Introduction to Programming (ECS 102)

Instructors:

Dr. Jasabanta Patro and Dr. Rini Smita Thakur

Control Statements

Recap Basics:

First, we will need to form logical expressions that are either true or false. To do so, we can use the four *relational operators*, `<`, `<=`, `>`, `>=`, and the two *equality operators*, `==` and `!=` (see Sec. 3.3).

EXAMPLE 6.1 Several logical expressions are shown below.

```
count <= 100

sqrt(a+b+c) > 0.005

answer == 0

balance >= cutoff

ch1 < 'T'

letter != 'x'
```

The first four expressions involve numerical operands. Their meaning should be readily apparent.

In the fifth expression, `ch1` is assumed to be a char-type variable. This expression will be true if the character represented by `ch1` comes before `T` in the character set, i.e., if the numerical value used to encode the character is less than the numerical value used to encode the letter `T`.

The last expression makes use of the char-type variable `letter`. This expression will be true if the character represented by `letter` is something other than `x`.

In addition to the relational and equality operators, C contains two *logical connectives* (also called *logical operators*), && (AND) and || (OR), and the *unary negation operator* ! (see Sec. 3.3). The logical connectives are used to combine logical expressions, thus forming more complex expressions. The negation operator is used to reverse the meaning of a logical expression (e.g., from true to false).

EXAMPLE 6.2 Here are some logical expressions that illustrate the use of the logical connectives and the negation operator.

```
(count <= 100) && (ch1 != '*')

(balance < 1000.0) || (status == 'R')

(answer < 0) || ((answer > 5.0) && (answer <= 10.0))

!((pay >= 1000.0) && (status == 's'))
```

Since the relational and equality operators have a higher precedence than the logical operators, some of the parentheses are not needed in the above expressions (see Table 3-1 in Sec. 3.5). Thus, we could have written these expressions as

```
count <= 100 && ch1 != '*'

balance < 1000.0 || status == 'R'

answer < 0 || answer > 5.0 && answer <= 10.0

!(pay >= 1000.0 && status == 's')
```

Table 3-1 Operator Precedence Groups

Operator category	*Operators*	*Associativity*
unary operators	- ++ -- ! sizeof (*type*)	R → L
arithmetic multiply, divide and remainder	* / %	L → R
arithmetic add and subtract	+ -	L → R
relational operators	< <= > >=	L → R
equality operators	== !=	L → R
logical *and*	&&	L→ R
logical *or*	||	L → R
conditional operator	? :	R → L
assignment operators	= += -= *= /= %=	R → L

The *conditional operator* `?:` also makes use of an expression that is either true or false (see Sec. 3.5). An appropriate value is selected, depending on the outcome of this logical expression. This operator is equivalent to a simple *if - else* structure (see Sec. 6.6).

EXAMPLE 6.3 Suppose `status` is a char-type variable and `balance` is a floating-point variable. We wish to assign the character `C` (current) to `status` if `balance` has a value of zero, and `O` (overdue) if `balance` has a value that is greater than zero. This can be accomplished by writing

```
status = (balance == 0) ? 'C' : 'O'
```

Various types of control statements

- Branching
- Looping
- Switch
- Break
- Goto ..

6.2 BRANCHING: THE `if - else` STATEMENT

The `if - else` statement is used to carry out a logical test and then take one of two possible actions, depending on the outcome of the test (i.e., whether the outcome is true or false).

The `else` portion of the `if - else` statement is optional. Thus, in its simplest general form, the statement can be written as

```
if (expression) statement
```

The *expression* must be placed in parentheses, as shown. In this form, the *statement* will be executed only if the *expression* has a nonzero value (i.e., if *expression* is true). If the *expression* has a value of zero (i.e., if *expression* is false), then the *statement* will be ignored.

The *statement* can be either simple or compound. In practice, it is often a compound statement which may include other control statements.

