CS1010E Lecture 5
Control Structures and Data Files (Part 2)

Joxan Jaffar

Block COM1, Room 3-11, +65 6516 7346
www.comp.nus.edu.sg/~joxan
cs1010e@comp.nus.edu.sg

Semester II, 2016/2017

Lecture Outline

- Loop structures
- Example: Temperature Conversion.
- Data Files.

Loop Structures

- Loops are used to implement repetitive structures.
- C contains three different loop structures:
  - The `while` loop.
  - The `do/while` loop.
  - The `for` loop.
- C allows two additional statements with loops to modify their performance:
  - The `break` statement (which we used with the `switch` statement.)
  - The `continue` statement.

while Loop

- The general form of a `while` loop is:
  - `while` (condition)
    `{`
        statements;
    `}`
- The condition is evaluated before the statements within the loop are executed.
- If the condition is false, the loop statements are skipped, and execution continues with the statement following the `while` loop.

while Loop

- If the condition is true, then the loop statements are executed, and the condition is evaluated again.
- If it is still true, then the statements are executed again, and the condition is evaluated again.
- This repetition continues until the condition is false.
- The statements within the loop must modify variables that are used in the condition; otherwise, the value of the condition will never change, and we will never be able to exit the loop.
- An **infinite loop** is generated if the condition in a `while` loop is always true.

while Loop

- The following pseudocode and program use a `while` loop to generate a conversion table for converting degrees to radians.

- The degree values start at 0°, increment by 10°, and go through 360°:

    - *set degrees to* 0
      while degrees $\leq$ 360
          convert degrees to radians
          print degrees, radians
          add 10 *to degrees*

while Loop

```
/*-------------------------------------------------*/
/*  Program chapter3_1                             */
/*                                                 */
/*  This program prints a degree-to-radian table   */
/*  using a while loop structure.                  */

#include <stdio.h>
#define PI 3.141593

int main(void)
{
   /*  Declare and initialize variables.  */
   int degrees=0;
   double radians;

   /*  Print radians and degrees in a loop.  */
   printf("Degrees to Radians \n");
   while (degrees <= 360)
   {
      radians = degrees*PI/180;
      printf("%6i %9.6f \n",degrees,radians);
      degrees += 10;
   }

   /*  Exit program.  */
   return 0;
}
/*-------------------------------------------------*/
```

while Loop

- The first few lines of output from the program are the following:
  - ```
    Degrees to Radians
          0  0.000000
         10  0.174533
         20  0.349066
         . . .
    ```
    ```

do/while Loop

- The `do/while` loop is similar to the `while` loop except that the condition is tested at the end of the loop instead of at the beginning of the loop.
- Testing the condition at the end of the loop ensures that the `do/while` loop is always executed at least once.
- A `while` loop may not be executed at all if the condition is initially false.

do/while Loop

- The general form of the `do/while` loop is:
  - `do`
    `{`
    statements;
    `}` `while` (condition);

- The following pseudocode and program print the degree-to-radian conversion table using a `do/while` loop instead of a `while` loop:
  - *set degrees to* 0
    do
    convert degrees to radians
    print degrees, radians
    add 10 *to degrees*
    while degrees $\leq$ 360

do/while Loop

```
/*-------------------------------------------------*/
/*  Program chapter3_2                             */
/*                                                 */
/*  This program prints a degree-to-radian table   */
/*  using a do-while loop structure.               */

#include <stdio.h>
#define PI 3.141593

int main(void)
{
   /*  Declare and initialize variables.  */
   int degrees=0;
   double radians;

   /*  Print radians and degrees in a loop.  */
   printf("Degrees to Radians \n");
   do
   {
      radians = degrees*PI/180;
      printf("%6i %9.6f \n",degrees,radians);
      degrees += 10;
   } while (degrees <= 360);

