UNIT 5  CONTROL STRUCTURES - I

Structure                                                                 |
5.1 Introduction                                                                 | Page No. |
   Objectives                                                                   |          |
5.2 The while() and do - while() Loops                                          | 101      |
5.3 The Comma Operator                                                          | 108      |
5.4 The Transfer of Control from Within Loops                                   | 110      |
5.5 The (Ternary) if()- then - else Operator                                    | 113      |
5.6 The switch - case - default Statement                                        | 115      |
5.7 Summary                                                                     | 123      |
5.8 Solutions/Answers                                                           | 124      |

5.1 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. In this Unit, we will introduce to loops. In Sec. 5.2, we will discuss the while() and do-while() loops. We will also introduce you the the switch-case-default statement in Sec. 5.6. In Sec. 5.4, we will discuss how to transfer control from within loops. In Sec. 5.3, we will introduce you to the comma operator.

Objectives

After studying this unit, you should be able to

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

5.2 THE while () AND do - while () LOOPS

Let us begin our discussion of loops by looking at a common task you may have performed many times. Suppose you want look up a word in the dictionary. For example, you are looking up 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 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 instruction repetitively; in computer jargon, you’ve
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 instruction—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 unit. 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 even 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 be 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 a a roundabout of several roads radiating outwards, of which only one will (optionally) take you where you want to go. You have to decide from the possibilities available, and choose the appropriate path. If 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.

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:

```c
/* Program 4.12; file name:unit4-prog12.c */
#include <stdio.h>
int main()
{
    int input, cycle_length = 0;
    ```
begin_again:
printf("Enter a positive trail 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 terminating \
digit after %d iteration(s)!", cycle_length);
return (0);
}

Listing 1: Collatz problem again.

Note that the first goto in Program 4.12 transfers control over the 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 trail number.");
scanf("%d", &input);
if (input <= 0)
  goto begin_again;

In effect we have the following situation:

do
{
  printf("Enter a positive trail 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 to 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 the input is less than or equal to zero, the terminating Boolean remains true, and the program asks for another value for input. Once 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 Listing 2 on the following page.

Moral: Use the do - while() loop wherever a statement (or a set of statements, in which case they must be enclosed in braces) is to be executed at least once, and possibly oftener.
Furthermore, in Program 4.12, the statements between the labels iterate and finished are repeatedly executed as long as the value of input is different from 1.

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

This is 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 statement:

    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. The 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 Listing 2 uses loops instead of go tos, see how much more readable and elegant it has become than our earlier version.

    /* Program 5.1; file name:unit5-progl.c */
    # include <stdio.h>
    int main()
      {
        int input, cycle_length = 0;
        do
          {
            printf("Enter a positive trial number:");
            scanf("%d", &input);
          }
        while (input <= 0); /* End of do */
        while (input != 1)
          {
            ...
if (input % 2 == 1) /* input was odd */
    input = input * 3 + 1;
else
    input /= 2;
    cycle_length++;
}
printf("1 appeared as the terminating digit \nafter %d iteration(s)", cycle_length);
return (0);
}
Listing 2: Collatz problem using while() loop.

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 rightmost 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; File name: unit5-prog2.c */
#include <stdio.h>
int main ()
{
    int number, sum_of_digits = 0;
    printf("Enter a ");
    do
    {
        printf("positive number only: ");
        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);
    return (0);
}
Listing 3: Finding the sum of the 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();

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 Listing 4 on the following page below.

Assume the rightmost digit is stored in rdigit. Then new_number must be 1000 * rdigit + number / 10. Now it is quite possible that there may be no number containing the property sought. The int variable sentinel, initialised to 1000 before the while loop was none, the program reports
/* Program 5.3; File name: unit5-prog3.c */
#include <stdio.h>

int 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 \n with the property. \n");
  return (0);
}

Listing 4: Use of sentinels.

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 Listing 5 below to generate the Fibonacci 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 Bonaccio"), 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 attacks befell 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; File name: unit5-prog4.c */
#include <stdio.h>

int main()
{
  int fib1, fib2, pairs, loop_index = 3, month;
  printf("Rabbit pairs produced in which month?...: ");
  scanf("%d", &month);
  if (month == 1 || month == 2)
    pairs = 1;
  else
  { fib1 = fib2 = 1;
{  
pairs = fib1 + fib2;  
fib1 = fib2;  
fib2 = pairs;  
}/*End while*/  
}/*End else*/  
printf("The number of pairs produced in \nmonth %d is %d.\n", month,pairs);  
return (0);
}

Listing 5: Fibonacci Sequence.

To quickly recapitulate the concepts we've discussed so far, let's work through Listing 6 below:

/* Program 5.5; File name:unit5-prog5.c */
#include <stdio.h>
int 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);/*End do*/  
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;  
            }
printf("a = %d, b = %d, C = %d\n", a, b, c);
return (0);
}

Listing 6: A recapitulation.

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() remains 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 false; the values printed for a, b and c are 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:

```c
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 0; 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. Try the following exercise to check your understanding of the while and do-while loops.

E1) For a prime p, the ring \( \mathbb{Z}_p \) is a finite field of p elements and the non-zero elements of the field form a cyclic group of order \( p-1 \). Write a program that reads in a prime p and a natural number a less than p and finds the order of a in \( \mathbb{Z}_p \).

We close this section here. In the next section, we will discuss the comma operator.

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 statements 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.

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.
Program in Listing 7 illustrates the role of the comma as an operator. The output of each program is appended. Consider first Listing 7.

/* Program 5.6; file name: unit5-prog6.c */
#include <stdio.h>
int 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);
    return (0);
}

Listing 7: An example of role of comma operator.

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 \( x = 1 \)

\[
x = 3
\]

\[
x = 2, y = 3, z = 1
\]

The fact that \( x \) gets the value 4 in the third line of output is intriguing. It implies that the denominator was evaluated after the numerator, in which \( x \) was set equal to 2. However, this order is purely compiler dependent. This output was obtained after compiling the program using gcc. The order may be different in a different compiler, i.e. the numerator may be evaluated after the denominator. 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. Listing 8 below was executed in VAX C and on a PC with ANSI C. Note the differences in the outputs.

/* Program 5.7; file name = unit5-prog8.c */
#include <stdio.h>
int 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);
  return (0);
}

Listing 8: Non-unique order of evaluation in printf().

Program 5.7: Output from VAX C
x = 3, y = -4, z = 0
x = 6, y = 5, z = 1
x = 6, y = 6, z = 1

Program 5.7: 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 languages.

E2) Modify Program 5.4 so that it continues 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.

Often, we may want to terminate a loop in between or to transfer the control outside the loop. We will see how to do this in the next section.

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.

Listing 9 illustrates how the break statement enables control to be transferred outside a while() loop. It computes the month in which, beginning at month 2, rabbits reproducing according to the Fibonacci formula will exceed a limit scanned from the keyboard.

/* Program 5.10; file name:unit5-progl0*/
#include <stdio.h>
int main ()
{
    int fib1 = 1, fib2 = 1, fib3, limit, month = 2;
    do
    {
        printf("Upper limit for fibonacci's \n rabbits? (must exceed 1): ");
        scanf("%d", &limit);
    }
    while (limit <= 1);
    while (fib3 = fib1 + fib2)
    {
        month ++;
        if (fib3 >= limit)
            break;
        fib1 = fib2;
        fib2 = fib3;
    }
    printf("Limit is reached or exceeded \n in month: %d. \n", month);
    return (0);
}

Listing 9: The break statement.

