Reading from the command line

Oftentimes, it is very useful to read information from the command line. What this means is: When you first run your executable file, you can also provide it some information at the same time. For example, you typically run your executable file by simply typing:

```
a.out
```

However, we can also provide the program some numbers or strings of text. For example, we could execute our program as follows:

```
a.out hello 3 4
```

a.out will execute the program, but it will take the following strings of text with it: "hello", "3", and "4". **In order for this to work, your program must be written in such a way to allow this.** The following program does this; it reads information from the command line and prints it out. It is important to note that this method reads in the information on the command line as **text**. If you want to turn that text into real number or integer variables, you must use the **sscanf** function that was covered in a previous section.

The main difference between the following code and our previous codes is that the function **main** includes 2 arguments: **number_of_strings** and **string_array**. You could name these variables anything you would like, but the important thing is that the first one is an integer and the second is a special type of char variable. Immediately before the brackets in **main**, these variables are declared. **number_of_strings** is an integer, and **string_array** is a "pointer to a pointer char variable". We have not dealt with pointer to pointer char variables yet, so just memorize this template for now. Essentially, this creates an array of strings (i.e., text). The integer **number_of_strings** tells how many command line arguments there are. This number includes the **a.out** itself. Each piece of text is stored in an array position within **string_array**. As we'll see in the next section, arrays in C start at position 0. Therefore, **string_array** will contain **number_of_string** positions (each with text in them), starting at position **string_array[0]** and ending at **string_array[*number_of_*strings**-1]. As usual, this is all best learned by example:

code27.c

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

int main(number_of_strings, string_array)
  int number_of_strings;
  char **string_array;
{

int i;

   // Integer variable argc tells how many
   // strings of test were included on the command line.

   fprintf(stdout,"There are %d strings on text on the command line\n",
              number_of_strings);

   // Print each string of text by looping through the array of strings

   for (i=0; i <= (number_of_strings-1); i++)
   {
       fprintf(stdout,"%s\n",string_array[i]);
   }

return 0;
}
```

Now, run the code by simply typing

a.out

This is the output you should get:

```
There are 1 strings on text on the command line
a.out
```

You code determined that there was 1 string of text on the command line, which was simply the command **a.out** itself. Now lets try something a little more interesting:

a.out Newton Laplace Euler Fourier Joule Kepler
You should now get the following output:

```
There are 7 strings on text on the command line
a.out
Newton
Laplace
Euler
Fourier
Joule
Kepler
```

Arrays

The previous exercise was a very brief exercise in arrays. Arrays are very important in scientific programming because they allow us to store information in dimensional form which can lend itself very conveniently toward repetitive calculation. Furthermore, matrix or tensor calculations could become quite tedious without using arrays.

Any data type can be expanded into an array. For example, you could have an array of integers, doubles, or even text. As mentioned before, an array in C starts at position 0. This is in contrast to the Fortran programming language in which arrays start at position 1. Another important thing to point out is that once you create an array, the values of its components are **NOT** necessarily equal to 0. **Never** assume that an array is created with values of zero already initialized in its positions. If you want the array to have all zeros upon its creation, you must do so yourself.

There are a couple of ways to use arrays in C. The first way (that we'll cover first) is the simplest, yet it is not the most convenient for larger, more-sophisticated programs. This first way is to simply define a variable as an array. The main disadvantages of this simpler method are that (1) the array is static (its size cannot be modified during program execution), and (2) its maximum size is limited (this is determined by the specific compiler). If you try to declare an array that is too large for the compiler, it will issue an error upon compiling. The second way, which is less simple, but more flexible, is to create a sequence of memory pointers. This more complicated method allows for very large dynamic arrays that can be modified throughout the execution of the program. This allows one to conserve memory much more efficiently. Let's focus on the simple way first.

The following code declares and uses arrays:

code28.c

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

int main(void)
{

// Let's declare an array of integers that
// contains 20 integer positions.

int my_integers[20];

// Let's also declare an array of doubles
// that contains 10 double positions.

double my_real_numbers[10];

// Let's loop through each array position to
```

```
// initialize the values to zero.

int i;

   // loop through the integer array first
   for (i=0; i <= 19; i++)
   {
     my_integers[i] = 0;
   }

   // now loop through the double array
   for (i=0; i <= 9; i++)
   {
     my_real_numbers[i] = 0.0;
   }

   // Lets provide some numbers to the
   // array positions.

    my_integers[3] = -100;
    my_integers[18] = 99;
    my_integers[8] = 4;


    my_real_numbers[0] = 1.0;
    my_real_numbers[1] = 3.0;
    my_real_numbers[8] = 7.0;

    // Lets loop through each array
    // and print out the contents

   // loop through the integer array
   for (i=0; i <= 19; i++)
   {
     fprintf(stdout,"Integer array:  Position: %d Value: %d\n",
           i,my_integers[i]);
   }

   // loop through the double array
   for (i=0; i <= 9; i++)
   {
     fprintf(stdout,"Double array:  Position: %d Value: %f\n",
           i,my_real_numbers[i]);
   }

return 0;
}
```