   /*  Exit program.  */
   return 0;
}
/*-------------------------------------------------*/
```

for Loop

- Many programs require loops that are based on the value of a variable that increments (or decrements) by the same amount each time through a loop.
- When the variable reaches a specified value, we will want to exit the loop.
- This type of loop can be implemented as a `while` loop, but it can also be easily implemented with the **for loop**.

for Loop

- The general form of the `for` loop is:
  - `for` (expression_1; expression_2; expression_3)
    {
        statements;
    }

- expression_1 is used to initialize the **loop-control variable**.

- expression_2 specifies the condition that should be true to continue the loop repetition.

- expression_3 specifies the modification to the loop-control variable.

for Loop

- Example: to execute a loop 10 times, with the value of `k` going from 1 to 10 in increments of 1:
  - `for (k=1; k<=10; k++) {`
    statements;
    `}`
- Example: to execute a loop with the value of the variable `n` going from 20 to 0 in increments of -2: the this loop structure:
  - `for (n=20; n>=0; n-=2) {`
    statements;
    `}`
- The `for` loop can also be written like this:
  - `for (n=20; n>=0; n=n-2) {`
    statements;
    `}`

for Loop

- The following is the number of times that a `for` loop will be executed:

$$\text{floor}\left(\frac{\text{final value} - \text{initial value}}{\text{increment}}\right) + 1.$$

- If this value is negative, the loop is not executed.

for Loop

- If a `for` loop has the structure:
  - ```
    for (k=5; k<=83; k+=4) {
        statements;
    }
    ```

- It would be executed the following number of times:

$$\text{floor}\left(\frac{83-5}{4}\right)+1=\text{floor}\left(\frac{78}{4}\right)+1=20.$$

- The value of `k` would be 5, then 9, then 13, and so on, until the final value of 81.

- The loop would not be executed with the value of 85, because the loop condition is not true when `k` is equal to 85.
    ```

Nested for Loops

- Consider the following set of nested `for` statements:
  - ```
    for (k=1; k<=3; k++)
        for (j=0; j<=1; j++)
            count++;
    ```
- The outer `for` loop will be executed 3 times.
- The inner `for` loop will be executed twice each time the outer `for` loop is executed.
- Thus, the variable `count` will be incremented 6 times.
    ```

for Loop

- The following pseudocode and program print the degree-to-radian conversion table shown earlier with a `while` loop, now modified to use a `for` loop (note that the pseudocode for the `while` loop solution is identical to the pseudocode for the `for` loop solution):

  - *set degrees to* 0
    while degrees $\leq$ 360
        convert degrees to radians
        print degrees, radians
        add 10 *to degrees*

for Loop

```
/*-------------------------------------------------*/
/*  Program chapter3_3                             */
/*                                                 */
/*  This program prints a degree-to-radian table   */
/*  using a for loop structure.                    */

#include <stdio.h>
#define PI 3.141593

int main(void)
{
   /*  Declare variables.  */
   int degrees;
   double radians;

   /*  Print radians and degrees in a loop.  */
   printf("Degrees to Radians \n");
   for (degrees=0; degrees<=360; degrees+=10)
   {
      radians = degrees*PI/180;
      printf("%6i %9.6f \n",degrees,radians);
   }

   /*  Exit program.  */
   return 0;
}
/*-------------------------------------------------*/
```

for Loop

- The initialization and modification expressions in a `for` loop can contain more than one statement:
  - ```
    for (k=1, j=5; k<=10; k++, j++)
    {
        sum_1 += k;
        sum_2 += j;
    }
    ```

- When more than one statement is used, they are separated by commas and are executed from left to right. This comma operator is executed last in operator precedence.
    ```

break and continue Statements

- We used the `break` statement previously with the `switch` statement.
- The `break` statement can also be used with any of the loop structures to immediately exit from the loop in which it is contained.
- The `continue` statement is used to skip the remaining statements in the current pass, or **iteration**, of the loop and then continue with the next iteration of the loop.
- The `break` and `continue` statements should be used sparingly.

break and continue Statements

- In a `while` loop or a `do/while` loop, the condition is evaluated after the `continue` statement is executed to determine if the statements in the loop are to be executed again.

- In a `for` loop, the loop-control variable is modified, and then the repetition-continuation condition is evaluated to determine whether the statements in the loop are to be executed again.

- Both the `break` and `continue` statements are useful in exiting either the current iteration or the entire loop when error conditions are encountered.

break Statement

- To illustrate the difference between the `break` and `continue` statements, consider the following loop that reads values from the keyboard:

```
sum = 0;
for (k=1; k<=20; k++)
{
   scanf("%lf",&x);
   if (x > 10.0)
      break;
   sum += x;
}
printf("Sum = %f \n",sum);
```

break Statement

- This loop reads up to 20 values from the keyboard.
- If all 20 values are less than or equal to 10.0, then the statements compute the sum of the values and print the sum.
- But if a value is greater than 10.0, then the `break` statement causes control to break out of the loop and execute the `printf` statement.
- Thus, the sum printed is only the sum of the values lower than 10.0 and excludes the value greater than 10.0.

continue Statement

- Now consider a variation of the previous loop:

```
sum = 0;
for (k=1; k<=20; k++)
{
   scanf("%lf",&x);
   if (x > 10.0)
      continue;
   sum += x;
}
printf("Sum = %f \n",sum);
```

continue Statement

- In this loop, the sum of all 20 values is printed if all the values are less than or equal to 10.0.