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 Listing 10, 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;

/***********************************************************/
#include <stdio.h>
int 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; \n square root = %d. \n", input, loop_index);
    else
        printf("The square root of %d \n lies between %d and %d. \n", input, loop_index - 1, loop_index);
    return (0);
}

Listing 10: Example of break statement.
The `continue` statement causes `loop_index` to be incremented for as long as `loop_index * loop_index < input`; the square root of `input` has been 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 I 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 shouldn't `b` range from 0 through 9?) Listing 11 solves the problem.

```c
/* Program 5.12; file name:unit5-prog12.c */
#include <stdio.h>

int 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 ==
                            1100 * a + 11 * b)
                            goto writeanswer;
                        else
                            break;
                }
                b ++;
            }
        a ++;
    }

writeanswer: printf("Car's number is : \
%d\n", 1100 * a + 11 * b);
    return (0);
}
```

Listing 11: Use of `break` statement.
E3) How many terms of the series:

\[1 \times 1 + 2 \times 2 + 3 \times 3 + \ldots + n \times n\]

must be summed before the total exceeds 25,000?

E4) Referring to Listing 11 on the preceding page, why should the variable \texttt{sqrt} begin at 33 in the innermost \texttt{while()} loop?

E5) Rewrite the program in Listing 11 on the facing page without using a \texttt{goto} statement.

We close this section here. In the next section, we will discuss \texttt{if-then-else} operator.

5.5 THE (TERNARY) \texttt{if()} - then - else OPERATOR

The \texttt{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:

\texttt{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 \texttt{second\_exp} if \texttt{first\_exp} is \texttt{true}, else it returns the value of \texttt{third\_exp}. Hence its name: the \texttt{it - then - else} operator. In other words, the ternary operator returns a value according to the formula:

\texttt{Boolean\_expression \? this\_value\_if\_true : else\_this}

Semantically, therefore, the statement which assigns one of two values to a variable \texttt{x} via the ternary operator:

\texttt{x = first\_exp \? second\_exp : third\_exp;}

is equivalent to:

\texttt{if (first\_exp)}
\texttt{x = second\_exp;}
\texttt{else}
\texttt{x = third\_exp;}

Often there's little to choose between the two alternative mechanisms of assigning a value to \texttt{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 \texttt{second\_exp} and \texttt{third\_exp} are expressions of different types, the value returned by the ternary operator has the wider type, irrespective of whether \texttt{first\_exp} was \texttt{true} or \texttt{false}. For example:

\[x = (5 > 3) \? (\text{int} \ 2.3 : (\text{float} \ 5);\]

will return to \texttt{x} the value 2.0, rather than 2.

E6) Give the output of the following program:

```c
/* Program 5.13; file name:unit5-prog13.c */
#include <stdio.h>

int 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");
    return (0);
}```
E7) Would the output of the program above be different if the cast operator (float) was removed from the third operand of the ternary operator?

E8) Observe that the following statement 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 parentheses 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 val_1 and val_2. (For the sake of computational efficiency, it is necessary to determine the lesser of the multiplier and the multiplicand which from 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 31, L = 19, G = 31, 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.

The program in Listing 12 below illustrates the use of the ternary operator to encode the Russian Peasant Algorithm:

```
/* Program 5.14; file name:unit5-progl4.c */
#include <stdio.h>
int main()
{
    int val_1, val_2, lesser, greater, result = 0;
    printf("Russian Peasant Multiplication Algorithm\n");
    printf("\nEnter multiplier: ");
    scanf("%d", &val_1);
    printf("\nEnter multiplicand: ");
    scanf("%d", &val_2);
    
