**C Programming Assignment #007**

Date: Kamis, 15 Oktober 2015

PT. Bahasa Kinerja Utama

Husni Fahmi

[fahmi@bahasakita.co.id](mailto:fahmi@bahasakita.co.id)

**Intro to File Input/Output in C**

1. **Redirection:**

One way to get input into a program or to display output from a program is to use *standard input* and *standard output*, respectively. All that means is that to read in data, we use scanf() (or a few other functions) and to write out data, we use printf().

When we need to take input from a file (instead of having the user type data at the keyboard) we can use input redirection:

% **a.out < inputfile**

This allows us to use the same scanf() calls we use to read from the keyboard. With input redirection, the operating system causes input to come from the file (e.g., inputfile above) instead of the keyboard.

Similarly, there is *output redirection*:

% **a.out > outputfile**

that allows us to use printf() as before, but that causes the output of the program to go to a file (e.g., outputfile above) instead of the screen.

Of course, the 2 types of redirection can be used at the same time...

% **a.out < inputfile > outputfile**

1. **C File I/O:**

While redirection is very useful, it is really part of the operating system (not C). In fact, C has a general mechanism for reading and writing files, which is more flexible than redirection alone.

**stdio.h**

There are types and functions in the library **stdio.h** that are used for file I/O. Make sure you always include that header when you use files.

**Type**

For files you want to read or write, you need a file pointer, e.g.:

FILE \*fp;

What is this type "FILE \*"? Realistically, you don't need to know. Just think of it as some abstract data structure, whose details are hidden from you. In other words, the only way you can use a FILE \* is via the functions that C gives you.

**Note:** In reality, FILE is some kind of structure that holds information about the file. We must use a FILE \* because certain functions will need to change that information, i.e., we need to pass the information around *by reference*.

**Functions**

Reading from or writing to a file in C requires 3 basic steps:

* 1. Open the file.
  2. Do all the reading or writing.
  3. Close the file.

Following are described the functions needed to accomplish each step.

A complete program that includes the example described below, plus an input file to use with that program, is available to [download](http://www.cs.bu.edu/teaching/c/file-io/intro/download/).

1. **Opening a file:**

In order to open a file, use the function fopen(). Use it as:

fp = fopen(*filename*, *mode*);

where:

* 1. *filename* is a string that holds the name of the file on disk (including a *path* like /cs/course if necessary).
  2. *mode* is a string representing how you want to open the file. Most often you'll open a file for reading ("r") or writing ("w").

Note that fopen() returns a FILE \* that can then be used to access the file. When the file cannot be opened (e.g., we don't have permission or it doesn't exist when opening for reading), fopen() will return NULL.

Here are examples of opening files:

FILE \*ifp, \*ofp;

char \*mode = "r";

char outputFilename[] = "out.list";

ifp = fopen("in.list", mode);

if (ifp == NULL) {

fprintf(stderr, "Can't open input file in.list!\n");

exit(1);

}

ofp = fopen(outputFilename, "w");

if (ofp == NULL) {

fprintf(stderr, "Can't open output file %s!\n",

outputFilename);

exit(1);

}

Note that the input file that we are opening for reading ("r") must already exist. In contrast, the output file we are opening for writing ("w") does not have to exist. If it doesn't, it will be created. If this output file does already exist, its previous contents will be thrown away (and will be lost).

**Note:** There are other modes you can use when opening a file, such as append ("a") to append something to the end of a file without losing its contents...or modes that allow you to both read and write. You can look up these other modes in a good C reference on stdio.h.

1. **Reading from or writing to a file:**

Once a file has been successfully opened, you can read from it using fscanf() or write to it using fprintf(). These functions work just like scanf() and printf(), except they require an extra first parameter, a FILE \* for the file to be read/written.

**Note:** There are other functions in **stdio.h** that can be used to read or write files. Look them up in a good C reference.

Continuing our example from above, suppose the input file consists of lines with a *username* and an *integer test score*, e.g.:

in.list

------

foo 70

bar 98

...

and that each username is no more than 8 characters long.

We might use the files we opened [above](http://www.cs.bu.edu/teaching/c/file-io/intro/#fileio1) by copying each username and score from the input file to the output file. In the process, we'll increase each score by 10 points for the output file:

char username[9]; /\* One extra for nul char. \*/

int score;

**...**

while (fscanf(ifp, "%s %d", username, &score) != EOF) {

fprintf(ofp, "%s %d\n", username, score+10);

}

**...**

The function fscanf(), like scanf(), normally returns the number of values it was able to read in. However, when it hits the end of the file, it returns the special value EOF. So, testing the return value against EOF is one way to stop the loop.

