Printable View of: Week 13: Miscelaneous cool features

Print Save to File

File: returns from standard functions: scanf(), fopen()

returns from standard functions: scanf(), fopen()

Returns from standard functions

fopen( )

I figured I would start with the obvious one. When you do an fopen( ) there are 2 obvious results. The file is opened succesfully, or it is not. Let's consider the statement

```
fp=fopen(filename,"r");
```

Since you are opening the file to read from it, it would follow that if the file is not there or cannot be opened, that is a bad thing, and you want to know about it. All you need to do is check the value stored in the variable "fp" after the fopen( ) is called. fopen( ) puts a pointer to the file in "fp" if it opens the file successfully. If the file cannot be opened or does not exist, fopen( ) returns a null pointer, so we can test "fp" for that. Let's make a first try at this:

```
fp=fopen(filename,"r");
if (fp!=NULL){
  <Do stuff here>
}else{
  <Error handling>
}
```

Now, this looks nice, but if you think about it, in practice the "Do stuff here" part will usually be the rest of the program, and the "Error handling" part will usually contain an error message printf( ) and an exit( ). The exit( ) is usually there because if the program needs to read stuff from a file that can't be opened, what else can you do? So, instead of checking for `fp!=NULL` we can reverse the condition and just exit( ) if it is true. We don't even need the else.

```
fp=fopen(filename,"r");
if (fp==NULL){
  printf("cannot open %s for reading\n",filename);
  exit(3);
}
<Rest of the program>
```

Please make sure you convince yourself that these two are equivalent before you continue. The "3" in the exit( ) parameter list is just so that if you are testing the exit status of the program you would know why the program terminated. 0 is usually normal termination, and any other value is an error condition. All the different error conditions would be explained in the progam documentation.

Now that you are ready to go on, we can continue to improve this code. Notice that you don't need to compare fp to NULL. Since fp is either NULL (zero) or a valid address (non-zero) it is already "true" or "false" on it's own. Instead of `fp==NULL`, all we need to do is `!fp`.

```
fp=fopen(filename,"r");
if (!fp){
  printf("cannot open %s for reading\n",filename);
  exit(3);
}
<Rest of the program>
```

Again, make sure you are convinced this is the same. Try going through both possible cases and see that you get the same true/false result either way. Also, notice in the error message we included the name of the file. You should always put such information in error messages so that the programmer (usually you!) will have an easier time diagnosing the problem. It may be as simple as an incorrect filename, and with this error message, you would know right away.

Finally, because we are now real C programmers, we know that we can combine the assignment and the test on "fp". Take a look at this baby:

```
if (!(fp=fopen(filename,"r"))){
  printf("cannot open %s for reading\n",filename);
  exit(3);
}
<Rest of the program>
```

Notice that you need the extra set of ( )'s because the "!" has precedence of the assignment operator ("="). This is the way you should **always** implement an fopen( ). fopen( ) failure is a very common cause for programming crashes. If you always test, you will save yourself lots of time chasing non-existenet errors. You would be suprised how long it can take you to figure out the you have the filename wrong, or you fogot to put the file in the directory, or that you accidentally deleted it. Trust me on this one! ;)

Now, if you are really trying to make a clean program, you can make an error handling function that takes an integer input so it knows what error message to print and what exit status to send when it calls the exit( ) function. So, the whole big mess from our first example could be reduced to this:

```
if (!(fp=fopen(filename,"r"))) errorhandler(3);
```

Now, what about opening a file for writing?

```
fp=fopen(filename,"w");
```

Everything is pretty much the same. If you want to write stuff to a file and the fopen( ) fails, you usually want to exit. But, in either case, reading or writing, you don't have to exit. If you are trying to get data, you could ask the user for another file name, or you could simply ask the user to enter the data, if it isn't alot. In the case of writing, you could ask the user for another filename, or you could just write to standard output, again if there isn't too much data. But, usually, you just exit and let the user fix the problem and re-run the program. Sometimes this can be handled by a shell program that is running the C program. This is where the exit status value comes in. But that is for your shell programming course :)

scanf( ) and fscanf( )