```
greater = (lesser = val_1 < val_2 ? val_1 : val_2)
== val_1 ? val_2 : val_1;
while (lesser)
{
    result += lesser % 2 ? greater : 0;
    lesser /= 2;
    greater *= 2;
}
printf("%d\n", result);
return (0);

Listing 12: Russian Peasant Algorithm.

E9) Write a C program which uses the ternary operator to print -1, 0 or 1 if the value input to it is negative, zero or positive.

E10) Execute the program listed below, and explain its output.

/* Program 5.15; file name: unit5-prog15.c */
#define TRUE 1
#define FALSE 0
#define T "true"
#define F "false"
#include <stdio.h>
int 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;
    }
    return (0);
}

(Note that the Program 5.15 contains two while() loops, one nested fully inside the other. The outer loop is executed for each value of i, that i, 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.)

E11) Write a C language program to print a truth table for the Boolean expression i && (j || k), where i, j and k range from FALSE to TRUE.

We end this section here. In the next section we will discuss the switch - case - default statement.

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() - else may be used in this sort of situation. But if() - else become cumbersome when the number of alternative exceeds three or four; then the switch -
**Introduction to the C Programming Language**

**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:

\[ Zeller = ((\text{int}) ((13 \times \text{Month} - 1) / 5) + \text{Day} + \text{Year} \% 100 + (\text{int}) ((\text{Year} \% 100) / 4) - 2 \times (\text{int}) (\text{year} / 100) + (\text{int}) (\text{Year} / 400) + 91) \% 7 \]

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:

```c
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:

```c
switch (discrete variable or expression)
{
    case val_1 : statements;
        break;
    case val_2 : statements;
        break;
    case val_3 : statements;
        break;
    ...
    case val_n : statements;
        break;
    default   : statements;
        break;
}
```

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

```c
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;
}

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 variable 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 is not there, control would fall through to execute the next case. This property of the case may useful when multiple values of the 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, val_2 or val_3, control will reach the corresponding case and fall 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 Listing 13 on the following page is the scanf() statement written as a component 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 scanf() contains non-format character—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:

```c
if (Month < 3)
    ZMonth = Month + 10;
else
    ZMonth = Month - 2;
if (ZMonth < 10)
    ZMonth = 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 the printf() statements with the cases.

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

```c
exit(n)
```
closes all files that may be open, flushes memory buffers, and returns the value n, the exit status, to be 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; file name: unit5-prog16.c */
#include <stdio.h>
#include <stdlib.h>
#define LEAP_YEAR (Year % 4 == 0 && Year % 100 != 0) || Year % 400 == 0
int main()
{
    int Day, Month, Year, Zeller, ZMonth, ZYear;
    printf("nThis program finds the day \ncorresponding to a given date.\n");
    printf("nEnter date, month, year... \nformat is dd-mm-yyyy.\n");
    printf("nEnter a 1 or 2-digit number \nfor day, followed by a\n");
    printf("ndash, followed by a 1 or 2-digit \nnumber for month,\n");
    printf("nfollowed by a dash, followed by a 2 or 4-digit number\n");
    printf("nfor the year. Valid year range \nis 1582-4902, inclusive.\n");
    printf("n(A 2-digit number for the year will \nimply 20th century\n");
    printf("nyears.\n\n\na Enter dd-mm-yyyy: ");
    while (scanf("%d-%d-%d", &Day, &Month, &Year) != 3) {
        printf("nInvalid number of arguments \nor hypens mismatched\n");
    }
    
