More on Formatting

The following code illustrates some interesting ways in which we can format our output. Please play around and experiment a bit on your own. To make nice looking output files, learning these formatting options will surely pay off.

code18.c

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

// This program shows how to format output differently.

int main(void)
{

// declare a double and an integer

double A=1.23456789e3;
int P=3;

// Here is the standard default way to print a real number
fprintf(stdout,"%f \n",A);

//By putting a plus sign in front of the f, we can force
//it to include a sign in front of the number.
fprintf(stdout,"%+f \n",A);
fprintf(stdout,"%+f \n",-1*A);

//We can force the field of the number to be a
//certain size.  In this example, 15.
//15 spaces will be reserved for the number.
fprintf(stdout,"%15f \n",A);

// Here, we reserve the 15 spaces, but force
// zeros to occupy any empty spaces
fprintf(stdout,"%015f \n",A);

// Here, we force the number to display a
// certain number of decimal places.  In this case,
// only 2 spaces.
fprintf(stdout,"%.2f \n",A);

// You can mix and match too.  In the following
// example, we combine several things.
fprintf(stdout,"%+015.2f \n",A);

// If you want to skip 3 lines, you can do the following
fprintf(stdout,"\n\n\n");

// He we print the number in scientific notation
fprintf(stdout,"%e \n", A);
// We can make the e and E in the scientific notation
fprintf(stdout,"%E \n", A);

// We can use the same tricks for e or E that we could for F.
// For example
fprintf(stdout,"%30.10e \n", A);

// Ok, on to integers.

// The default integer print is as follows
fprintf(stdout,"%d \n", P);

// To make the field 8 spaces wide
fprintf(stdout,"%8d \n",P);

// To make the field 8 spaces wide
// and to pad the empty spaces with
// zero.
fprintf(stdout,"%.8d \n",P);

// To force the integer to have a sign in front
fprintf(stdout, "%+d \n", P);

return 0;
}
```

Output

```
1234.567890
+1234.567890
-1234.567890
   1234.567890
00001234.567890
1234.57
+00000001234.57



1.234568e+03
1.234568E+03
              1.2345678900e+03
3
       3
00000003
+3
```

Writing to Files

Writing data to a file is one of the more common tasks in programming. Typically, the procedure involves (1) opening a file, (2) writing to the file, and (3) closing the file. To do so, we need to use a particular type of variable called a **file pointer**. A **pointer** in C is a variable that points to a particular memory address. A **file pointer** is relatively straightforward and is much easier to understand than the pointers we will deal with later.

A **file pointer** is a variable of type **FILE**. We can name a file pointer variable any name we would like (as long as they aren't one of the special words as shown earlier); however, it must have a star immediately before the name. We'll see later that all pointers are prefixed by a star. It is common practice to name file pointer variables something that starts with letters "*fp*". I like to name file pointers names such as *fpin* or *fpout* (or something like that), but that is purely my personal preference. You can name them any permissible variable name.

To declare a file pointer named *fpout*, we would type:

```
FILE *fpout;
```

You can have multiple file pointers open at one time. This may be useful if you are reading and writing to multiple files in the same code. **A note of caution** however: **There is a maximum number of files that can be safely open at one time**! This number is machine dependent. For example, on my machine, 16 is the maximum number of files that can be safely opened simultaneously. There is a variable included in *stdio.h* called **FOPEN_MAX** that tells how many files can be safely opened at one time.

To open a file, we use the function named **fopen(***name of file*, *type_of_access***)**. Where *type_of_access* can be **r** for read, **w** for write, or **a** for append. **w** will assume that you are writing the file from scratch, meaning if the file already exists, it will be overwritten. **a** will simply add to a file that already exists.

Writing to a file is as simple as writing to the screen. We'll use the **fprintf** function, but instead of using *stdout* (which is a file pointer to standard output (i.e., the terminal)), we'll use the name of our file pointer.

To close a file, we use the function named **fclose(***file_pointer***)**. **fclose** detaches the file pointer from the file. Its good practice to close file, but in actuality, all open files automatically close when the code finishes.

The following tutorial shows how to print something to file.

code19.c

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


int main(void)
{

// We declare a file pointer here.
// We'll name the file pointer fpout.

FILE *fpout;

// Open a file named my_file.out
// We do this by setting the file pointer to the fopen
// function.  We specify that this is a file
// to be written to.

  fpout = fopen("my_file.out", "w");

// Lets write something to this file

  fprintf(fpout,"Hey, whats up? Hows it going?\n");

// Lets write some more to this file

  fprintf(fpout,"Not much. Still circling the drain.\n");

// Lets close the file now.

  fclose(fpout);

return 0;
}
```

