The char Data type

Before we go into the subject of reading from files, it is convenient to describe a new data type called **char**. The **char** data type is simply text. A variable of type **char** is actually an array, with each position of that array consisting of a single character. We often call a variable of type **char** a **string variable.**

To declare a string variable named *my_string* that is at least 200 characters long, you would declare the following:

```
char my_string[200];
```

In the above line, **char** specifies the type of variable that *my_string* is. The **[200]** tells it that *my_string* has a maximum length of 200. The text you want to store in *my_string* doesn't have to be 200 positions long, but it is important to make sure it is less than or equal to 200 positions long. In my experience 200 positions is long enough for most work, so I typically define character variables as this length.

To assign text to a **char** variable, I typically use the **sprintf** function. This function consists of 2 arguments. The first argument is the name of the **char** variable, and the second argument is the text that you want to put into it.

```
sprinf(character_variable, text to put into the character variable);
```

To print a **char** variable, we use the placeholder **%s**

The following code shows how to deal with this:

code22.c

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

int main(void)
{

// Here, we define a char variable named
// "my_cereal".  This variable is 200
// characters long (maximum).
char my_cereal[200];

//Lets put some text into "my_cereal" using
// the sprintf function.

sprintf(my_cereal,"fruity pebbles are awesome!");


// Print the contents of "my_cereal"
fprintf(stdout,"%s\n",my_cereal);
return 0;
}
```

You should get the following output:

```
fruity pebbles are awesome!
```

Reading from Files

There are multiple ways to read files. In this course, I will focus on one way to do this (my way). Reading from a file consists of (1) opening the file, (2) reading lines of text from the file, (3) possibly translating that text to other variables, and (4) closing the file.

Once a file is opened, there is an imaginary arrow pointed at the first line of that file. To read in that line of the file, you use the **fgets** function. The **fgets** function has 3 arguments. The first is the name of a **char** variable that you are dumping that line of file to. The second is the maximum number of columns that you are extracting from that line of file. The third is the name of that file's file pointer.

```
fgets(char variable, max number of columns, file pointer)
```

Each time fgets is called, the imaginary arrow will move down to the next line in the file. If it gets to a line that doesn't exist (i.e., the end of the file), then it will return a **NULL** value. **Note:** The end of a file's content is not necessarily the end of a file because blank lines are considered information too.

Before we practice reading from files, let's make a file to read from. Using vi, create a file named *data.input* that contains the following:

```
Here are some integers
0 1 2 3
Here are some doubles
0.1111 0.2222 0.3333
Here are some strings
crunchy creamy
```

Our first example will focus on just reading these 6 lines of file *data.input.*

code23.c

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

int main(void)
{

//Declare a char variable to send the file content to.
//We'll name this variable "temp_string".
char temp_string[200];

//Declare a file pointer.  Lets name
// it "fpin".
FILE *fpin;

//Open the file and check that it exists
  fpin = fopen("data.input","r");
  if (fpin == NULL)
  {
    fprintf(stdout,"ERROR-file does not exist.\n");
    exit(10);
  }

//Read the lines of the file and print the text
  fgets(temp_string,200,fpin);
  fprintf(stdout,"%s",temp_string);

  fgets(temp_string,200,fpin);
  fprintf(stdout,"%s",temp_string);

  fgets(temp_string,200,fpin);
  fprintf(stdout,"%s",temp_string);

  fgets(temp_string,200,fpin);
  fprintf(stdout,"%s",temp_string);

  fgets(temp_string,200,fpin);
  fprintf(stdout,"%s",temp_string);

  fgets(temp_string,200,fpin);
  fprintf(stdout,"%s",temp_string);

return 0;
}
```

You should get the following output:

```
Here are some integers
0 1 2 3
Here are some doubles
0.1111 0.2222 0.3333
Here are some strings
crunchy creamy
```

Note that each time **fgets** is called, we will move one more line down the file. When **fgets** cannot go any further because the end of file is reached, it will send back a **NULL** value to the code. We can use this property to have a code read lines of a file until it has reached the end. We will do this in the following example.