EXAMPLE 6.5 Several representative if statements are shown below.

```
if (x < 0) printf("%f", x);

if (pastdue > 0)
   credit = 0;

if (x <= 3.0)   {
   y = 3 * pow(x, 2);
   printf("%f\n", y);
}

if ((balance < 1000.) || (status == 'R'))
   printf("%f", balance);

if ((a >= 0) && (b <= 5))   {
   xmid = (a + b) / 2;
   ymid = sqrt(xmid);
}
```

The general form of an `if` statement which includes the `else` clause is

```
if (expression) statement 1 else statement 2
```

If the *expression* has a nonzero value (i.e., if *expression* is true), then *statement 1* will be executed. Otherwise (i.e., if *expression* is false), *statement 2* will be executed.

EXAMPLE 6.6 Here are several examples illustrating the full `if - else` statement.

```
if (status == 'S')
   tax = 0.20 * pay;
else
   tax = 0.14 * pay;

if (pastdue > 0)   {
   printf("account number %d is overdue", accountno);
   credit = 0;
}
else
   credit = 1000.0;
```

```
if (x <= 3)
   y = 3 * pow(x, 2);
else
   y = 2 * pow(x - 3), 2);
printf("%f\n", balance);

if (circle)  {
   scanf("%f", &radius);
   area = 3.14159 * radius * radius;
   printf("Area of circle = %f", area);
}
else  {
   scanf("%f %f", &length, &width);
   area = length * width;
   printf("Area of rectangle = %f", area);
}
```

Following are the same

```
if (status == 'S')
   tax = 0.20 * pay;
else
   tax = 0.14 * pay;
```

```
tax = (status == 'S') ? (0.20 * pay) : (0.14 * pay);
```

It is possible to *nest* (i.e., embed) `if - else` statements, one within another. There are several different forms that nested `if - else` statements can take. The most general form of two-layer nesting is

```
if e1 if e2 s1
      else s2
else  if e3 s3
      else s4
```

Some other forms of two-layer nesting are

```
if e1 s1
else if e2 s2

if e1 s1
else if e2 s2
     else s3

if e1 if e2 s1
      else s2
else s3

if e1 if e2 s1
      else s2
```

```
if e1 if e2 s1
       else s2
```

```
if e1  {
   if e2 s1 else s2
}
```

If we wanted to associate the `else` clause with *e1* rather than *e2*, we could do so by writing

```
if e1  {
   if e2 s1
}
else s2
```

In some situations it may be desirable to nest multiple `if - else` statements, in order to create a situation in which one of several different courses of action will be selected. For example, the general form of four nested `if - else` statements could be written as

```
if e1 s1
else if e2 s2
     else if e3 s3
          else if e4 s4
               else s5
```

EXAMPLE 6.7 Here is an illustration of three nested `if - else` statements.

```
if ((time >= 0.) && (time < 12.)) printf("Good Morning");
else if ((time >= 12.) && (time < 18.)) printf("Good Afternoon");
     else if ((time >= 18.) && (time < 24.)) printf("Good Evening");
          else printf("Time is out of range");
```

6.3 LOOPING: THE `while` STATEMENT

The `while` statement is used to carry out looping operations, in which a group of statements is executed repeatedly, until some condition has been satisfied.

The general form of the `while` statement is

```
while (expression) statement
```

The *`statement`* will be executed repeatedly, as long as the *`expression`* is true (i.e., as long *`expression`* has a nonzero value). This *`statement`* can be simple or compound, though it is usually a compound statement. It must include some feature that eventually alters the value of the *`expression`*, thus providing a stopping condition for the loop.