```
if (Year < 0)
{
    printf("Invalid year value...Program aborted..");
    exit(1);
}

if (Year < 100)
    Year += 1900;
if (Year < 1582 || Year > 4902)
{
    printf("Invalid year value...Program aborted..");
    exit(1);
}

if (!LEAP_YEAR) && (Month == 2) && (Day > 28))
{
    printf("Invalid date...Program aborted..");
    exit(1);
}

if ((LEAP_YEAR) && (Month == 2) && (Day > 29))
{
    printf("Invalid date...Program aborted..");
    exit(1);
}

if (Month < 1 || Month > 12)
{
    printf("Invalid month...Program aborted..");
    exit(1);
}

if (Day < 1 || Day > 31)
{
    printf("Invalid date...Program aborted..");
    exit(1);
}

if ((Day > 30) && (Month == 4 || Month == 6 ||
    Month == 9 || Month == 11))
{
    printf("Invalid date...Program aborted..");
    exit(1);
}

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

Zeller = ((int) ((13 * ZMonth - 1) / 5) + Day + ZYear % 100 +
(int) ((ZYear % 100) / 4) - 2 * (int) (ZYear / 100) +
(int) (ZYear / 400) + 91) % 7;
printf("\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;
}
return (0);

Listing 13: Zeller’s formula.

A long replacement string of a \texttt{\#define} may be continued into the next line by placing a backslash at the end of the part of the string in the current line. (Reread the third and fourth lines of Listing 13.)

\begin{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.
\end{note}

E12) In Listing 13 why is it preferable to write
\begin{verbatim}
if ( ((Day > 30) && (Month == 4 || Month == 6 || Month == 9
    || Month == 11))
etc. instead of
if ( ((Month == 4 || Month == 6 || Month == 9 || Month == 11) && (Day > 30))
etc?
\end{verbatim}

E13) Is the cast operator (\texttt{\textbf{int}}) required in the expression for Zeller in Listing 13? Rewrite the expression without the case operator, and without redundant parentheses.

E14) Since 91 is exactly divisible by 7, is its presence required in the expression for Zeller in Listing 13? Explain why or why not.

E15) In Listing 13, is \texttt{! (LEAP\_YEAR)} different from \texttt{! LEAP\_YEAR}?

E16) The \texttt{switch - case - default} statement is not always a better choice than the \texttt{if()} - \texttt{else}, even when there may be several cases to include in a program. Rewrite the following program, which scans a \texttt{char} input and determines its hexadecimal value (if it has one) by using a \texttt{switch} instead of the \texttt{if() - else}.

\begin{verbatim}
/* Program 5.17 */
#include <stdio.h>
int main()
\end{verbatim}
{ char digit;
 printf("Enter hex digit: ");
 scanf("%c", &digit);
 if (digit == 'a' || digit == 'f')
   printf("Decimal value of hex digit \n is %d.\n", digit = digit - 'a' + 10);
 else
   if (digit >= 'A' && digit <= 'F')
     printf("Decimal value of hex digit \n is %d.\n", digit = digit - 'A' + 10);
   else
     if (digit >= '0' && digit <= '9')
       printf("Decimal value of hex digit \n is %d.\n", digit - '0');
     else
       printf("You typed a non-hex digit.\n");
 return (0);
}

E17) Compile and execute the program below and state its output:

/* Program 5.18; file name: unit5-prog18.c */
#include <stdio.h>
int main()
{
 printf("%s", "How many strings do we have?");
 return (0);
}

In Program 5.17 note that the alphabetical hex digits may be entered both in lowercase 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 situation; it returns the uppercase equivalent of its character argument (if it was a lowercase character) or its argument itself if it wasn't. The values of its argument remains unaltered. Thus toupper('x') returns 'X', toupper('?') returns '?'. To use the toupper() function, #include the <ctype.h> just as you do the file <stdio.h>. See Listing 14 below.

/* Program 5.19; file name: unit5-prog19.c */
#include <stdio.h>
#include <ctype.h>
int 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 = digit - 'a' + 10: digit = 'A' + 10);
 else
   if (digit >= '0' && digit <= '9')
     printf("Decimal value of hex digit \n is %d.\n", digit - '0');
   else
     printf("You typed a non-hex digit.\n");
 return (0);
}
printf("You typed a non-hex digit.\n");
    return (0);
}

Listing 14: Using function toupper().

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 hexadecimal digit

Note that the <ctype.h> function isxdigit (c) makes Program 5.17 and 5.19 somewhat superfluous!

In the program in Listing 15 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 is 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 = getchar ()) != EOF)

note that

c = getchar ()

is performed before 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 the DOS environment on a PC, EOF is sent by pressing the CTRL and Z keys together. In the bash shell in Linux EOF is sent by pressing the CTRL and D keys together.

/* Program 5.20; file name:unit5-prog20.c */
#include <stdio.h>
int main ()
{
    int c;
    long keystrokes = 0, spaces = 0, punct_marks = 0,
    lines = 0, vowels = 0;
    printf("Enter text, line by line, and I'll give you some\nstatistics about it...\nterminate your input by entering\nCTRL-Z as the first character of a newline...\nthat's the\nDOS EOF character (may be different in your computing \nenvironment);\n\n");