- However, if a value is greater than 10.0, then the `continue` statement causes control to skip the rest of the statements in that iteration of the loop, and it will continue with the next iteration of the loop.

- Hence, the sum printed is the sum of all values in the 20 values that are less than or equal to 10.0.

Temperature Conversion Table

- Write a program that displays a temperature conversion table between degrees Fahrenheit and degrees Celsius.
- Allow the user to enter the starting temperature, ending temperature, and interval temperature.
- Display all temperatures with three decimal places.
- Show also the number of resulting Celsius values that are negative.
- Recall that $T_C = (T_F - 32^\circ\mathrm{F}) \times \frac{5}{9}$.

ftoc1.c

```
/*-------------------------------------------------------*/
/*  Program ftoc1                                        */
/*                                                       */
/*  This program generates a table of temperatures       */
/*  in degrees Fahrenheit and degrees Celsius.           */

#include <stdio.h>

int main(void)
{
```

Code on Next Slide

```
    /*  Exit program.  */
    return 0;
}
```

ftoc1.c (body only)

```
printf("Enter starting temperature in degrees F: ");
scanf("%lf", &start_f);
printf("Enter ending temperature in degrees F: ");
scanf("%lf", &end_f);
printf("Enter temperature interval in degrees F: ");
scanf("%lf", &interval_f);
printf("\n");
/*  Print the table header.  */
printf("Fahrenheit    Celsius\n");
/*  Loop through each Fahrenheit temperature.  */
for (f = start_f; f <= end_f; f += interval_f) {
    /*  Convert Fahrenheit to Celsius.  */
    c = (f - 32)*5/9;
    /*  See if the Celsius temperature is negative.  */
    if (c < 0)  num_negative++;
    /*  Print out the table value.  */
    printf("%10.3f %10.3f\n", f, c);
}  /*  End of for-loop.  */

/*  Print out the number of negative Celsius temperatures.  */
printf("\n");
printf("Number of negative Celsius temperatures: %i\n\n",
       num_negative);
```

Data Files

- Engineering problem solutions often involve large amounts of data.
- These data can be output data generated by the program, or input data that are used by the program.
- It is not generally feasible to either print large amounts of data to the screen or to read large amounts of data from the keyboard.
- In these cases, we usually use data files to store the data.

I/O Statements

- Each data file used in a program must have a **file pointer** associated with it.

- If a program uses two files, then each file requires a different file pointer.

- A file pointer is defined with a `FILE` declaration, as in:
  - `FILE *sensor;`

  (Pointers described later in lectures.)

- The `FILE` data type is defined in the header file `stdio.h`, and thus the word `FILE` is capitalized to match the definition in the header file.

Opening a File

- The `fopen` function obtains the information needed to assign a file pointer to a specific file.

- The two arguments for this function are the file name and a character that indicates the file status, which is also called the **file open mode**. Both the file name and the character need to be enclosed in double quotes.

- If we are going to read information from a file with a program, the file open mode is `r` for read.

- If we are going to write information to a file with a program, the file open mode is `w` for write.

Opening a File

- The following statement specifies that the file pointer `sensor` is going to be used with a file named `sensor1.txt`, from which we will read information:
  - `sensor = fopen("sensor1.txt","r");`
- If the data file cannot be opened, `fopen` returns a value of `NULL`.
- `NULL` is a symbolic constant defined in `stdio.h` and has the value of a character zero.

Errors

- A common reason that a program might not be able to open a data file is because the file cannot be found by the program.

