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 :

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.