EXAMPLE 6.8 Consecutive Integer Quantities Suppose we want to display the consecutive digits 0, 1, 2, . . . , 9, with one digit on each line. This can be accomplished with the following program.

```
#include <stdio.h>

main()     /* display the integers 0 through 9 */

{
    int digit = 0;

    while (digit <= 9)  {
          printf("%d\n", digit);
          ++digit;
    }
}
```

0

1

2

3

4

5

6

7

8

9

This program can be written more concisely as

```
#include <stdio.h>

main()      /* display the integers 0 through 9 */
{
    int digit = 0;

    while (digit <= 9)
          printf("%d\n", digit++);
}
```

```
/* convert a line of lowercase text to uppercase */

#include <stdio.h>
#include <ctype.h>

#define EOL   '\n'

main()
{
    char letter[80];
    int tag, count = 0;


 /* read in the lowercase text */
 while ((letter[count] = getchar()) != EOL)  ++count;
 tag = count;

 /* display the uppercase text */
 count = 0;
 while (count < tag)  {
       putchar(toupper(letter[count]));
       ++count;
 }
}
```

```
/* calculate the average of n numbers */

#include <stdio.h>

main()

{
    int n, count = 1;
    float x, average, sum = 0;

    /* initialize and read in a value for n */
    printf("How many numbers? ");
    scanf("%d", &n);

    /* read in the numbers */
    while (count <= n)  {
          printf("x = ");
          scanf("%f", &x);
          sum += x;
          ++count;
    }

    /* calculate the average and display the answer */
    average = sum/n;
    printf("\nThe average is %f\n", average);
}
```

6.4 MORE LOOPING: THE do - while STATEMENT

When a loop is constructed using the `while` statement described in Sec. 6.3, the test for continuation of the loop is carried out at the *beginning* of each pass. Sometimes, however, it is desirable to have a loop with the test for continuation at the *end* of each pass. This can be accomplished by means of the `do - while` statement.

The general form of the `do - while` statement is

```
do statement while (expression);
```

EXAMPLE 6.11 Consecutive Integer Quantities In Example 6.8 we saw two complete C programs that use the `while` statement to display the consecutive digits 0, 1, 2, . . . , 9. Here is another program to do the same thing, using the do - `while` statement in place of the `while` statement.

```
#include <stdio.h>

main()    /* display the integers 0 through 9 */
{
    int digit = 0;

    do
    printf("%d\n", digit++);
    while (digit <= 9);
}
```

```
/* convert a line of lowercase text to uppercase */

#include <stdio.h>
#include <ctype.h>

#define EOL .'\n'

main()
{
    char letter[80];
    int tag, count = -1;

    /* read in the lowercase text */
    do ++count; while ((letter[count] = getchar()) != EOL);
    tag = count;


    /* display the uppercase text */
    count = 0;
    do {
            putchar(toupper(letter[count]));
            ++count;
    }  while (count < tag);
}
```

```
/* calculate the average of n numbers */

#include <stdio.h>

main()
{
    int n, count = 1;
    float x, average, sum = 0;

    /* initialize and read in a value for n */
    printf("How many numbers? ");
    scanf("%d", &n);


  /* read in the numbers */
  do {
        printf("x = ");
        scanf("%f", &x);
        sum += x;
        ++count;
  }  while (count <= n);

  /* calculate the average and display the answer */
  average = sum/n;
  printf("\nThe average is %f\n", average);
}
```

6.5 STILL MORE LOOPING: THE `for` STATEMENT

The general form of the `for` statement is

```
for (expression 1; expression 2; expression 3) statement
```

where *`expression 1`* is used to initialize some parameter (called an *index*) that controls the looping action, *`expression 2`* represents a condition that must be true for the loop to continue execution, and *`expression 3`* is used to alter the value of the parameter initially assigned by *`expression 1`*. Typically, *`expression 1`* is an assignment expression, *`expression 2`* is a logical expression and *`expression 3`* is a unary expression or an assignment expression.

When the `for` statement is executed, *`expression 2`* is evaluated and tested at the *beginning* of each pass through the loop, and *`expression 3`* is evaluated at the *end* of each pass. Thus, the `for` statement is equivalent to

```
expression 1;
while (expression 2)  {
   statement
   expression 3;
}
```

```c
#include <stdio.h>

main()   /* display the numbers 0 through 9 */

{
    int digit;

    for (digit = 0; digit <= 9; ++digit)
        printf("%d\n", digit);
}
```

From a syntactic standpoint all three expressions need not be included in the `for` statement, though the semicolons must be present. However, the consequences of an omission should be clearly understood. The first and third expressions may be omitted if other means are provided for initializing the index and/or altering the index. If the second expression is omitted, however, it will be assumed to have a permanent value of 1 (true); thus, the loop will continue indefinitely unless it is terminated by some other means, such as a `break` or a `return` statement (see Secs. 6.8 and 7.2). As a practical matter, most `for` loops include all three expressions.