Once you run the code, you should get the following output:

```
Integer array:  Position: 0 Value: 0
Integer array:  Position: 1 Value: 0
Integer array:  Position: 2 Value: 0
Integer array:  Position: 3 Value: -100
Integer array:  Position: 4 Value: 0
Integer array:  Position: 5 Value: 0
Integer array:  Position: 6 Value: 0
Integer array:  Position: 7 Value: 0
Integer array:  Position: 8 Value: 4
Integer array:  Position: 9 Value: 0
Integer array:  Position: 10 Value: 0
Integer array:  Position: 11 Value: 0
Integer array:  Position: 12 Value: 0
Integer array:  Position: 13 Value: 0
Integer array:  Position: 14 Value: 0
Integer array:  Position: 15 Value: 0
Integer array:  Position: 16 Value: 0
Integer array:  Position: 17 Value: 0
Integer array:  Position: 18 Value: 99
Integer array:  Position: 19 Value: 0
Double array:  Position: 0 Value: 1.000000
Double array:  Position: 1 Value: 3.000000
Double array:  Position: 2 Value: 0.000000
Double array:  Position: 3 Value: 0.000000
Double array:  Position: 4 Value: 0.000000
Double array:  Position: 5 Value: 0.000000
Double array:  Position: 6 Value: 0.000000
Double array:  Position: 7 Value: 0.000000
Double array:  Position: 8 Value: 7.000000
Double array:  Position: 9 Value: 0.000000
```

In the above program, you created an integer array, named *my_integers*, which contains 20 positions (from 0 through 19), and you created a double array, named *my_real_numbers*, which contains 10 positions (from 0 through 9). You first initialized them to set every array position to 0. You then assigned a few numbers to some of their positions. Then, you printed everything out. Carefully read the output to make sure you make the connection to what your code instructed.

There are a couple points to make here:

1) The index for an array position must always be an integer. This means that an array position can only be defined by an integer value. For example, to access position 3 of an array, you would use *my_array*[2] and **NEVER** something like *my_array*[2.0]. The latter could cause some **severe problems** with your code. It may actually run, but you couldn't trust the results very well.
2) Be very careful never to "overwrite" an array. For example, if your array is only 10 positions long, only act on array positions 0 though 9. C actually allows the user to overwrite arrays, but unless you intentionally planned this (for reasons that are beyond me) you can expect serious trouble. If you overwrite the array, you

may actually overwrite other variables that are stored in memory. **If your code is acting very wacky when using arrays, the first thing to check is that you didn't overwrite an array somewhere!**

In C, you can have as many dimensions to an array as you would like. The above example used one-dimensional arrays. Matrix calculations typically employ 2D arrays. For example, consider the following matrix calculation

$$\overline{\overline{F}} = \overline{\overline{A}}\,\overline{\overline{B}}$$

which states that matrix F is equal to matrix A times matrix B. If A and B are 2x2 arrays:

$$\overline{\overline{A}} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \qquad \overline{\overline{B}} = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

then the solution is simply:

$$\overline{\overline{F}} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

The following code, performs the following matrix operation to determine F:

$$\overline{\overline{F}} = \overline{\overline{A}}\,\overline{\overline{B}}$$

where

$$\overline{\overline{A}} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad \overline{\overline{B}} = \begin{pmatrix} 1 & -1 \\ -1 & 4 \end{pmatrix}$$

As is standard notation, we will use the integer variable ***i*** to represent the row number and the integer variable ***j*** to represent the column number. **NOTE: to avoid confusion, we will declare these arrays as 3x3 arrays and simply ignore the values in the zero positions.** Also, as is conventional, the first dimension of the array will refer to the row and the second dimension will refer to the column.

code29.c

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

int main(void)
{

// Declare 2 integers to use as array
// indices. "i" will refer to row, and
// "j" will refer to column.

int i, j;

// declare double arrays A, B, and F

double A[3][3];
double B[3][3];
double F[3][3];

// Fill in array A
// Note, we ignore the 0 positions

   A[1][1] = 1.0;
   A[1][2] = 2.0;
   A[2][1] = 3.0;
   A[2][2] = 4.0;

// Fill in array B
// Note, we ignore the 0 positions

   B[1][1] = 1.0;
   B[1][2] = -1.0;
   B[2][1] = -1.0;
   B[2][2] = 4.0;

// To be paranoid, lets fill F with zeros
// to start with

   F[1][1] = 0.0;
   F[1][2] = 0.0;
   F[2][1] = 0.0;
   F[2][2] = 0.0;

// Perform the matrix calculation
// Note that the following 2 loops perform
// the following:
//   F[1][1] = A[1][1]*B[1][1] + A[1][2]*B[2][1];
//   F[1][2] = A[1][1]*B[1][2] + A[1][2]*B[2][2];
//   F[2][1] = A[2][1]*B[1][1] + A[2][2]*B[2][1];
//   F[2][2] = A[2][1]*B[1][2] + A[2][2]*B[2][2];
```