After running the code, use vi to open the file *my_file.out*. You should see the following lines in the file:

```
Hey, whats up? Hows it going?
Not much. Still circling the drain
```

The next code is an example of (1) Writing to multiple files at once, and (2) appending to a file. We'll write to a new file named *my_new_file.out* and we'll append to our previous file *my_file.out*.

code20.c

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

int main(void)
{

// We declare 2 file pointers here.

FILE *file_pointer_1;
FILE *file_pointer_2;

// We'll set file_pointer_1 to append to the
// pre-existing file, my_file.out.

  file_pointer_1 = fopen("my_file.out", "a");

// We'll set file_pointer_2 to write to the
// new file, my_new_file.out. If my_new_file
// already exists, it will be overwritten.

  file_pointer_2 = fopen("my_new_file.out", "w");

// Lets write something to these 2 files

  fprintf(file_pointer_1,"Stop your complaining\n");

  fprintf(file_pointer_2,"Hey, I'm a new file\n");

// Lets close the files.

  fclose(file_pointer_1);
  fclose(file_pointer_2);

return 0;
}
```

Now check your 2 files to make sure it worked.

Oftentimes, it is useful to have some error control when opening files. For example, if you want to open a file to read from, you would like your code to tell you if that file does not exist. OR, if you want to write to a file, sometimes you may not be able to because the disk is full or perhaps you don’t have write access. In this case, you would want your code to inform you about this, and perhaps you would even want it to exit. Therefore, it is better to have a “safer” way to open files.

Before we continue, it is worthwhile to explain what happens if the **fopen** function fails to execute. If **fopen** cannot open the file, it will return a value called **NULL**. Therefore, we can check whether fopen returns a **NULL** value.

In the following tutorial, we will put this into action. Say we want to open a file named *FILE01.input* for reading. We would like our code to quit if *FILE01.input* doesn't exist.

code21.c

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

int main(void)
{

// Declare a file pointer named fpin
FILE *fpin;

// Try to open for reading.  If it doesn't exist, quit the code.
fpin = fopen("FILE01.input","r");
if (fpin  == NULL)
{
   fprintf(stdout,"This file does not exist dumdum\n");
   exit(10);
}
else
{
   fprintf(stdout,"This file does exist\n");
   exit(10);
}

return 0;
}
```

Because FILE01.input does not exist, it couldn't be opened for reading. Therefore, the file pointer *fpin* was set to **NULL**. Our IF statement checked for this, and promptly gave us a nasty error message and exited the code.

```
This file does not exist dumdum
```

Ok, lets give it a file that we do know exists. Change the **fopen** function to open *my_file.out*, the file we created in an earlier tutorial.

```
fpin = fopen("my_file.out","r");
```

Now the code should output the following:

```
This file does exist
```

Here, is *code19.c* recoded in a safer way. If the file *my_file.out* is not able to be created, the code will stop.

code19.c (recreated in a safer way)

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

// This code will simply print
// the words "Hi there" to a file
// named "my_file".

int main(void)
{

// We declare a file pointer here.
// We'll name the file pointer fpout.

FILE *fpout;

// Open a file named my_file.out
// We do this by setting the file pointer to the fopen
// function.  We specify that this is a file
// to be written to.

  fpout = fopen("my_file.out", "w");
  if (fpout == NULL)
  {
     fprintf(stdout,"The file cannot be opened\n");
     exit(10);
  }

// Lets write something to this file

  fprintf(fpout,"Hey, whats up? Hows it going?\n");

// Lets write some more to this file

  fprintf(fpout,"Not much. Still circling the drain.\n");

// Lets close the file now.

  fclose(fpout);

return 0;
}
```