EXAMPLE 6.15 Consecutive Integer Quantities Revisited Here is still another example of a C program that generates the consecutive integers 0, 1, 2, . . . , 9, with one digit on each line. We now use a `for` statement in which two of the three expressions are omitted.

```
#include <stdio.h>

main()   /* display the numbers 0 through 9 */

{
    int digit = 0;

    for (; digit <= 9; )
        printf("%d\n", digit++);
}
```

```
/* convert a line of lowercase text to uppercase */

#include <stdio.h>
#include <ctype.h>

#define EOL   '\n'

main()

{
    char letter[80];
    int tag, count;

    /* read in the lowercase text */
    for (count = 0; (letter[count] = getchar()) != EOL; ++count)
    ;
    tag = count;

    /* display the uppercase text */
    for (count = 0; count < tag; ++count)
        putchar(toupper(letter[count]));
}
```

```
/* calculate the average of n numbers */

#include <stdio.h>

main()

{
    int n, count;
    float x, average, sum = 0;

    /* initialize and read in a value for n */
    printf("How many numbers? ");
    scanf("%d", &n);


   /* read in the numbers */
   for (count = 1; count <= n; ++count)  {
       printf("x = ");
       scanf("%f", &x);
       sum += x;
   }

   /* calculate the average and display the answer */
   average = sum/n;
   printf("\nThe average is %f\n", average);
}
```

6.6 NESTED CONTROL STRUCTURES

Loops, like `if - else` statements, can be *nested*, one within another. The inner and outer loops need not be generated by the same type of control structure. It is essential, however, that one loop be completely embedded within the other — there can be no overlap. Each loop must be controlled by a different index.

Moreover, nested control structures can involve both loops and `if - else` statements. Thus, a loop can be nested within an `if - else` statement, and an `if - else` statement can be nested within a loop. The nested structures may be as complex as necessary, as determined by the program logic.

```
/* calculate averages for several different lists of numbers */

#include <stdio.h>

main()

{
    int n, count, loops, loopcount;
    float x, average, sum;

    /* read in the number of lists */
    printf("How many lists? ");
    scanf("%d", &loops);


    /* outer loop (process each list of numbers */
    for (loopcount = 1; loopcount <= loops; ++loopcount)  {

        /* initialize and read in a value for n */
        sum = 0;
        printf("\nList number %d\nHow many numbers? ", loopcount);
        scanf("%d", &n);

        /* read in the numbers */
        for (count = 1; count <= n; ++count)  {
            printf("x = ");
            scanf("%f", &x);
            sum += x;
        }    /* end inner loop */

        /* calculate the average and display the answer */
        average = sum/n;
        printf("\nThe average is %f\n", average);

    }    /* end outer loop */
}
```

```
How many lists? 3

List number 1
How many numbers? 4
x = 1.5
x = 2.5
x = 6.2
x = 3.0

The average is 3.300000

List number 2
How many numbers? 3
x = 4
x = -2
x = 7

The average is 3.000000

List number 3
How many numbers? 5
x = 5.4
x = 8.0
x = 2.2
x = 1.7
x = -3.9

The average is 2.680000
```

```
/* convert several lines of text to uppercase
   continue the conversion until the first character in a line is an asterisk (*) */

#include <stdio.h>
#include <ctype.h>

#define EOL '\n'

main()
{
    char letter[80];
    int tag, count;

    while((letter[0] = getchar()) != '*')   {

        /* read in a line of text */
        for (count = 1; (letter[count] = getchar()) != EOL; ++count)
            ;
        tag = count;



        /* display the line of text */
        for (count = 0; count < tag; ++count)
            putchar(toupper(letter[count]));
        printf("\n\n");
}    /* end outer loop */

    printf("Good bye");
}
```