Next, modify the code to put the **fgets** in a loop that will end when the end of file has reached. We'll use **while** to do this.

code23b.c

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

int main(void)
{

//Declare a char variable to send the file content to.
//We'll name this variable "temp_string".
char temp_string[200];

//Declare a file pointer.  Lets name
// it "fpin".
FILE *fpin;

//Declare an integer named "keep_going".
//This integer will be used to tell the while
//loop when to stop.
int keep_going;

//Open the file and check that it exists
  fpin = fopen("data.input","r");
  if (fpin == NULL)
  {
    fprintf(stdout,"ERROR-file does not exist.\n");
    exit(10);
  }

//Create a while loop that will continue to
//read from file and print the info to the screen
//until it reaches the end of the file.

   keep_going=1;
   while (keep_going)
   {
      if (fgets(temp_string,200,fpin)==NULL)
      {
        //We have reached the end of
        //file so set keep_going to 0
        // to stop the loop.

        keep_going = 0;
      }
      else
      {
        fprintf(stdout,"%s",temp_string);
      }
   }

return 0;
}
```

The output should be the same as before. Notice that the **fgets** is called, even as it is included within the IF statement.

Also note that in the above examples I was sloppy because I did not include an **fclose** function to close the file. This is ok because the file automatically closes when I exit the code. For small codes this is ok, but for longer, more advanced codes, it is often beneficial to close files before the code has finished.

The next step is to translate the information that we read from the file to actual variables. Currently, we are just storing the file content in a **char** variable named *temp_string*. Oftentimes, you will want to dissect the **char** variable into different data types such as ints or doubles. You can do this using **sscanf** function. This function will take a **char** variable and parse its contents into variables (of various data types) of your choosing. As with most things in C, the more flexibility you have, the more room for error! Many compiling errors can be attributed to an incorrect usage of the **sscanf** function.

```
sscanf(a char variable, “ placeholders “, memory addresses to variables);
```

Above, we see our first reference to “memory address”. We will see later, when we deal with pointers, that directly controlling memory is a central theme to C programming. We don’t need to deal with that too much now, but keep in mind that when using sscanf, we must enter the memory addresses of a variable, not simply the variable name. Ok, I’m sure this doesn’t make any sense, so let’s look at the following example.

Say we have a **char** variable named “my_string” that contains the following content:

```
1.2 1.2  1.3 1    2
```

Remember, because this is a **char** variable, this is just text, not really numbers as far the program is concerned. Let’s say that we want to turn this content into 3 double variables (the first 3 numbers) and 2 integer variables (the last 2 numbers). Let’s name these variables the following:

```
double number1, number2, number3;
int  integer1, integer2;
```

Here is how we would turn the text in “my string” to the double and integer variables:

```
sscanf(my_string,   “  %lf  %lf  %lf   %d  %d”, &number1, &number2, number3,
&integer1,&integer2)
```

You should notice 2 new things here! First of all, note that we use the **&** symbol before the variable name to indicate the **memory address**. We’ll get into this more later, but for now, just remember to put **&** before the variable names in **sscanf**. Secondly, notice that the placeholders for our doubles are written as **%lf**. If we had simply used **%f**, we would likely have huge problems in the code. This could actually cause the worst type of problem that is the trickiest to find, called a memory problem. When we are **printing** to the screen or file, if is fine to use **%f** as a placeholder for either a float or a double;

however, when **reading**, it is critical to use **%lf** for doubles and **%f** for floats. If you mess this up, you'll be in big trouble. **%d** is the placeholder for integers, for both reading and printing.

Let's learn by example by creating a modified version of *code23.c*.

Code24.c

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

int main(void)
{

//Declare a char variable to send the file content to.
//We'll name this variable "temp_string".
char temp_string[200];

//Declare a file pointer.  Lets name
// it "fpin".
FILE *fpin;

//Declare some variables to store file data to.
int ii,jj,kk,mm;
double A,B,C;

//Open the file and check that it exists
  fpin = fopen("data.input","r");
  if (fpin == NULL)
  {
    fprintf(stdout,"ERROR-file does not exist.\n");
    exit(10);
  }

//Line 1 of file
  fgets(temp_string,200,fpin);

//Line 2 of file
  fgets(temp_string,200,fpin);

  //We were expecting some integers here.
  //Lets assign them to ii,jj,kk,mm using sscanf

  sscanf(temp_string,"%d %d %d %d",&ii,&jj,&kk,&mm);

  //Lets print out these integers
  fprintf(stdout,"The integers are: %d %d %d %d\n",ii,jj,kk,mm);

//Line 3 of file
  fgets(temp_string,200,fpin);

//Line 4 of file
```

```
  fgets(temp_string,200,fpin);

  //We were expecting some doubles here.
  //Lets assign them to A,B,C using sscanf

  sscanf(temp_string,"%lf %lf %lf",&A,&B,&C);

//Lets print out these doubles
  fprintf(stdout,"The doubles are: %f %f %f\n",A,B,C);

return 0;
}
```

You should get the following output:

```
The integers are: 0 1 2 3
The doubles are: 0.111100 0.222200 0.333300
```