    while ((c = getchar ()) != EOF)
    {
        switch (c) {
            case ' ':     // space
                spaces ++;
                break;
            case '
':     // newline
                lines ++;
                keystrokes ++;
                break;
            case '	':    // horizontal tab
                keystrokes ++;
                break;
            case '\r':  // formfeed
                keystrokes ++;
                break;
            case '\n':  // carriage return
                keystrokes ++;
                break;
            case 'a':    // vowel
                vowels ++;
                break;
            case 'e':    // vowel
                vowels ++;
                break;
            case 'i':    // vowel
                vowels ++;
                break;
            case 'o':    // vowel
                vowels ++;
                break;
            case 'u':    // vowel
                vowels ++;
                break;
        }
    }
    printf("Keystrokes: %ld\nSpaces: %ld\nPunctuation marks: %ld\nLines: %ld\nVowels: %ld\n\n", keystrokes, spaces, punct_marks, lines, vowels);
    return 0;
}
while ((c = getchar()) != EOF)
{
    switch (c)
    {
    case 'n': ++lines;
                keystrokes ++;
                break;
    case 't':
    case ' ': spaces ++;
                keystrokes ++;
                break;
    case ',':
    case ';':
    case ':':
    case ';':
    case ';':
    case ';':
    case '!':
    case '?': punct_marks ++;
                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;
    }
}

printf("Input statistics:\n\nlines = %ld,\nkeystrokes = %ld,\nvowels = %ld,\npunctuation marks = %ld, spaces = %ld\n", lines,
keystrokes, vowels, punct_marks, spaces);
return (0);
}

Listing 15: Program to count the number of characters.

The switch statement is useful only when the cases are controlled by integer values; in a program where the conditions to control branching are set in terms of floating point values, the if() - else statement must be used.

5.7 SUMMARY

In this unit we have

1. discussed the while() and do-while() loops.
In the while() loop the expression is evaluated before entering the loop. In the do-while() loop, the expression is evaluated at the end of the loop. So, the do-while() loop is used if the loop has to be executed at least once.

2. discussed the ternary operator if-then-else

\[
x = \text{condition} ? \text{first_val} : \text{second_val};
\]

In this case the operator tests the condition and return one of the two values, the first if the condition is true and the second if it is not.

3. discussed the comma operator than can be used to control the order of evaluation of expressions.

5.8 SOLUTIONS/ANSWERS

E1) /*File name: unit5-ans-ex1.c*/
#include <stdio.h>
int main()
{
    int a, p, b, i = 1;
    printf("Enter p,a\n");
    scanf("%d,%d", &p, &a);
    b = a;
    while (a != 1)
    {
        a = b;
        a %= p;
        i++;
    }
    printf("The order is %d.,i);
    return (0);
}

E2) /* File name: unit5-ans-ex2.c */
#include <stdio.h>
int main()
{
    int fib1, fib2, pairs, loop_index = 3, month;
    printf("Rabbit pairs produced in which month?...: ");
    scanf("%d", &month);
    if (month == 1 || month == 2)
        pairs = 1;
    else
    {
        fib1 = fib2 = 1;
        do
        {  
            fib1 = fib2 = 1;  
            fib2 = fib2 + fib1;  
        }  
        while (loop_index != 0);
    }
    printf("The pairs are: %d", pairs);
    return (0);
}
while (++loop_index <= month);

} /*End else*/

printf("The number of pairs produced in \nmonth %d is %d.\n", month.pairs);
return (0);
}

E3) /*File name:unit5-ans-ex3.c*/
#include <stdio.h>
#define LIMIT 25000
int main ()
{
  int sum = 1, i=1;
  while (sum <= LIMIT)
  {
    i++;
    sum += i*i;
  }
  printf("The sum exceeds %d when \n" , LIMIT, i);
  return (0);
}

E4) Observe that the number itself is divisible by 11 and must be a square. The small number divisible by 11 whose square is also a 4 digit number is 33.

E5) /* Program 5.12; file name:unit5-ans-ex4.c */
#include <stdio.h>
int main ()
{
  int a = 1, b, sqroot;
  while (a <= 9)
  {
    b = 1;
    while (b <= 9)
    {
      sqroot = 32;
      while (sqroot ++ < 99)
      {
        if (sqroot * sqroot <
            1100 * a + 11 * b)
          continue;
        else
          if(sqroot * sqroot ==
            1100 * a + 11 * b)
            printf("Car's number is : \n" %d\n", 1100 * a + 11 * b);
          else
            break;
      }
      b ++;
    }
    a ++;
  }
  return (0);
}

E6) Since the value of the expression is 2, it will print the first statement.
As mentioned, `val_1 < val_2` **less** will get the value **val_1**, otherwise, it will get the value **val_2**. If the value of **less** is **val_1** `less` `val_1 < val_2`? **val_1 : val_2)` == **val_1** will have the value 1, so **greater** will get the value **val_2**. Otherwise, the value will be zero, i.e. **val_1** is the bigger number and **greater** will get the value **val_1**. The outer parentheses are not necessary.