Note: Except for **where** the function is getting the data, these two functions are identical, so anything I say about scanf( ) holds for fscanf( ).

scanf( ) returns the number of items successufly read. This can be very useful, particularly with fscanf( ) if you are having trouble detecting an end-of-line or end-of-file condition. Just as with the fopen( ), you could do something like this:

```
if (scanf("%d %s %f",&int1,str1,&float1)!=3)
```

Or:

```
if (3!=scanf("%d %s %f",&int1,str1,&float1))
```

both are the same, it is just a question of which one **you** can read better. Also, you may need to save the value returned from scanf( ) for further error processing, so you could do:

```
if (3!=(n=scanf("%d %s %f",&int1,str1,&float1)))
```

Or:

```
if ((n=scanf("%d %s %f",&int1,str1,&float1))!=3)
```

In any of these cases, the if expression would evaluate to false if the scanf( ) failed to read all 3 items. You can use this to debug, or to intentionally stop processing once you run out of data. The latter would be the case if you were not sure that you had an "even" amount of data in the file. You might have a file with many lines of data, but the last line is incomplete. Or, there is a line in the middle of the file that is incomplete and you need to identify that line. In any case, you get the idea. :)

The last trick about fscanf( ) is that it returns `EOF` (which is just -1) if it detects an end-of-file *before* it finds any data to match what it is trying to read. If there is data, but it doesn't match the first conversion character in the parameter list, the fscanf( ) will just return 0. This is, of course, very handy when you don't know when the data in the file will be ending, or if there is even any there to start with.

File: sscanf(), sprint()

sscanf(), sprint()

Reading from strings and writing to strings

sprintf( )   (pronounced "es-print-ef")

You already know how to put characters into a char array:

```
char string[80];
strcpy(string,"Win one for the Gipper");
```

You can also use `scanf("%s",string);` and have the user type the characters. Or use gets( ), etc. But I

bet you didn't know this one:
```
sprintf(string,"Win one for the Gipper");
```
The sprintf( ) here does exactly the same thing as the strcpy( ) above. In the case of the sprintf( ) notice that it looks just like and fprintf( ) except instead of a filename to write to, you give a char array name to write to. So after the sprintf( ) executes, the character array "string" contains "Win one for the Gipper".

Now, at first you may say, "big deal Sean", but consider this. Suppose I wanted to create a filename based on the user input. The user needs to tell me what "number" file this is and I build the filename and then open it, or read from it, or do whatever it is I need to do with it. After getting the integer "n" from the user, I could do this:
```
char filename[80];
int n;
...
scanf("%d",&n);
sprintf(filename,"userdata%d.dat",n);
fp=fopen(filename,"r");
etc...
```
If the user enters a 23, then the file "userdata23.dat" is the file that the fopen( ) will try to open. Please be very sure that you are clear about the variable "filename". I used it here just to confuse you :) Actually, I used it to make sure you were clear on what is going on. The var "filename" is a character array. In the sprintf( ) we are putting characters into the array "filename". In the fopen( ) we are using the contents of the array as the name of the file we are trying to open. Make sure this is clear.

This is just one example of how to use sprintf( ). Now that you know it, I'm sure you can thing of billions more. Remember project 1? Could you have used an sprintf( ) to help you managing file names there? Probably :)

sscanf( )   (pronounced "es-scan-ef")

This function works the same as sprintf( ), analogous to fscanf( ). Instead of reading from a file and giving the filename as the first parameter, sscanf( ) reads from a character array and gives the name of the character array as the first parameter. Just to give an example, let's load a string with a strcpy( ) and then do an sscanf( ) ...
```
char string[80],dummy[80];
int hours;
float pay;
...
strcpy(string,"I made $75.67 for 3 hours work");
sscanf(string,"%s %s $%f %s %d",dummy,dummy,&pay,dummy,&hours);
printf("pay = %.2f hours = %d\n",pay,hours);
```

If you run this code (and you should!) you will get

```
pay = 75.67 hours = 3
```

as output. Notice that the sscanf( ) read the "I" and "made" and "for" into a string called "dummy" because we weren't interested in those strings, we just wanted to get past them. Also notice that I put the "$" in the control string of the sscanf( ) to match the "$" exactly in the input so that I could then match the float with a %f right after it. All these are tricks that work exactly the same with scanf( ) or fscanf( ), I just wanted to use this space to show this stuff too.