The bad thing about testing against EOF is that if the file is not in the right format (e.g., a letter is found when a number is expected):

in.list

------

foo 70

bar 98

biz A+

...

then fscanf() will not be able to read that line (since there is no integer to read) and it won't advance to the next line in the file. For this error, fscanf() will not return EOF (it's not at the end of the file)....

Errors like that will at least mess up how the rest of the file is read. In some cases, they will cause an *infinite loop*.

One solution is to test against the number of values we expect to be read by fscanf() each time. Since our format is "%s %d", we expect it to read in 2 values, so our condition could be:

while (fscanf(ifp, "%s %d", username, &score) == 2) {

**...**

Now, if we get 2 values, the loop continues. If we don't get 2 values, either because we are at the end of the file or some other problem occurred (e.g., it sees a letter when it is trying to read in a number with %d), then the loop will end.

Another way to test for end of file is with the library function feof(). It just takes a file pointer and returns a true/false value based on whether we are at the end of the file.

To use it in the above example, you would do:

while (!feof(ifp)) {

if (fscanf(ifp, "%s %d", username, &score) != 2)

break;

fprintf(ofp, "%s %d", username, score+10);

}

Note that, like testing != EOF, it might cause an infinite loop if the format of the input file was not as expected. However, we can add code to make sure it reads in 2 values (as we've done above).

**Note:** When you use fscanf(...) != EOF or feof(...), they will not detect the end of the file until they try to read past it. In other words, they won't report end-of-file on the last valid read, only on the one after it.

1. **Closing a file:**

When done with a file, it must be closed using the function fclose().

To finish our example, we'd want to close our input and output files:

fclose(ifp);

fclose(ofp);

Closing a file is very important, especially with output files. The reason is that output is often *buffered*. This means that when you tell C to write something out, e.g.,

fprintf(ofp, "Whatever!\n");

it doesn't necessary get written to disk right away, but may end up in a *buffer* in memory. This output buffer would hold the text temporarily:

Sample output buffer:

----------------------------------------------

| a | b | c | W | h | a | t | e | v | e | r |

----------------------------------------------

| ! | \n | | | | | | | | | |

----------------------------------------------

| | | | | | | | | | | |

----------------------------------------------

| | | | | | | | | | | |

----------------------------------------------

**...**

(The buffer is really just 1-dimensional despite this drawing.)

When the buffer fills up (or when the file is *closed*), the data is finally written to disk.

So, if you forget to close an output file then whatever is still in the buffer may not be written out.

**Note:** There are other kinds of buffering than the one we describe here.

1. **Special file pointers:**

There are 3 special FILE \*'s that are always defined for a program. They are stdin (*standard input*), stdout (*standard output*) and stderr (*standard error*).

**Standard Input**

*Standard input* is where things come from when you use scanf(). In other words,

scanf("%d", &val);

is equivalent to the following fscanf():

fscanf(stdin, "%d", &val);

**Standard Output**

Similarly, *standard output* is exactly where things go when you use printf(). In other words,

printf("Value = %d\n", val):

is equivalent to the following fprintf():

fprintf(stdout, "Value = %d\n", val):

Remember that standard input is normally associated with the keyboard and standard output with the screen, unless *redirection* is used.

**Standard Error**

*Standard error* is where you should display error messages. We've already done that above:

fprintf(stderr, "Can't open input file in.list!\n");

Standard error is normally associated with the same place as standard output; however, redirecting standard output does not redirect standard error.

For example,

% **a.out > outfile**

only redirects stuff going to standard output to the file **outfile**... anything written to standard error goes to the screen.

**Using the Special File Pointers**

We've already seen that *stderr* is useful for printing error messages, but you may be asking, "When would I ever use the special file pointers *stdin* and *stdout*?" Well, suppose you create a function that writes a bunch of data to an opened file that is specified as a parameter:

void WriteData(FILE \*fp)

{

fprintf(fp, "data1\n");

fprintf(fp, "data2\n");

...

}

Certainly, you can use it to write the data to an output file (like the one above):

WriteData(ofp);

But, you can also write the data to standard output:

WriteData(stdout);

Without the special file pointer *stdout*, you'd have to write a second version of WriteData() that wrote stuff to standard output.