- To be sure that our programs find their data files, it is a good practice to check the value returned by `fopen` to ensure that the file was successfully opened.

  - ```
    file1 = fopen(FILENAME,"r");
    if (file1 == NULL)
       printf("Error opening input file \n");
    else
    {
       ...
    }
    ```

- If the file is not successfully found, an error message is printed and the rest of the `if` statement is skipped:
    ```

Reading

- If each line in the `sensor1.txt` file contains a time and sensor reading, we can read one line of this information and store the values in the variables `t` and `motion` with this statement:
  - `fscanf(sensor,"%lf %lf",&t,&motion);`

- Note that the difference between the `scanf` function and the `fscanf` function is that the first argument in the `fscanf` function is the file pointer.

- Otherwise, both statements are the same.

Writing

- If we want to modify this program so that it generates a data file containing this set of data, we could use a pointer `temps` that would be associated with an output file named `temps1.txt` using these statements:
  - ```
    FILE *temps;
    ...
    temps = fopen("temps1.txt", "w");
    ```
- Then, as we compute this information, we can write it to the file with this statement:
  - ```
    fprintf(temps, "%10.3f %10.3f\n",
            f, c);
    ```
- Note again that first argument to `fprintf` is a file pointer.

Closing a File

- The `fclose` function is used to close a file after we are finished with it. The function argument is the file pointer.

- To close the two files used in these sample statements, we use the following:
  - ```
    fclose(sensor);
    fclose(temps);
    ```

- There is no distinction between closing an input file and closing an output file.

- If a file has not been closed when the `return` statement is executed, it will automatically be closed.
    ```

Preprocessor Directives

- A preprocessor directive is often used to specify the data file name because we frequently use the same program with different data files.

- It is easier to modify the preprocessor directives than it is to search through the statements for the `fopen` function.

- An example:
  - ```
    #define FILENAME "sensor1.txt"
    ...
    sensor = fopen(FILENAME,"r");
    ```
    ```

Reading Data Files

We will cover three ways of reading a file:

- Using a (manually) specified number of records
- Using a sentinel
- Using the return value from the `fscanf` function

Our example is about reading seismometer sensor information containing a number of records. Each record is a pair of numbers, the first is the time, and second the motion. We will read all the records, then output the minimum, maximum and average values of motion.

Specified Number of Records

- Assume that the first record in the sensor data file contains an integer that specifies the number of records of sensor information that follow.
- Each of the following lines contains a time and sensor reading in a file named `sensor1.txt`.
- The contents of this data file is shown in the next slide.

Specified Number of Records

```
10
0.0 132.5
0.1 147.2
0.2 148.3
0.3 157.3
0.4 163.2
0.5 158.2
0.6 169.3
0.7 148.2
0.8 137.6
0.9 135.9
```

Refinement in Pseudocode

```
set sum to 0
if (file cannot be opened) print error message
else {
   read number of data points
   set k to 1
   while (k ≤ number of data point) {
      read time, motion
      if (k == 1) { set max and min to motion }
      add motion to sum
      if (motion > max) set max to motion
      if (motion < min) set min to motion
      increment k by 1
   }  set average to sum/number of data points
   print average, max, min
}
```

C Program Code

```
/*-------------------------------------------------*/
/*  Program chapter3_5                             */
/*                                                 */
/*  This program generates a summary report from   */
/*  a data file that has the number of data points */
/*  in the first record.                           */

#include <stdio.h>
#define FILENAME "sensor1.txt"

int main(void)
{
   /*  Declare and initialize variables.  */
   int num_data_pts, k;
   double time, motion, sum=0, max, min;
   FILE *sensor;

   Code on Next Slide

   /*  Exit program.  */
   return 0;
}
```

Body code for Program chapter3_5

```
sensor = fopen(FILENAME,"r");
if (sensor == NULL) printf("Error opening input file.\n");
else {
   fscanf(sensor,"%d",&num_data_pts);
   /*  Read data and compute summary information. */
   for (k=1; k<=num_data_pts; k++) {
      fscanf(sensor,"%lf %lf",&time,&motion);

      /*  Initialize variables using first data point */
       if (k == 1) max = min = motion;

      /*  Update summary data.  */
      sum += motion;
      if (motion > max) max = motion;
      if (motion < min) min = motion;
   }
}
```

Executing

- The following report will be printed by this program using the `sensor1.txt` file:
  - ```
    Number of sensor readings: 10
    Average reading:           149.77
    Maximum reading:           169.30
    Minimum reading:           132.50
    ```
- The next two programs also print the above identical report.
    ```