Let’s do another example. In this example, we will have a file that includes a bunch of real numbers. The code will add these numbers up. There will be a number on each line. The file will also have a header that tells the code how many numbers to add up. Using vi, make a file named *my_number.data* that includes the following:

```
5  this is the number of numbers to be added up
3.1
-2.3
1.1
-3.5
9.2
```

The following code will add these numbers up.

code25.c

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

int main(void)
{

//Declare a char variable to send the file content to.
//We'll name this variable "temp_string".
char temp_string[200];

//Declare a file pointer.  Lets name
// it "fpin".
FILE *fpin;

//Declare some variables.
int number_of_numbers;
int n;
double sum, current_number;

//Open the file and check that it exists
  fpin = fopen("my_number.data","r");
  if (fpin == NULL)
  {
    fprintf(stdout,"ERROR-file does not exist.\n");
    exit(10);
  }

// Line 1 of the file is a header
// that tells how many numbers to add.

  // read line 1 into "temp_string"
  fgets(temp_string,200,fpin);

  // extract the integer number from
  // the header.  Store this as "number_of_numbers".

  sscanf(temp_string,"%d",&number_of_numbers);

  //Print how many numbers we're adding
  fprintf(stdout,"We are adding %d numbers\n",number_of_numbers);

  // Loop through all the numbers. Keep adding them
  // to "sum".  We must first set "sum" to zero though.

  sum = 0.0;
  for (n=1; n<=number_of_numbers; n++)
  {
     //read the next line of the file
     fgets(temp_string, 200, fpin);

     //parse that number out of "temp_string"
     sscanf(temp_string,"%lf", &current_number);

     //print out the current number
     fprintf(stdout,"number %d is: %f\n",n,current_number);

     // add that to the sum;
     sum = sum + current_number;
  }

  //print the sum
  fprintf(stdout,"The sum of all numbers is: %f\n",sum);

return 0;
}
```

You should get the following output:

```
We are adding 5 numbers
number 1 is: 3.100000
number 2 is: -2.300000
number 3 is: 1.100000
number 4 is: -3.500000
number 5 is: 9.200000
The sum of all numbers is: 7.600000
```

It is worthwhile here to consider using **sscanf** to parse a **char** variable into several different **char** variables. Remember that a **char** variable is actually an array. In C, if you just give the name of an array without specifying the position, then you are actually providing the **memory pointer** to that array. This is confusing at first, but will hopefully become more clear as we deal with **pointers** more later. For example, if we have the following **char** variable:

```
char  some_string[200];
```

then its **memory address** is simply

```
some_string
```

No **&** is required in front of the variable name.

Lets modify *code25.c* to parse the last line of the file *data.input*. No need to rewrite *code25.c*; you can simply add to it. Modified Lines are shown in red.

code25b.c

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

int main(void)
{

//Declare a char variable to send the file content to.
//We'll name this variable "temp_string".
char temp_string[200];

//Declare a file pointer.  Lets name
// it "fpin".
FILE *fpin;

//Declare some variables to store file data to.
int ii,jj,kk,mm;
double A,B,C;
char peanut_butter1[200];
char peanut_butter2[200];
//Open the file and check that it exists
  fpin = fopen("data.input","r");
  if (fpin == NULL)
  {
    fprintf(stdout,"ERROR-file does not exist.\n");
    exit(10);
  }

//Line 1 of file
  fgets(temp_string,200,fpin);


//Line 2 of file
  fgets(temp_string,200,fpin);

  //We were expecting some integers here.
  //Lets assign them to ii,jj,kk,mm using sscanf

  sscanf(temp_string,"%d %d %d %d",&ii,&jj,&kk,&mm);

  //Lets print out these integers
  fprintf(stdout,"The integers are: %d %d %d %d\n",ii,jj,kk,mm);

//Line 3 of file
  fgets(temp_string,200,fpin);

//Line 4 of file
  fgets(temp_string,200,fpin);

  //We were expecting some doubles here.
  //Lets assign them to A,B,C using sscanf

  sscanf(temp_string,"%lf %lf %lf",&A,&B,&C);

//Lets print out these doubles
  fprintf(stdout,"The doubles are: %f %f %f\n",A,B,C);

//Line 5 of file
  fgets(temp_string,200,fpin);

//Line 6 of file
  fgets(temp_string,200,fpin);

  //We were expecting some strings here.
  //Lets assign them to peanut_butter1 and
 // peanut_butter2 using sscanf

  sscanf(temp_string,"%s %s",peanut_butter1, peanut_butter2);

//Lets print out these strings
  fprintf(stdout,"The strings are: %s and %s \n",peanut_butter1,peanut_butter2);

return 0;
}
```

You should get the following output:

```
The integers are: 0 1 2 3
The doubles are: 0.111100 0.222200 0.333300
The strings are: crunchy and creamy
```

Tip for a near future homework. In a future homework, we will apply the trapezoid rule to a double integral. Imagine that you have particular values of a 2D function *f(x,y)* organized as a mesh of points. Each point has an x and y coordinate, and each point can be associated with a particular row or column of points. When applying the trapezoid rule to such a mesh, it is very convenient to discern whether a particular point is one a **corner** of that mesh, the **edge** of that mesh, or **inside** that mesh. The following code shows how to do that.

