_index * loop_index == input)
               printf ("%d is a perfect square; square root %d.\n" input, loop_index) ;
               else
               printf ("The square root of %d lies between %d and %d.\n",
                                                input, loop_index - 1,
                                                loop_index);
       }
```

The continue statement causes loop_index to be incremented for as long as loop_index * loop_index input; the square root of input has neen neither reached nor exceeded, but when either of these events happens, the break statement forces control out of the loop. The next statement prints the square root if there is an exact one, or else the integers between which it lies.

C loops may be nested one inside the other, to any depth. The only condition is that any included loops must be entirely enveloped within the surrounding loop. Processing inside nested loops may give rise to a situation when the goto statement may justifiably be used to transfer control from the innermost loop to outside all of the surrounding loops, as soon as a desired condition is found, instead of letting control fall through all of the intervening break statements. While the goto must be avoided to the extent possible, one must not make a fetish of avoiding it. Here's a problem for which a goto transferring control out of nested loops is appropriate:

Three school girls walking along see a car hit a cyclist. The driver drives off without stopping. They report the matter to the police like this: the first girl says "I couldn't see the entire number of the car, but 1 remember that it was a four-digit number and its first two digits were divisible by 11." The second girl says, "I could see only the last two digits -of the car's number, and they too formed a number divisible by 11." The third girl said, 'I remember nothing about the number, except that it was a perfect square." Write a C program to help the police trace the car.

Clearly, the number we seek is a square number of the form aabb, where a and b will range from 1 through 9. (Why shoudn't b range from 0 through 9?) Program 5. 12 solves the problem.

```
/* Program 5. 12 */
#include <stdio.h
main ()
       {
        int a = 1, b, sqroot;
        while  (a <= 9)
            {
             b = 1;
             while  (b < 9)
                 {
                  sqroot = 32;
                  while  (sqroot ++)
                      {
                       if (sqroot*sqroot<1100*a+1l*b)
                      continue;
                       else
                         if  (sqroot * sqroot == 1 100 * a + 1 1b)
                           goto writeanswer;
                         else
                        break;
                      }
                    b ++;
                   }
```

```
                a++;
               }
  writeanswer: prinlf ("Car'\s number is: %d\n", 1100 * a + 11 * b);
 }
```

Check Your Progress 2

Question 1: How many terms of the series:
1 * 1 + 2 * 2 + 3 * 3 +... + n * n
must be summed before the total exceeds 25,000?

Question 2: Referring to Program 5.12,why should the variable sqroot begin at 33 in the innermost while ( ) loop ?

Question 3: Rewrite Program 5.12 without using a goto statement.

5.5 THE (TERNARY) If - Then - Else OPERATOR

The if , then - else operator has three operands, the first of which is a Boolean expression, and the second and third are expressions that compute to a value:

```
first_exp ? second_exp: third_exp
```

The three expressions are separated by a question mark and a colon as indicated. The operator returns the value of second _exp if first _exp is true, else it returns the value of third _exp. Hence its name: the if - then - else operator. In other words, the ternary operator returns a value according to the formula:

```
Boolean _expression ? this _value _if _true: else _this
```

Semantically. therefore, the statement which assigns one of two values to a variable x via the ternary operator:

```
x = first _exp ? second _exp : third _exp;
```

is equivalent to:

```
if (first _exp)
        x = second--exp;
    else
        x = third _exp;
```

Often there's little to choose between the two alternative mechanisms of assigning a value to x; but the ternary operator makes for more concise and elegant, (though at times obfuscating) code.

The ternary operator has a Priority just above the assignment operator, and it groups from right to left. *If second_exp and third _exp are expressions of different types, the value returned by the ternary operator has the wider type, irrespective of whether first_ exp was true or false.* For example:

```
x = (5 3) ? (int) 2. 3: (float) 5,
```

will return to x the value 2.0, rather than 2.

Check Your Progress 3

Question 1: Give the Output of the following program

```
/* Program 5.13 */
#include
main ( )
                    {
                     int I = 1;
                     if (i / (5 3) ? (int) 2..3 : (float) 5)
                       printf ("Will this line be printed.?\n");
                     else
                       printf (,,Or will this line be printed ?\n");
                    }
```

Question 2: Would the output of the Program above be different if the cast operator (float) was removed from the third operand of the ternary operator ?

Question 3: Observe that the following statements assigns the lesser of two values val_1 and val_2 to a variable named lesser:

```
lesser = val_1 < val_2? val_1: val_2;
```

Prove that the greater and lesser values of two values val_1 and val_2 may be assigned to variables named greater and lesser respectively by one statement as follows

```
greater = (( lesser = val_1 < val_2? val_1: val_2) = = val_1)? val_2:val_1:
```

Which of the pairs of parenthesses above are necessary?

Our next example applies the ternary operator to encode the Russian Peasant Multiplication Algorithm. Apparently peasants in Russia use the following algorithm when they multiply two integers, say vai_l and val~2. (For the sake of computational efficiency, it is necessary to determine the lesser of the multiplier and the multiplicand which form the product.)

Let L be the lesser and G the greater of val.1 and val-2, and let P be the variable to store their product, initialised to 0.

```
while (L is not equal to zero)
    {
    if L is odd, P = P + G;
    halve L, ignoring any remainder;
    double G;
    }
```

For example, to multiply 19 with 3 1, L = 19, G = 3 1, P = 0. Then:

```
P      =     31
L=9
G      =     62
P=           93
L=4
G      =    124
P=93
L=2

G      =    248
P=           93
L      =      1
G      =    496
P=          589
L=0
```

and the product is 589.

Program 5.14 below illustrates the use of the ternary operator to encode the Russian Peasant Algorlthm:

```
/* Program 5.14 */
#Include <stdio.h
main ( )
          {
          int val_ I, val_2, lesser, greater, result = 0;
          printf ("Russian Peasant Multiplication Algorithm\n");
          printf ("\nEnter multiplier: ");
          scanf ("%d", &val_l);
          printf ("\nEnter multiplicand: ");
          scanf ("%d", &val-2);
          greater = ((lesser = val 1
          while (lesser)
                 {
                 result+= lesser 'Yo-2 ? greater: 0;
             /* contd. */

                  lesser /= 2;
                  greater *= 2;
                 }
              printf ("%d\n', result);
             }
```

Question l: Write a C program which uses the ternary operator to print - 1,()or l if the value input to it is negative, zero or positive.

Question 2: Execute the program listed below, and explain its output.

```
/* Program 5. 1 5 */
#define TRUE 1
#define FALSE 0
#define T "true"
#define F "false"
#include <stdio.h
main ( )
          {
           int i = FALSE, j = FALSE;
           while (i < TRUE) '
                 {
                  while (j <TRUE)
                        {
                          ( printf ("%s && %s equals %s\n", i ? T: F,
                         j ? T: F, i && j ? T: F);
                         j ++;
                        }
                   i ++;
                   j = FALSE;
                  }
             }
```

(Note that Program 5.15 contains two while () loops, one nested fully inside the other. The outer loop is executed for each value of i, that is, twice: once when i is FALSE, and once when it is TRUE; the inner loop is executed for each value of j. When i is FALSE, j is FALSE and TRUE; that makes for two executions of the inner loop. When i is TRUE, j again ranges from FALSE to TRUE, which makes for two further iterations of the inner loop.)

Question 4: Write a C language program to print a truth table for die Boolean expression i && (j ll k), where i, j and k range from FALSE to TRUE.

5.6 THE switch - case - default STATEMENT

The switch - case - default statement is useful where control flow within a program must be directed to one of several possible execution paths. We've already seen that a chain of if () - elses may be used in this sort of situation. But if () - elses become cumbersome when the number of alternatives exceeds three or four; then the switch - case - default statement is appropriate. For an illustrative program, let's write one to answer questions such as: What day was it on 15 August 1947? What day will it be on New Year's Day in AD 4000? there's a famous algorithm, called Zeller's congruence, that one can use to find the day for any valid date between 1581 AD and 4902 AD:

Step 1: January and February are considered the eleventh and twelfth months of the preceding year; March, April, May,..., December are considered the first, second, third tenth months of the current year. For example, to find the day which fell on 23 January 1907, the Month number must be set at 11, the Year at 1906.

Step 2: Then, given the values of Day, Month and Year, from Step 1 the expression:

```
Zeller =
        ((int) ((13 * Month - 1) /5) + Day + Year % 100 +
        (int) ((Year % 100) / 4) - 2 * (int) (Year / 100) +
        (int) (Year /400) + 77) % 7
```

will have one of the values 0, 1, 2, 3,...,6. (Why ?)

Step 3: 0 corresponds to a Sunday, 1 to a Monday, etc. and 6 corresponds to a Saturday.

Let's first consider Step 3 of our program. The if () - else way to handle it is unwieldy:

```
if (Zeller == 0)
  printf ("That was a Sunday.\n");
else
  if (Zeller == 1)
    printf ("That was a Monday\n");
  else
    if ( ) .....
```

There is the very real danger of further conditions and object statements overflowing the right margin of the page!

In the switch - case - default statement, a discrete variable or expression is enclosed in parentheses following the keyword switch, and the cases, each governed by one or more distinct values of the switch variable, are listed in a set of curly braces as below:

```
switch (discrete_variable or expression)
        {
case val_1          :     statements;
                          break;
case val_2          :     statements;
                          break;
case val_3          :     statements;

                          break;
                    ...
case val_n          :     statements;
                          reak;
Default             :     statements;
                          break;
}
```

To use the switch statement to encode Zcller's algorithm, for example, one would write:

```
switch (Zeller)
        {
        case 0:
                        printf ("%d-%d-%d was a Sunday.\n", Day, Month, Year);
                        break;
        case 1:
                        printf ("%d-%d-%d was a Monday.\n", Day, Month, Year);
                        break;
        case 2:
                        printf ("%d-%d-%d was a Tuesday.\n", Day, Month, Year);
                        break,
        case 3:
                        printf ("%d-%d-%d was a Wednesday.\n", Day, lvlonth, Year);
                        break;
        case 4:
                        printf ("%d-%d-%d was a Thursday.\n", Day, Month, Year);
                        break;


        Case 5:
                       Prinif ("%d-%d-%d was a Friday.\n", Day, Month, Year);
                       Break;
        Case 6:
                       Printf ("%d-%d-%d was a Saturday.\jl", Day, Month, Year);
                       Break;
        }
```

Then, when the switch is entered, depending upon the value of discrete _variable, the corresponding set of statements (following the keyword case) is executed. But if the switch able has a value, different from those listed with any of the cases, the statements corresponding to the keyword default are executed. The default case is optional. If it does not occur in the switch statement, and if the value obtained by the switch variable does not correspond to any listed in the cases, the statement is skipped in its entirety- We don't have a default statement in the example above: there's no way that an int value can leave a remainder outside the set [0 - 6] on division by 7!

The break statement, the last statement of every case including the default, is necessary: if it were absent, control would fall through to execute the next case!

This regrettable property of the case statement, that control slips through from one case to the next if isn't forced out by the placement of a break statement, may be used to advantage when multiple values of discrete_variable say val_1, val_2 or val_3, must trigger the same set of statements:

```
                    Case val_1          :
                    case val_2          :
                   case val_3           :   Statements;
                                            break;
```

If discrete_variable happens to get any of the values val_1, vai_2 or val_ 3, control will reach the corresponding case and Call through to the set of statements corresponding to val_3. Though more than one case value may set off the same set of statements, case values must all be different; and they may be in any order.

Providing a break after the statements listed against default ensures that if you add another case after the program is up and running, and discrete-variable happens to get a value corresponding to default, control will not then flow forwards to execute the case you added last! The break statement must particularly be kept in mind by Pascal programmers writing C programs, because the equivalent Pascal statement does not need such a mechanism to escape from the cases.

Probably the most surprising line of Program 5.16 is the scanf () statement written as a cornponent of the Boolean controlling the while () loop:

```
        while (scanf ("%d-%d-%d", &Day, &Month, &Year)!= 3)
```

Here we have made used of the scanf () property that it returns the number of values read and assigned, which in this case must be three: for Day, Month and Year. The while () loop is executed until the expected number and types of values for these variables have been entered. Note also that the scanr () contains non-format characters _ the hyphens _ in its control string: it expects the values of Day, Month and Year to be separated by single hyphens, and no other characters. These, and other properties of scanf () are discussed in Unit 7.

An important point to note in the Program below are the tests to determine whether a date entered is valid. For Zeller's congruence to work correctly, no year value can lie outside the range 1582 - 4902; moreover, no month number can be less than one, or greater than twelve no month can have more days than 31, and no date in a month can have a value less than 1; February can have 29 days only in a leap year, and February, April, June, September and November have less than 31 days apiece.

Note also that we have chosen to introduce two additional int variables ZMonth and ZYear which are assigned values by the if () statements below:

```
if (Month < 3)
  ZMonth = Month + 10;
else
  ZMonth = Month - 2;
if (ZMonth < 10)
  ZYear = Year - 1;
else
  ZYear = Year;
```

Observe that these statements are in accord with Step 1 of Zeller's Algorithm.

It is always good policy to retain the values of input variables; since the formula calls for changed values of Month and Year, we store the new values in ZMonth and ZYear, leaving Month and Year untouched. That way, they'll be available when we need their values later, in [he print()' statements with the cases.

Finally, the exit () function is useful when immediate program termination is required. The call

```
exit (n)
```

closes all files that may be open, flushes memory buffers, and returns the value n, the exit status, to the function that called it; in a Unix like programming environment, n is usually zero for error free termination, non-zero otherwise; the value of n may be used to diagnose the error.