A typical interactive session, illustrating the execution of the program, is shown below. Note that the input text supplied by the user is underlined, as usual.

```
Now is the time for all good men to come to the aid . . .
NOW IS THE TIME FOR ALL GOOD MEN TO COME TO THE AID . . .

Fourscore and seven years ago our fathers brought forth . . .
FOURSCORE AND SEVEN YEARS AGO OUR FATHERS BROUGHT FORTH . . .

*
Good bye
```

EXAMPLE 6.20 Encoding a String of Characters Let us write a simple C program that will read in a sequence of ASCII characters and write out a sequence of encoded characters in its place. If a character is a letter or a digit, we will replace it with the next character in the character set, except that Z will be replaced by A, z by a, and 9 by 0. Thus 1 becomes 2, C becomes D, p becomes q, and so on. Any character other than a letter or a digit will be replaced by a period (.).

```
/* read in a string, then replace each character with an equivalent encoded character */

#include <stdio.h>

main()

{
    char line[80];
    int count;

    /* read in the entire string */

    printf("Enter a line of text below:\n");
    scanf("%[^\n]", line);


     /* encode each individual character and display it */

     for (count = 0; line[count] != '\0'; ++count)  {
         if (((line[count] >= '0') && (line[count] < '9')) ||
             ((line[count] >= 'A') && (line[count] < 'Z')) ||
             ((line[count] >= 'a') && (line[count] < 'z')))
                  putchar(line[count] + 1);
         else if (line[count] == '9') putchar('0');
              else if (line[count] == 'Z') putchar('A');
                   else if (line[count] == 'z') putchar('a');
                        else putchar('.');
     }
 }
```

Enter a line of text below:

The White House, 1600 Pennsylvania Avenue, Washington, DC

Uif.Xijuf.Ipvtf..2711.Qfootzmwbojb.Bwfovf..Xbtijohupo..ED

6.7 THE switch STATEMENT

The switch statement causes a particular group of statements to be chosen from several available groups. The selection is based upon the current value of an expression which is included within the switch statement.

The general form of the switch statement is

```
switch (expression) statement
```

where *expression* results in an integer value. Note that *expression* may also be of type char, since individual characters have equivalent integer values.

The embedded *statement* is generally a compound statement that specifies alternate courses of action. Each alternative is expressed as a group of one or more individual statements within the overall embedded *statement*.

In general terms, each group of statements is written as

```
case expression :
    statement 1
    statement 2
    . . . . .
    statement n
```

or, when multiple case labels are required,

```
case expression 1 :
case expression 2 :
    . . . . .
case expression m :
    statement 1
    statement 2
    . . . . .
    statement n
```

EXAMPLE 6.23 A simple `switch` statement is illustrated below. In this example, `choice` is assumed to be a char-type variable.

```
switch (choice = getchar())  {

case 'r':
case 'R':
     printf("RED");
     break;


case 'w':
case 'W':
     printf("WHITE");
     break;

case 'b':
case 'B':
     printf("BLUE");
}
```

Thus, `RED` will be displayed if `choice` represents either `r` or `R`, `WHITE` will be displayed if `choice` represents either `w` or `W`, and `BLUE` will be displayed if `choice` represents either `b` or `B`. Nothing will be displayed if any other character has been assigned to `choice`.

EXAMPLE 6.24 Here is a variation of the `switch` statement presented in Example 6.23.

```
switch (choice = toupper(getchar()))  {

case 'R':
     printf("RED");
     break;

case 'W':
     printf("WHITE");
     break;

case 'B':
     printf("BLUE");
     break;

default:
     printf("ERROR");
}
```

The `switch` statement now contains a `default` group (consisting of only one statement), which generates an error message if none of the case labels matches the original *`expression`*.