```c
#include <stdio.h>
int main ()
{   int a, b;
    printf("Enter a number...
\n");
    scanf("%d", &a);
    b = (a < 0? -1:1)* (a == 0?0:1);
    printf("%d", b);
    return (0);
}
```

This program outputs the truth table of the Boolean expression `i && j` where `i`, `j` range from FALSE to TRUE.

```c
/* Program 5.15; file name: unit5-prog15.c */
#define TRUE 1
#define FALSE 0
#define T "true"
#define F "false"
#include <stdio.h>
int main()
{    int i = FALSE, j = FALSE, k = FALSE;
    while (i <= TRUE)
    {
        while (j <= TRUE)
        {
            while (k <= TRUE)
            {
                printf("%s && (%s || %s) equals %s\n",  
                        k++;
            }  
            k = FALSE;
            j++;
        }  
        j = FALSE;
        i++;
    }
    return (0);
}
```

If Day is less than 30, in the first case, the program will not check the month. In the second case, it will check the month and the the Day also.

```c
Zeller = ((13 * ZMonth - 1) / 5 + Day + ZYear % 100 +  
        ZYear % 100) / 4 - 2 * (ZYear / 100) +  
        (ZYear / 400) + 91) % 7
```

`91` is there to make sure that the answer is never negative. In the number is negative `%` operator gives negative answer. The worst case is Month = 2, Year = 4900 and Day = 1. The value of the expression will be -90 without the added multiple of 7. The smallest multiple of 7 bigger than -90 is 91 and we have added this number.
E14) Yes. In the first case, \(! (\text{LEAP\_YEAR}, !)\) is applied to the entire expression. In the second case, it is applied just to the first term,
(Year \% 4 == 0 \&\& Year \% 100 != 0) and so the expression becomes
\(! (Year \% 4 == 0 \&\& Year \% 100 != 0)\) || Year \% 400 == 0 which means the remainder of Year on division by 4 is 0 or the remainder of Year on division by 100 is 0 or the remainder of Year on division by 400 is 0!

E15) Try it! It will have 32 cases if you do not use \(\text{if}()\) or any other decision structure.

E16) \textbf{How many strings do we have?}