```
/* Program 5.16 */
#include <stdio.h
# define LEAP_YEAR (Year % 4 == 0 && Year % 100!= 0) \
                    || Year % 400 == 0
main ( )
        {
         int Day, Month, Year, Zeller, ZMonth, ZYear;
         printf ("\nThis program finds the day corresponding to a given date.\n");
         printf ("nEnter date, month, year .... format is dd-mm-yyyy.\,n");
         prinif ("\nEnter a 1 or 2-digit number for day, followed by a\n");
         printf ("\ndash, followed by a 1 or 2-digit number for month,\n");
         printf ("\nfollowed by a dash, followed by a 2 or 4-digit number\n");
         printf ("\n for the year. Valid year range is 1582 -4902, inclusive.\n");
         prinif ("\n(A 2-digit number for the year will imply 20th century\n");
         printf ("\nycars.)\n\n\n\n Enter dd-mm-yyyy: ");
         while (scanf("%d-%d-%d",&Day,&Month,&Year)!=3)
              {
                printf ("\nInvalid number of arguments or hyphens mismatched\n");
                printf ("\nRe-enter:");
              }
           if (Year < 0)
```

```
                    {
                      printf      ("\nlnvalid      year      value....Program      aborted..");
                      exit                              (                              );
                    }
                 if                                                                 (Year
                   Year                          +=                                 1900;
                 if   (Year 4902)
                    {
                      printf    ("\nlnvalid    year    value    ....    Program    aborted..");
                      exit                              (                              );
                    }
                 if   (!(LEAP_YEAR) && (Month == 2) && (Day 28))
                    {
                      printf      ("\nInvalid      date      ....      Program      aborted..");
                      exit                              (                              );
                    }
                 if ((LEAP-YEAR) && (Month == 2) && (Day 29))
                    {
                /* contd.*/
             printf        ("\nInvalid        date        ...        Program        aborted..");
             exit(                                                                          )
            }
      if (Month < | || month 12)
            {
            printf              ("\nlnvalid              month....Program              aborted..");
            exit
            }
      if (Day 31)
            {
            printf      ("\nlnval      id      date      ....      Program      aborted..");
            exit                                (                                        );
            }
      if ((Day 30) && (Month == 4 || Month == 6 ||
            Month       =       =       9       ||       Month       ==       ||       ))
            {      printf      ("\nInvalid      date      ....      Progrwn      aborted..");
            exit                                (                                        );
            }
      if (Month < 3)
            ZMonth = Month + 10;
      else
            ZMonth = Month - 2;
      if (ZMonth 10)
            ZYear = Year - 1;
      else
            ZYear = Year;

  Zeller  = ((int) ((1 3 ZMonth - 1) 1 5) + Day + ZYear % 1 00 +
             (int) ((ZYear % 100) 1 4) - 2 * (int) (ZYear 1 100) +
             (int)         (Zyear         1400)         +         77)         %         7;

      printf                                                            ("\n\n\n\n\n      ");
      switch (Zeller)
                {
                case 0:
                     printf   (%d-%d-%d   was   a   Sunday.\n",   Day,   Month,   Year);
                     break;
                case 1:
                     printf   ("%d-%d-%d   was   a   Monday.\n',   Day,   Month,   Year);
                     break;
                case 2:
                     printf   ('%d-%d-%d   was   a   Tuesday.\n",   Day,   Month,   Year);
                     break;
            case 3:
                printf ("%d-%d-%d was a Wednesday.\n", Day, Month, Year);
                break;
            case 4:
                printf ("%d-%d-%d was a Thursday.\n', Day, Month, Year);
                break;
            case 5:
                printf ('%d-%d-%d was a Friday.\n", Day, Month, Year);
                break;
            case 6:
                printf ("%d-%d-%d was a Saturday.\n". Day, Month, Year);
                break;
            }
    }
```

A long replacement string of a #define may be continued into the next line by placing a back-slash at the end of the part of the string in the current line. (Reread the third and fourth lines of Program 5.16.) [Note: Recall that we can use this device with quoted strings: if you have to deal with a long string, longer than you can conveniently type in a single line, you may split it over several lines by placing a backslash as the last character of the part in the current line. For example:

```
"A long string, continued over two lines using the backslash \
character."
```

ANSI C treats string constants separated by white space characters as an unbroken string.]

Check Your Progress 5

Question 1: In Program 5.16 why is it preferable to write

```
if ((Day 30) && (Month =-- 4 ll Month == 6 ll Month == 9 ll Month == ll)) etc.
```

instead of

```
if ((Month = 4 ll Month = 6 ll Month = 9 ll Month = 1 1) && (Day 30)) etc. ?
```

Question 2: Is the cast operator (int) required in the expression for Zeller in Program 5.16? Rewrite the expression without the cast operator, and without redundant parentheses.

Question 3: Since 77 is exactly divisible by 7, is its presence required in the expression for Zeller in Program 5.16? Explain why or why not.

Question 4: In Program 5.16, is !(LEAP-YEAR) different from !LEATP-YEAR?

Question 5: The switch - case - default statement is not always a better choice than the if () - else, even when there may be several cases to include in a program. Rewrite the following program, which scans a char input and determines its hexadecimal value (if it has one) by using a switch instead of the if ( ) - elses.

```
/*          Program          5.17          */
#include                              <stdio.h
main ( )
    {
      char                              digit;
      printf        ("Enter       hex      digit:");
      scanf                ("%c",       &digit);
      if                      (digit='a'&&digit<='f')
       printf ("Decimal value of hex digit is %d.\n", digit = digit - 'a'+ 10);
      else
```

```
        if (digit = 'A' && digit <= 'F')
          printf ("Decimal value of hex digit is %d.\n", digit = digit - 'A' + 10);
        else
        if (digit='0'&&digit<='9')
          printf ("Decimal value of hex digit is %d.\n', digit - '0');
        else
          printf ("You typed a non- hex digit.\n');
      }
```

Question 6: If you have an ANSI C compiler execute the program below and state its output:

```
/* Program 5.18 */
#include <stdio.h
main ( )
      {
        printf ("%s", "How " "many ' 'strings ' "do - "we " "have?");
      }
```

In Program 5.17 note that the alphabetical hex digits may be entered both in lowerease or in uppercase characters. But this makes for a long if ( ) - else:

```
            if (digit='a'&&digit 'f')
            etc.
            else
            if (digit='A'&&digit 'F')
              etc.
```

The toupper () function may be used with advantage in such situations; it returns the uppercase equivalent of its character argument (if it was a lowercase character) or its argument itself if it wasn't. The value of its argument remains unaltered. Thus toupper ('x') returns 'X' toupper ('?') returns '?'. To use the toupper ( ) function, #include the file <ctype.h just as you do the file <stdio.h. See Program 5.19 below.

```
/* Program 5.19 */
#include <stdio.h
#include <ctype.h
main ()
      {
        char digit;
        printf ("Enter hex digit:");
        scanf ("%c", &digit);
        if (toupper (digit) = 'A' && toupper (digit) <= 'F')
          printf ("Decimal value of hex digit is %d.\n",
          digit = 'a' ? digit - 'a'+ 1 0 : digit - 'X + 1 0);
        else
        if (digit= 'O' && digit <= '9')
          printf ("Decimal value of hex digit is %d.\n", digit - '01);
        else
          printf ("You typed a non-hex digit.\n");
      }
```

Besides the functions toupper () and its analogue called tolower (), ctype h provides many other functions for testing characters which you may often find useful. These functions return a non-zero (true) value if the argument satisfies the stated condition:

isalnum	(c)	c is an alphabetic or numeric character
isalpha	(c)	c is an alphabetic character
iscntrl	(c)	c is a control character
isdigit	(c)	c is a decimal digit
isgraph	(c)	c is a graphics character
islower	(c)	c is a lowercase character
isprint	(c)	c is a printable character

ispunct	(c)	c is a punctuation character
isspace	(c)	c is a space, horizontal or vertical tab, formfeed, newline or carriage return character
isupper	(c)	c is an uppercase character
isxdigit	(c)	c is a hexadecimal digit

Note that the <ctype.h> function isxdigit (c) makes Programs 5. 17 and 5. 1 9 somewhat superfluous!

In Program 5.20 below we use the switch statement to count spaces (blanks or horizontal tabs), punctuation marks, vowels (both upper and lowercase), lines and the total number of keystrokes received. The program processes input until it is sent a character that signifies end of input This character is customarily called the "end of file" character, or EOF, but it's not really a character: for if EOF is to signify end of input to a program, its value must be different from that of any char. That is why the variable c in Program 5. 19 is declared an int; ints can include all chars in their range, as well as the EOF. A MACRO definition in <stdio.h #defines a value for it. So #including <stdio.h makes EOF automatically available to your program.

In the while () statement:

```
while ((c = gctchar ( )) ! = EOF)
```

note that

```
c = getchar ( )
```

is performed beforc, C is compared against EOF. c first gets a value'. that value is then compared against EOF. As long as C is different from EOF, the Boolean controlling the while ( ) remains true, and the loop is executed. When EOF is encountered, the loop is exited, and the program terminates. In DOS environment on a PC, EOF is sent by pressing the CTRL . and Z keys together.

```
/* program 5.20 */
# include <stdio.h
main ( )
        { int C;
        long keystrokes = 0, spaces = 0, punct_marks = 0, lines = 0,
        vowels = 0,
        prilitf ("Enter text, line by line. and 1\'11 give you some\,n\
        statistics about it ... tert-nitiate your input by entering\,,,\
        CTRL-Z as the first character of a newline ... thans the\,n\
        DOS EOF character (i-nay be different in your computing environi-nent).M\n");
        while ((c = getchar 0) != EOF)
                {
                switch (c)
                        {
                        case '\n' ++ lines;
                           keystrokes ++;
                           break;
                        case '\,t':
                        case ' ' : spaces ++;
                           keystrok'CS ++;
                           break;
                        case ',':
                        case '.' :
                        case ':' :
                        case ';' :
                        case '!' :
                        case '?' :Punet-mtlrks ++*,
                           keystrokes ++;
                           break,
                        case                    'a':
                        case        'A'            :
                        case        'e'            :
                        case        'E'            :
                        case        'i'            :
                        case        'l'            :
                        case        'o'            :
                        case                    'O':
                        case                    'u':
                        case                    'U':
                        case                    'y':
                        case 'Y' : vowels++;
                           keystrokes          ++;
                           break',
                        default:
                           keystrokes          ++;
                           break;
                        }
                }
            printif ("Input statistics*.\

\n lilies = %1d,\n keystrokes = 0/cld,\n vowels = %Id,\
\,n punctuation marks = %Id, spaces = %1d\n", lines, keystrokes, vowels, punct_ marks, spaces);
        }
```

The switch statement is Useful only when the cases are contolled by integer values; in a pro- gram where the conditions to control branching are set in term of floating point values, if ( ) - else statement must be used.

5.7 SUMMARY

C, like Pascal, Provides three loop structures to control the repeated execution of one or more statements. Two of these. the while () and do_ while () loops have been studied in this unit. A while () loop's statements are executed only if the loop condition is found to be true; and they are repetitively executed until the condition remains true. More Picturesquely, the while ()loop has a Boolean as sentry at its doorway. Control is permitted to flow into it only if the Boolean is true: then the sentry opens the doorway; else, control flows around it to execute the next sequential statement. The do - while () loops is less stringent. It's an exit condition loop. tested when control is on the point of exiting from the loop. That means that the condition to again execute the loop body is evaluated only after the loop has been executed one Usually an entry condition loop makes better sense, but sometimes it is necessary to use the do - while () loop. For instance, if a program requests the user to type in data, it would naturally need to test those data for validity before processing them. Think for a moment of the problem of finding the day which fell on a given date: it would not do to rush straightaway to Zeller's, algorithm and attempt to find the day which occurred on 30 February 1992; is such a date possible? Then if it is discovered within the loop body that the data received are bad, the Boolean condition could be written to repeatedly transfer control back to request for good data, until good data are received; at which time control can be made to flow forwards.

```
            do
                printf ("Enter valid date:...,, );
                scanf ("%d-%d-%d",&dd, &mm, &yy);
                Process dd, mm, yy as entered to
                determine if valid_date is true;
            while (!(valid_date)),
```

The comma operator, which has the, lowest Priority of all C operators, forces left to right evaluation of the expressions it separates. The continue statement, which can be used only in- side loops, forces a re-evaluation of the loop condition; and a further iteration is performed only if the condition evaluates to true. The break statement forces control out of any of the loop structures and also out of the switch statement, which provides for multi_ way branching . The break statement can be used in only these contexts. The ternary operator tests a condition and returns one of two values, the first if the condition was true, the second if it was not:

```
x = condition ? first _ val : second _ val;
```