So, what good is sscanf( )? I thought you would never ask. The biggest and best way to use sscanf( ) is to "bullet-proof" your interactive code. Users are stupid. They are always entering data incorectly. If a regular scanf( ) or fscanf( ) fails, it is very difficult to deal with, and can often cause the program to behave unpredictably, or even to crash. Yuc. The trick is to read a line of data into a string with a gets( ), or fgets( ), or %[ ] (see next section). Now you have the whole line in a character array and you can read it with an sscanf( ) and not worry about crashing the program. You can also read it as many times as you like with several different sscanf( )'s until one reads the data correctly. Pretty neat, huh?

You can process filenames, for example, with sscanf( ). If you have a number in the filename somewhere, as in our "userdata23.dat", you can find that value with an sscanf( ). Because you know the format of the filename, you can include most of the characters in the control string since the will match exactly:
```
sscanf(filename,"userdata%d.dat",&n);
```
and now "n" holds the value 23, or whatever was there in the character array.

File: reading strings with %[ ] and %*

reading strings with %[ ] and %*

Tricky conversion characters

Using `%[ ]` instead of `%s`

If you are familiar with UNIX, you will recognize the "[ ]" construct. UNIX uses this to match any single character enclosed in the [ ]'s. In C scanf( ) control strings the usage is simmilar. If I want to read only lower case characters into a string, I can use:

`scanf("%[a-z]",string);` instead of `scanf("%s",string);`

If I type "funwithDickandJane" (sorry about the lack of spacing, but I'm trying to keep this simple) then the 2nd scanf( ) above using the %s would load all the characters into the array "string". The first scanf( ) with the %[a-z] would stop at the "D" because it doesn't match the condition. So, in the first case, the characters in "string" after the scanf( ) would be only "funwith".

You can do ranges with %[ ] as we have above, or you can specify individual characters. If we want to allow spaces as well, we can do:

```
scanf("%[a-z ]",string);
```

Now, if the input is "fun with Dick and Jane", this scanf will load "fun with " into the string "string". This is because we included any lower case char **and** a blank as valid characters to read in. Notice also that if we just used %s, all we would get in "string" is "fun", because %s stops reading at blank spaces.

You can also *exclude* specific characters with %[ ] so you get a "read anything except this character" effect. For example:

```
scanf("%[^0-9]",string);
```

The ^ (carat) means that this scanf( ) will load any character *except* ones that match the characters listed. So if the input is "I made $75.67 for 3 hours work" then the array "string" will contain "I made $" after the above scanf( ), because it stops loading characters when it sees the "7". This is because the "7" is included in the range "0-9".

All this leads us to the most common usage of the %[ ] which is to read an entire line, regardless of what characters are on the line. We can accomplish this by using the ^ to *disallow* only a newline character (\n) and thus allow any other char

```
scanf("%[^\n]",string);
```

Now whatever is on the line, is loaded into "string". The only minor problem is that the newline is **not** loaded, and if you try to do 2 of these in a row, the second will fail (not read anything) because the first character it sees is the newline that caused the last scanf( ) to finish. You could do this:

```
scanf("%[^\n]%c",string,&dummy);
```

where "dummy" is a char variable, but we will have a much better solution in the next section :)

Using `%*` to *skip* data

The %* does not stand by itself, it goes with a conversion character. %*s or %*c for example. This has just about my favorite name in programming ... it is called *assignment supression*. Which just means that you match the int, char, float, or string, but you do not *assign* the value to a variable. This takes care of the problem we just had with the newline character:

```
scanf("%[^\n]%*c",string);
```

How cool is that? The %c matches the newline character, but because the * is there, the scanf( ) doesn't try to put it into a variable. So, we don't need to put a variable in the parameter list and we don't need to declare "dummy" at all.