EXAMPLE 6.25 Here is another typical `switch` statement. In this example `flag` is assumed to be an integer variable, and `x` and `y` are assumed to be floating-point variables.

```
switch (flag)  {

case -1:
     y = abs(x);
     break;

case 0:
     y = sqrt(x);
     break;

case 1:
     y = x;
     break;

case 2:
case 3:
     y = 2 * (x - 1);
     break;

default:
     y = 0;
}
```

In a practical sense, the `switch` statement may be thought of as an alternative to the use of nested `if - else` statements, though it can only replace those `if - else` statements that test for equality. In such situations, the use of the `switch` statement is generally much more convenient.

6.8 THE `break` STATEMENT

The `break` statement is used to terminate loops or to exit from a switch. It can be used within a `for`, `while`, `do - while`, or `switch` statement.

The `break` statement is written simply as

```
break;
```

without any embedded expressions or statements.

EXAMPLE 6.27 Consider once again the `switch` statement originally presented in Example 6.24.

```
switch (choice = toupper(getchar()))  {

case 'R':
     printf("RED");
     break;

case 'W':
     printf("WHITE");
     break;

case 'B':
     printf("BLUE");
     break;

default:
     printf("ERROR");
     break;
}
```

If a `break` statement is included in a `while`, `do - while` or `for` loop, then control will immediately be transferred out of the loop when the `break` statement is encountered. This provides a convenient way to terminate the loop if an error or other irregular condition is detected.

EXAMPLE 6.28 Here are some illustrations of loops that contain `break` statements. In each situation, the loop will continue to execute as long as the current value for the floating-point variable `x` does not exceed 100. However, the computation will break out of the loop if a negative value for `x` is detected.

First, consider a `while` loop.

```
scanf("%f", &x);
while (x <= 100)  {
      if (x < 0)    {
         printf("ERROR - NEGATIVE VALUE FOR X");
         break;
      }

      /* process the nonnegative value of x */
      . . . . .
      scanf("%f", &x);
}
```

Now consider a do - while loop that does the same thing.

```
do  {
       scanf("%f", &x);
       if (x < 0)  {
          printf("ERROR - NEGATIVE VALUE FOR X");
          break;
    }

    /* process the nonnegative value of x */
    . . . . .
} while (x <= 100);
```

Finally, here is a `for` loop that is similar.

```
for (count = 1; x <= 100; ++count)   {
    scanf("%f", &x);
    if (x < 0)  {
       printf("ERROR - NEGATIVE VALUE FOR X");
       break;
    }

    /* process the nonnegative value of x */
    . . . . .
}
```

EXAMPLE 6.29 Consider the following outline of a `while` loop embedded within a `for` loop.

```
for (count = 0; count <= n; ++count)  {
    . . . . .
    while (c = getchar() != '\n')  {
          if (c = '*')  break;
          . . . . .
    }
}
```

If the character variable `c` is assigned an asterisk (`*`), then the `while` loop will be terminated. However, the `for` loop will continue to execute. Thus, if the value of `count` is less than `n` when the breakout occurs, the computer will increment `count` and make another pass through the `for` loop.

6.9 THE `continue` STATEMENT

The `continue` statement is used to *bypass* the remainder of the current pass through a loop. The loop does *not* terminate when a `continue` statement is encountered. Rather, the remaining loop statements are skipped and the computation proceeds directly to the next pass through the loop. (Note the distinction between `continue` and `break`.)

The `continue` statement can be included within a `while`, a `do - while` or a `for` statement. It is written simply as

```
continue;
```

EXAMPLE 6.30 Here are some illustrations of loops that contain `continue` statements.

