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;
```

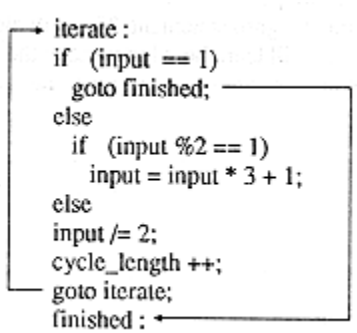
In effect we have the following situation:

```
                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.



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+11*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 'I' :
                            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;
```