Trailer or Sentinel Signals

- Assume that the data file `sensor2.txt` contains the same information as the `sensor1.txt` file.
- Instead of giving the number of valid data records at the beginning of the file, a final record contains a trailer signal.
- The time value on the last line in the file will contain a negative value so that we know that it is not a valid line of information.
- Note that we still will have two numbers in this last trailer line

Trailer or Sentinel Signals

```
0.0 132.5
0.1 147.2
0.2 148.3
0.3 157.3
0.4 163.2
0.5 158.2
0.6 169.3
0.7 148.2
0.8 137.6
0.9 135.9
-99 -99
```

Refinement in Pseudocode

```
set sum and number of points to 0
if (file cannot be opened) print error message
else {
   read time, motion
   set max and min to motion
   do {
      add motion to sum
      if (motion > max) set max to motion
      if (motion < min) set min to motion
      increment number of points by 1
      read time, motion
   } while (time ≥ 0)
   set average to sum/number of data points
   print average, max, min
}
```

C Program Code

```
/*-------------------------------------------------------*/
/* Program chapter3_6                                    */
/*                                                       */
/* This program generates a summary report from a data */
/* file that has a trailer record with negative values */

#include <stdio.h>
#define FILENAME "sensor2.txt"

int main(void)
{
   /*  Declare and initialize variables.  */
   int num_data_pts = 0;
   double time, motion, sum=0, max, min;
   FILE *sensor;

   Code on Next Slide

   /*  Exit program.  */
   return 0;
}
```

Body code for Program chapter3_6

```
sensor = fopen(FILENAME,"r");
if (sensor == NULL) printf("Error opening input file.\n");
else {
   fscanf(sensor,"%lf %lf",&time,&motion);
   /* Initialize variables using first data point */
   max = min = motion;
   /*  Update summary data until trailer  */
   /*  record read.                       */
   do {
      sum += motion;
      if (motion > max) max = motion;
      if (motion < min) min = motion;
      num_data_pts++;
      fscanf(sensor,"%lf %lf",&time,&motion);
   } while (time >= 0);
   /*  Print summary information.  */
   printf("Num of sensor readings: %d \n", num_data_pts);
   printf("Average reading:        %.2f \n", sum/num_data_pts);
   printf("Maximum reading:        %.2f \n",max);
   printf("Minimum reading:        %.2f \n",min);
   /*  Close file.  */
   fclose(sensor);
}
```

End-of-File

- A special **end-of-file indicator** is inserted at the end of every data file.
- The `feof` function in the Standard C library can be used to detect when this indicator has been reached in a data file.
- The `fscanf` function can also be used to detect when the end of the data has been reached in a file.
- Recall that the `fscanf` function returns the number of values successfully read each time that it is executed.

End-of-File

- If the function returns a value that is different from the number of values that it was supposed to read, then the end of the data file has been reached, or there are errors in the information in the data file.

- If the information in the data file is valid, then the `fscanf` function can be used to determine when the end of the file is reached.

End-of-File

- Consider the following statements:
  - ```
    while ((fscanf(data,"%lf",&x)) == 1)
    {
        count++;
        sum += x;
    }
    ave = sum/count;
    ```

- The `fscanf` function attempts to read a value for `x` from a data file.

- If a value is read, the function returns a value of 1, and the statements within the loop are executed.
    ```

End-of-File

- If the end of the data file is reached, there is no more data; thus, the function does not return a value of 1, and control passes to the statement following the `while` loop.

- We assume that the data file `sensor3.txt` contains the same information as the `sensor2.txt` file, except that it does not include the trailer signal.

- The contents of `sensor3.txt` are shown in the next slide.

End-of-File

```
0.0 132.5
0.1 147.2
0.2 148.3
0.3 157.3
0.4 163.2
0.5 158.2
0.6 169.3
0.7 148.2
0.8 137.6
0.9 135.9
```

Refinement in Pseudocode

```
set sum and number of points to 0
if (file cannot be opened) print error message
else {
   read time, motion
   set max and min to motion
   while (not at the end of the file) {
      read time, motion
      increment number of points by 1
      if (k == 1) set min and max to motion
      add motion to sum
      if (motion > max) set max to motion
      if (motion < min) set min to motion
   }
   set average to sum/number of data points
   print average, max, min
}
```

C Program Code

```
/*-----------------------------------------------------*/
/* Program chapter3_7                                  */
/*                                                     */
/* This program generates a summary report from a data */
/* file that does not have a header not trailer record */