First, consider a `do - while` loop.

```
do  {
       scanf("%f", &x);
       if (x < 0)  {
          printf("ERROR - NEGATIVE VALUE FOR X");
          continue;
    };

    /* process the nonnegative value of x */

       . . . . .
} while (x <= 100);
```

Here is a similar for loop.

```
for (count = 1; x <= 100; ++count)   {
    scanf("%f", &x);
    if (x < 0)  {
       printf("ERROR - NEGATIVE VALUE FOR X");
       continue;
    }


      /* process the nonnegative value of x */

      . . . . .
  }
```

In each case, the processing of the current value of x will be bypassed if the value of x is negative. Execution of the loop will then continue with the next pass.

```c
/* calculate the average of the nonnegative numbers in a list of n numbers */

#include <stdio.h>

main()

{
    int n, count, navg = 0;
    float x, average, sum = 0;

    /* initialize and read in a value for n */
    printf("How many numbers? ");
    scanf("%d", &n);

    /* read in the numbers */
    for (count = 1; count <= n; ++count)  {
        printf("x = ");
        scanf("%f", &x);
        if (x < 0) continue;
        sum += x;
        ++navg;
    }

    /* calculate the average and write out the answer */
    average = sum/navg;
    printf("\nThe average is %f\n", average);
}
```

```
How many numbers? 6
x = 1
x = -1
x = 2
x = -2
x = 3
x = -3

The average is 2.000000
```

This is the correct average of the positive numbers. Note that the average would be zero if all of the numbers had been averaged.

6.10 THE COMMA OPERATOR

We now introduce the comma operator (,) which is used primarily in conjunction with the `for` statement. This operator permits two different expressions to appear in situations where only one expression would ordinarily be used. For example, it is possible to write

```
for (expression 1a, expression 1b; expression 2; expression 3) statement
```

Similarly, a `for` statement might make use of the comma operator in the following manner.

```
for (expression 1; expression 2; expression 3a, expression 3b) statement
```

EXAMPLE 6.32 Searching for Palindromes A *palindrome* is a word, phrase or sentence that reads the same way either forward or backward. For example, words such as *noon*, *peep*, and *madam* are palindromes. If we disregard punctuation and blank spaces, then the sentence *Rise to vote, sir!* is also a palindrome.

Let us write a C program that will enter a line of text containing a word, a phrase or a sentence, and determine whether or not the text is a palindrome. To do so, we will compare the first character with the last, the second character with the next to last, and so on, until we have reached the middle of the text. The comparisons will include punctuation and blank spaces.

We can now outline our overall strategy as follows.

1. Define the symbolic constants EOL (end-of-line), TRUE and FALSE.
2. Declare all variables and initialize `loop` (i.e., assign TRUE to `loop`).
3. Enter the main loop.
   (*a*) Assign TRUE to `flag`, in anticipation of finding a palindrome.
   (*b*) Read in the line of text on a character-by-character basis, and store in `letter`.
   (*c*) Test to see if the uppercase equivalents of the first three characters are E, N and D, respectively. If so, break out of the main loop and exit the program.
   (*d*) Assign the final value of `count`, less 1, to `tag`. This value will indicate the number of characters in the line of text, not including the final escape character `\0`.
   (*e*) Compare each character in the first half of `letter` with the corresponding character in the second half. If a mismatch is found, assign FALSE to `flag` and break out of the (inner) comparison loop.
   (*f*) If flag is TRUE, display a message indicating that a palindrome has been found. Otherwise, display a message indicating that a palindrome has not been found.
4. Repeat step 3 (i.e., make another pass through the outer loop), thus processing another line of text.