Another favorite example of mine is reading dates. You know the date is month,day,year, but you don't know if it is:

```
4-9-88
4/9/88
4,9,88
```

You can use `scanf("%d%*c%d%*c%d",&month,&day,&year);` and you don't care what character is separating the integers.

Let's go back to that ugly sscanf( ) we did last section:
`sscanf(string,"%s %s $%f %s %d",dummy,dummy,&pay,dummy,&hours);`
first, we can cut down on the number of strings we need with the %[ ]
`sscanf(string,"%[^$]$%f%[^0-9]%d",dummy,&pay,dummy,&hours);`
Then we can get rid of the variable "dummy" with the %*
`sscanf(string,"%*[^$]$%f%*[^0-9]%d",&pay,&hours);`
Are we good, or what? Of course you will have to taylor this to whatever your input requires, but you can see the value here, both in increased flexibility and reduced requirements (we don't need the "dummy" variables all over the place).

File: fflush( )

fflush( )

the fflush( ) function

Note: you will need to include stdio.h for fflush( ) to work

Using `fflush(stdout)` for degugging

Here is the scenario: Your program is crashing. You put in 5 printf( ) statements (compiled conditionally, of course!) and run the program. You see the output from 3 of the printf( )'s so you start looking between the 3rd and 4th printf( )'s for the problem. You spend days and days looking. You lose your job and your self esteem. Then your landlord evicts you. Wait, we are getting off track :) The point is, you can't find the problem there. Well, that is because it is actually *after* the 4th printf( ). How? Because output is "buffered" which means the machine doesn't necessarily dump the output as soon as the printf( ) executes. And if the program crashes, the output that is in the buffer doesn't necessarily make it to the screen. So, you are looking in the wrong place. Perhaps the best solution is to learn how to use the symbolic debugger on your system. But, short of that, we have a quick fix. Each of your diagnostic printf( )'s should have right after it, the following statement:

```
fflush(stdout);
```

This causes any output in the output buffer to be cleaned out (sent to stdout) right away. So, if the program crashes, you have a much better idea of where!

Using `fflush(stdin)` with scanf( )

A very common problem occurs when you try to read character data after reading numeric data. Write a little program that does this, run it and see what happens (don't look ahead until you have run the program!):

```
printf("please enter an int\n");
scanf("%d",&n);
printf("please enter your initial\n");
scanf("%c",&initial);
```

So? You noticed that the 2nd scanf( ) didn't wait for you to type anything. What happened? Well, when asked for an integer, you typed the integer and then **you typed the *enter* key**! That is a character just like any other character. Is there anything in the control string "%d" that would match that character? Nope. So it (the newline char) just sits there waiting to be read. Now along comes the next scanf( ) and it has a %c which is looking for a single character. Well, there is one there on stdin waiting to be read. So, the newline gets put into "initial" and the program continues along happily. We can fix this. One way would be to match the newline with a %*c as in the last section, but that wouldn't help us if the user enterend a space and then the newline, or even accidentally hit some thing like a "-" because he/she has big fingers and when he/she typed the "0" he/she got the "-" too. In any case, the safer solution is to use `fflush(stdin)` to clear out the input buffer before you ask for more input:

```
printf("please enter an int\n");
scanf("%d",&n);
printf("please enter your initial\n");
fflush(stdin);
scanf("%c",&initial);
```

Now try your program again, and you get the desired effect! Yea!

File: Assignment 13

Assignment 13

Assignment 13

Problem solving in "C"
Assignment 13
0 points
Due: ???

The Assignment:

Fooled you! There is no assignment 13! Sort of like there is no 13th floor in a high rise building. I bet I had you worried though. :)

Finish up old assignments and get that project done. Post or email me questions.

Print **Save to File**