```
  for (i = 1; i<= 2; i++)
  {
  for (j = 1; j<= 2; j++)
  {
     F[i][j] = A[i][1]*B[1][j] + A[i][2]*B[2][j];
  }
  }

// Lets loop through F to print out results

  for (i = 1; i<= 2; i++)
  {
  for (j = 1; j<= 2; j++)
  {
     fprintf(stdout,"F[%d][%d] = %f\n",i,j,F[i][j]);
  }
  }

return 0;
}
```

You should have the following output:

```
F[1][1] = -1.000000
F[1][2] = 7.000000
F[2][1] = -1.000000
F[2][2] = 13.000000
```

You should convince yourself that the matrix multiplication is adequately performed with the following 2 loops working together:

```
  for (i = 1; i<= 2; i++)
  {
  for (j = 1; j<= 2; j++)
  {
     F[i][j] = A[i][1]*B[1][j] + A[i][2]*B[2][j];
  }
  }
```

This next exercise will assist you on a homework exercise. First of all, create a file named *code30.input*. This file will contain values of a 3x3 array that you will read into your code. The first line of the text file will contain 2 integers, the number of rows and the number of columns. The following lines will include values of each array position.

The code will read the input file name from the command line. Before the code declares the array, it will first read how many rows and columns there are (from the input file). Note how the array is declared in this code! We can read in the number of rows and columns, and then declare the array size once we have that information. Very convenient!

Make sure you read over (and fully understand) the following input file and code.

code30.input

```
3    3    # the number of rows,  the number of columns
1.0       #value of position A11
2.0       #value of position A12
3.0       #value of position A13
4.0       #value of position A21
5.0      #value of position A22
6.0      #value of position A23
7.0      #value of position A31
8.0      #value of position A32
9.0      #value of position A33
```

code30.c

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

int main(number_of_strings, string_array)
   int number_of_strings;
   char **string_array;
{

FILE *fpin;
char temp_string[200];
int number_of_rows;
int number_of_columns;
double value;
int i,j;

// Because we will read the name of the input
// file from the command line, lets insure that
// the number_of_strings is greater than 1.  If
// number_of_string equals 1, it means that we
// forgot to include the input file on the command
// line.

  if (number_of_strings <=1)
```

```
  {
    fprintf(stdout,"Error - forgot to include file name\n");
    exit(10);
  }

// Read the name of the input file from the command line

  fprintf(stdout,"The input file is %s\n",string_array[1]);

// Try to open the input file.  Exit if it cannot be opened.

  fpin = fopen(string_array[1], "r");
  if (fpin == NULL)
  {
    fprintf(stdout,"Error - cannot open input file\n");
    exit(10);
  }

// Read the first line of the input file

  fgets(temp_string,200,fpin);

// Parse the first line of the input file
// using sscanf.
// Attribute the first integer to the
// number of rows, and the second integer
// to the number of columns.

  sscanf(temp_string,"%d %d",&number_of_rows,&number_of_columns);

// Lets print out what we know so far

  fprintf(stdout,"There are %d rows\n",number_of_rows);
  fprintf(stdout,"There are %d columns\n",number_of_columns);

// Now lets declare an array of the proper size.
// We will ignore zeros, so we will make the array
// one dimension higher than actually needed (for
// convenience).

  double A[number_of_rows+1][number_of_columns+1];

// The rest of the input file is ordered in a specific
// manner which allows us to easily loop through the
// rows and columns of array A.  Each time through the
// loop, we will read from the input file and assign the
// value to its proper position within the array.


  for (i=1; i<= number_of_rows; i++)
  {
  for (j=1; j<= number_of_columns; j++)
  {
     // read next line of input file.
     fgets(temp_string,200,fpin);
```

```
        // Take the first number of that
        // line and assign it to its proper
        // array position.

        sscanf(temp_string,"%lf",&A[i][j]);

    }
    }

    // Lets print everything out to make sure we
    // read it in correctly.

    for (i=1; i<= number_of_rows; i++)
    {
    for (j=1; j<= number_of_columns; j++)
    {
        fprintf(stdout,"A[%d][%d]=%f\n",i,j,A[i][j]);
    }
    }


  return 0;
  }
```

To run the code, type **a.out code30.input**

You should get the following output:

```
The input file is code30.input
There are 3 rows
There are 3 columns
A[1][1]=1.000000
A[1][2]=2.000000
A[1][3]=3.000000
A[2][1]=4.000000
A[2][2]=5.000000
A[2][3]=6.000000
A[3][1]=7.000000
A[3][2]=8.000000
A[3][3]=9.000000
```

This code is one of the more advanced codes that we have dealt with so far, at least in terms of reading from file. Please make sure you have reversed-engineered the entire thing so that you understand it completely. Otherwise, you may not be very successful in homeworks.