Here is the corresponding pseudocode.

```
main()
{
   /* declare all variables and initialize as required */
   while (loop)  {
      flag = TRUE;   /* anticipating a palindrome */
      /* read in a line of text and store in letter */
      /* break out of while loop if first three characters
         of letter spell END (test uppercase equivalents) */
      /* assign number of characters in text to tag */
      for ((count = 0, countback = tag); count <= (tag - 1)/ 2;(++count, --countback)) {
          if (letter[count] != letter[countback])  {
             flag = FALSE;
             /* not a palindrome - break out of for loop */
          }
      }
      /* display a message indicating whether or not letter contains a palindrome */
   }
}
```

```
/* search for a palindrome */

#include <stdio.h>
#include <ctype.h>

#define EOL  '\n'
#define TRUE 1
#define FALSE 0

main()
{
    char letter[80];
    int tag, count, countback, flag, loop = TRUE;

    /* main loop */

    while (loop)   {
       flag = TRUE;

       /* read the text */

       printf("Please enter a word, phrase or sentence below:\n");
       for (count = 0; (letter[count] = getchar()) != EOL; ++count)
           ;
       if ((toupper(letter[0]) == 'E') && (toupper(letter[1]) == 'N') &&
           (toupper(letter[2]) == 'D')) break;
       tag = count - 1;

       /* carry out the search */

       for ((count = 0, countback = tag); count <= tag/2;
           (++count, --countback))   {

               if (letter[count] != letter[countback])   {
                  flag = FALSE;
                  break;
               }
       }
```

```
        /* display message */

        for (count = 0; count <= tag; ++count)
            putchar(letter[count]);
        if (flag) printf(" IS a palindrome\n\n");
        else printf(" is NOT a palindrome\n\n");
    }
}
```

```
Please enter a word, phrase or sentence below:
TOOT

TOOT IS a palindrome


Please enter a word, phrase or sentence below:
FALSE

FALSE is NOT a palindrome

Please enter a word, phrase or sentence below:
PULLUP

PULLUP IS a palindrome

Please enter a word, phrase or sentence below:
ABLE WAS I ERE I SAW ELBA

ABLE WAS I ERE I SAW ELBA IS a palindrome

Please enter a word, phrase or sentence below:
END
```

6.11 THE goto STATEMENT

The goto statement is used to alter the normal sequence of program execution by transferring control to some other part of the program. In its general form, the goto statement is written as

```
goto label;
```

where *label* is an identifier that is used to label the target statement to which control will be transferred.

Control may be transferred to any other statement within the program. (To be more precise, control may be transferred anywhere within the current *function*. We will introduce functions in the next chapter, and discuss them thoroughly in Chapter 7.) The target statement must be labeled, and the label must be followed by a colon. Thus, the target statement will appear as

```
label: statement
```

EXAMPLE 6.33 The following skeletal outline illustrates how the `goto` statement can be used to transfer control out of a loop if an unexpected condition arises.

```
/* main loop */

scanf("%f", &x);
while (x <= 100)   {
      . . . . .
      if (x < 0) goto errorcheck;
      . . . . .
      scanf("%f", &x);
}


   . . . . .

/* error detection routine */

errorcheck: {
              printf("ERROR - NEGATIVE VALUE FOR X");
              . . . . .
            }
```

All of the popular general-purpose programming languages contain a `goto` statement, though modern programming practice discourages its use. The `goto` statement was used extensively, however, in early versions of some older languages, such as Fortran and BASIC. The most common applications were:

1. Branching around statements or groups of statements under certain conditions.
2. Jumping to the end of a loop under certain conditions, thus bypassing the remainder of the loop during the current pass.
3. Jumping completely out of a loop under certain conditions, thus terminating the execution of a loop.

```
/* convert several lines of text to uppercase

   Continue conversion until the first character in a line is an asterisk (*).
   Break out of the program sooner if two successive dollar signs ($$) are detected */

#include <stdio.h>
#include <ctype.h>

#define EOL '\n'

main()

{
    char letter[80];
    int tag, count, linecount = 1;

    while ((letter[0] = getchar()) != '*')   {

          /* read in a line of text */
          for (count = 1; (letter[count] = getchar()) != EOL; ++count)
              ;
          tag = count;

          /* display the line of text */
          for (count = 0; count < tag; ++count)
              putchar(toupper(letter[count]));
          printf("\n\n");
          ++linecount;

          /* test for a break condition */
          for (count=1; count < tag; ++count)
              if (letter[count-1] == '$' && letter[count] == '$')   {
                 printf("BREAK CONDITION DETECTED - TERMINATE EXECUTION\n\n");
                 goto end;
              }
    }
    end: printf("Good bye");
}
```

Thank You