First of all, create a data file named *code26.data* The first line of this file will indicate the *number of rows* and the *number of columns*, respectively. The number of rows is simply the number of x values on that mesh, and the number of columns is the number of y values on that mesh. Following that header line will be the x and y coordinates of each point of that mesh. The data points are organized such that you start at the bottom-left of the mesh (row 1 and column 1), progress through each column (i.e., each y value), and then move up to the next row. Following this system to the end, you should realize that the last line of data will contain the x and y values of the top-right of the mesh. For a rectangular mesh composed of 3 rows and 4 columns with the following corners (x=0,y=0) and (x=3, y=2), the data file *code26.data* would look like this:

```
3  4
0.0  0.0
1.0  0.0
2.0  0.0
3.0  0.0
0.0  1.0
1.0  1.0
2.0  1.0
3.0  1.0
0.0  2.0
1.0  2.0
2.0  2.0
3.0  2.0
```

You will want to carefully reverse-engineer the following code, *code26.c*. This code reads each line of the file and determines whether the particular point is on the corner of the mesh, on the edge of the mesh, or inside of the mesh.

code26.c

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

int main(void)
{

char temp_string[200];
FILE *fpin;

int number_of_rows;
int number_of_columns;
int number_of_coordinates;
int icolumn, irow;
double x,y;

//Open the file and check that it exists
  fpin = fopen("code26.data","r");
  if (fpin == NULL)
  {
    fprintf(stdout,"ERROR-file does not exist.\n");
    exit(10);
  }

// Read the number of rows and columns
  fgets(temp_string,200,fpin);
  sscanf(temp_string,"%d %d",&number_of_rows,&number_of_columns);

// Determine the number of coordinates;

  number_of_coordinates = number_of_rows*number_of_columns;

  fprintf(stdout,"There are %d coordinates\n",number_of_coordinates);

// The data is arranged in a format that sweeps through each column
// of each row before advancing to the next row.  Therefore, we can
// read the coordinates in through the following loop.

  for (irow = 1 ; irow <= number_of_rows; irow++)
  {
  for (icolumn = 1 ; icolumn <= number_of_columns; icolumn++)
  {
     //read a line of data
     fgets(temp_string,200,fpin);
     //extract the x and y coordinates of that point
     sscanf(temp_string,"%lf %lf", &x,&y);

     //Print out some information.
     //Note, that I did not include the carriage return, \n
     fprintf(stdout,"Row: %d Column: %d X: %f Y:%f ",irow,icolumn,x,y);

     // Is this a corner point?
     if ( ((irow == 1) || (irow == number_of_rows)) &&
       ((icolumn == 1) || (icolumn == number_of_columns)))
     {
       fprintf(stdout,"This is a corner point\n");
```

```
         }

        // If not a corner, then perhaps an edge?

        else if ((irow ==1) || (irow == number_of_rows) || (icolumn == 1)
                || (icolumn == number_of_columns) )
        {
            fprintf(stdout,"This is an edge point\n");
        }

        // Otherwise, this must be an internal point

        else fprintf(stdout,"This is an internal point\n");

    }
    }

  return 0;
  }
```

You should get the following output:

```
There are 12 coordinates
Row: 1 Column: 1 X: 0.000000 Y:0.000000 This is a corner point
Row: 1 Column: 2 X: 1.000000 Y:0.000000 This is an edge point
Row: 1 Column: 3 X: 2.000000 Y:0.000000 This is an edge point
Row: 1 Column: 4 X: 3.000000 Y:0.000000 This is a corner point
Row: 2 Column: 1 X: 0.000000 Y:1.000000 This is an edge point
Row: 2 Column: 2 X: 1.000000 Y:1.000000 This is an internal point
Row: 2 Column: 3 X: 2.000000 Y:1.000000 This is an internal point
Row: 2 Column: 4 X: 3.000000 Y:1.000000 This is an edge point
Row: 3 Column: 1 X: 0.000000 Y:2.000000 This is a corner point
Row: 3 Column: 2 X: 1.000000 Y:2.000000 This is an edge point
Row: 3 Column: 3 X: 2.000000 Y:2.000000 This is an edge point
Row: 3 Column: 4 X: 3.000000 Y:2.000000 This is a corner point
```