```
            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 `#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.)

> 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.

E12) In Listing 13 why is it preferable to write

```
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?

E13) Is the cast operator (**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 `!(LEAP_YEAR)` different from `!LEAP_YEAR`?

E16) 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() - else**s.

```
/* Program 5.17 */
#include <stdio.h>
int 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");
        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 >= 'a' ?digit - 'a' + 10: 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");
    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
ispunet	(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 and 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; int**s can include all chars in their range, as well as the `EOF`. A MACRO definition in `<stdio.h>` **#define**s a value for it. So **#includ**ing `<stdio.h>` makes `EOF` automatically avail able 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\n \
statistics about it...terminate your input by entering\n\
CTRL-Z as the first character of a newline...that\'s the\n\
DOS EOF character (may be different in your computing \
enviornment);\n\n");
```

```
    while ((c = getchar()) !=  EOF)
        {
         switch(c)
             {
              case '\n':  ++ lines;
                          keystrokes ++;
                          break;
              case '\t':
              case ' ': spaces ++;
                        keystrokes ++;
                        break;
              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 lines = %ld,\n keystrokes = %ld,\n vowels = %ld,\
\n punctuation marks = %ld, spaces = %ld\n", lines,
keystrokes, vowels, punct_marks, spaces);
    return (0);
}
```

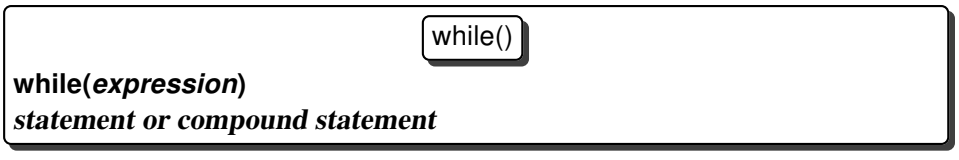
Listing 15: Program to count the number of charaters.

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.



do-while()

do
statement or compound statement
while()

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**

if-then-else

```
x = condition ?  first_val :  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
             {
              pairs = fib1 + fib2;
              fib1 = fib2;
              fib2 = pairs;
```

```
            }
               while( ++loop_index <= month);

        }/*End else*/
    printf("The number of pairs produced in \
month %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 \
i = %d", 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 : \
%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.

E7) As mentioned, `val_1 < val_2`, `lesser` will get the value `val_1`, otherwise, it will get the value `val_2`. If the value of lesser is `val_1 lesser = 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.

E8)
```
#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);
}
```

E9) This program outputs the truth table of the Boolean expression `i&&j` where `i, j` range from **FALSE** to **TRUE**.

E10)
```
/* 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",
i ? T : F, j ? T : F, k? T : F,i && (j||k) ? T : F);
                   k++;
                  }
               k = FALSE;
               j++;
             }
        i++;
        j = FALSE;
        }
    return (0);
}
```

E11) 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.

E12)
```
Zeller = ((13 * ZMonth - 1) / 5 + Day + ZYear % 100 +
ZYear % 100) / 4 - 2 *  (ZYear / 100) +
(ZYear / 400) + 91) % 7
```

E13) 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, `!(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 400 is 0!

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

E16) **Howmanystringsdowehave?**