#include <stdio.h>
#define FILENAME "sensor2.txt"

int main(void)
{
   /*  Declare and initialize variables.  */
   int num_data_pts = 0;
   double time, motion, sum=0, max, min;
   FILE *sensor;

   Code on Next Slide

   /*  Exit program.  */
   return 0;
}
```

Body code for Program chapter3_7

```
sensor = fopen(FILENAME,"r");
if (sensor == NULL) printf("Error opening input file.\n");
else {
   /* While not at the end of the file, */
   /* read and accumulate information.  */
   while ((fscanf(sensor,"%lf %lf",&time,&motion)) == 2) {
      num_data_pts++;
      /* Initialize variables using first data point */
      if (num_data_pts == 1) max = min = motion;
      /* Update summary data.  */
      sum += motion;
      if (motion > max) max = motion;
      if (motion < min) min = motion;
   }
   /*  Print summary information.  */
   printf("Num of sensor readings: %d \n", num_data_pts);
   printf("Average reading:        %.2f \n", sum/num_data_pts);
   printf("Maximum reading:        %.2f \n",max);
   printf("Minimum reading:        %.2f \n",min);
   /*  Close file.  */
   fclose(sensor);
}
```

Generating a Data File

- Generating a data file is similar to printing a report.
- Instead of writing the line to the terminal screen, we write it to a data file.
- Before we generate the data file, we must decide which of the three file structures to use:
  - Specified number of records.
  - Trailer or sentinel signals.
  - End-of-file.

Generating a Data File

- There are advantages and disadvantages to each of the three file structures.

- If the initial record in the data file contains the number of lines of actual data, we must know how many lines of data will be in the file before we generate the file. It may not always be easy to determine this number.

- A file with a trailer signal is simple to use, but choosing a value for the trailer signal must be done carefully so that it does not contain values that could occur in the valid data.

Generating a Data File

- The simplest file to generate is the one that contains only the valid information, with no special information at the beginning or end of the file.

- If the information in the file is going to be used with a plotting package, it is usually best to use this third file structure, which includes only valid information.

Generating a Data File

- We will now modify program `ftoc1` stored in `ftoc1.c`.
- The program printed a temperature conversion table onto the screen.
- In our modified program, we will write the temperature conversion table into a data file.
- Compare the modified program with the original program.

C Program Code

```
/*-------------------------------------------------*/
/*  Program ftoc2                                  */
/*                                                 */
/*  This program generates a table of temperatures */
/*  in degrees Fahrenheit and degrees Celsius.     */
/*  Stores the table into an output file.          */

#include <stdio.h>
#define FILENAME "temps1.txt"

int main(void)
{
    /*  Declare and initialize variables.  */
    double start_f, end_f, interval_f, f, c;
    int num_negative = 0;
    FILE *temps;
```

Code on Next Slide

```
    /*  Exit program.  */
    return 0;
}
```

Body code for ftoc2.c

```
/* Prompt for starting, ending, and interval temperatures. */
printf("\nEnter starting temperature in degrees F: ");
scanf("%lf", &start_f);
printf("Enter ending temperature in degrees F: ");
scanf("%lf", &end_f);
printf("Enter temperature interval in degrees F: ");
scanf("%lf", &interval_f);
printf("\n");
/* Open output file.  */
temps = fopen(FILENAME, "w");
/* Loop through each Fahrenheit temperature.  */
for (f = start_f; f <= end_f; f += interval_f)  {
    /*  Convert Fahrenheit to Celsius.  */
    c = (f - 32)*5/9;
    /* See if the Celsius temperature is negative.  */
    if (c < 0) num_negative++;
    /* Print out the table value into the output file.  */
    fprintf(temps, "%10.3f %10.3f\n", f, c);
}  /* End of for-loop. */
/*  Close file.  */
fclose(temps);
/*  Print out the number of negative Celsius temperatures.  */
printf("Num of negative Celsius temps: %i\n\n", num_negative);
```

Generating a Data File

- The first few lines of the data file `temps1.txt` generated by this program using some input values are shown below:
  - ```
    20.000     -6.667
    24.000     -4.444
    28.000     -2.222
    32.000      0.000
    36.000      2.222
    40.000      4.444
    ```

- This file is in a form that can be easily plotted using a package such at MATLAB.
    ```

References

Etter Sections 3.1 to 3.4
Etter Sections 3.6

Next Lecture

Modular Programming with Functions

(Part 1)