*BU CAS CS - Intro to File Input/Output in C   
Copyright © 1993-2000 by Robert I. Pitts< rip at bu dot edu>. All Rights Reserved.*

===============x==========================================================

**Strings as arrays, as pointers, and string.h**

1. **Strings as arrays:**

In C, the abstract idea of a string is implemented with just an array of characters. For example, here is a string:

char label[] = "Single";

What this array looks like in memory is the following:

------------------------------

| S | i | n | g | l | e | \0 |

------------------------------

where the beginning of the array is at some location in computer memory, for example, location 1000.

**Note:** Don't forget that one character is needed to store the *nul character* (\0), which indicates the end of the string.

A character array can have more characters than the *abstract string* held in it, as below:

char label[10] = "Single";

giving an array that looks like:

------------------------------------------

| S | i | n | g | l | e | \0 | | | |

------------------------------------------

(where 3 array elements are currently unused).

Since these strings are really just arrays, we can access each character in the array using subscript notation, as in:

printf("Third char is: %c\n", label[2]);

which prints out the third character, **n**.

A disadvantage of creating strings using the character array *syntax* is that you must say ahead of time how many characters the array may hold. For example, in the following array definitions, we state the number of characters (either implicitly or explicitly) to be allocated for the array.

char label[] = "Single"; /\* 7 characters \*/

char label[10] = "Single";

Thus, you must specify the maximum number of characters you will ever need to store in an array. This type of array allocation, where the size of the array is determined at compile-time, is called *static allocation*.

1. **Strings as pointers:**

Another way of accessing a *contiguous* chunk of memory, instead of with an array, is with a *pointer*.

Since we are talking about *strings*, which are made up of *characters*, we'll be using *pointers to characters*, or rather, char \*'s.

However, pointers only hold an address, they cannot hold all the characters in a character array. This means that when we use a char \* to keep track of a string, the character array containing the string must already exist (having been either statically- or dynamically-allocated).

Below is how you might use a *character pointer* to keep track of a string.

char label[] = "Single";

char label2[10] = "Married";

char \*labelPtr;

labelPtr = label;

We would have something like the following in memory (e.g., supposing that the array label started at memory address 2000, etc.):

label @2000

------------------------------

| S | i | n | g | l | e | \0 |

------------------------------

label2 @3000

------------------------------------------

| M | a | r | r | i | e | d | \0 | | |

------------------------------------------

labelPtr @4000

--------

| 2000 |

--------

**Note:** Since we assigned the pointer the address of an *array of characters*, the pointer must be a *character pointer*--the types must match.

Also, to assign the address of an array to a pointer, we do not use the *address-of* (&) operator since the name of an array (like label) behaves like the address of that array in this context. That's also why you don't use an ampersand when you pass a string variable to scanf(), e.g,

int id;

char name[30];

scanf("%d%s", &id, name);

Now, we can use labelPtr just like the array name label. So, we could access the third character in the string with:

printf("Third char is: %c\n", labelPtr[2]);

It's important to remember that the only reason the pointer labelPtr allows us to access the label array is because we made labelPtr point to it. Suppose, we do the following:

labelPtr = label2;

Now, no longer does the pointer labelPtr refer to label, but now to label2 as follows:

label2 @3000

------------------------------------------

| M | a | r | r | i | e | d | \0 | | |

------------------------------------------

labelPtr @4000

--------

| 3000 |

--------

So, now when we subscript using labelPtr, we are referring to characters in label2. The following:

printf("Third char is: %c\n", labelPtr[2]);

prints out **r**, the third character in the label2 array.

1. **Passing strings:**

Just as we can pass other kinds of arrays to functions, we can do so with strings.

Below is the definition of a function that prints a label and a call to that function:

void PrintLabel(char the\_label[])

{

printf("Label: %s\n", the\_label);

}

...

int main(void)

{

char label[] = "Single";

...

PrintLabel(label);

...

}

Since label is a character array, and the function PrintLabel() expects a character array, the above makes sense.

However, if we have a pointer to the character array label, as in:

char \*labelPtr = label;

then we can also pass the pointer to the function, as in:

PrintLabel(labelPtr);

The results are the same. *Why??*

**Answer:** When we declare an array as the parameter to a function, we really just get a pointer. Plus, arrays are always automatically passed by reference (e.g., a pointer is passed).

So, PrintLabel() could have been written in two ways:

void PrintLabel(char the\_label[])

{

printf("Label: %s\n", the\_label);

}

OR

void PrintLabel(char \*the\_label)

{

printf("Label: %s\n", the\_label);

}

There is no difference because in both cases the parameter is really a *pointer*.

**Note:** In C, there is a difference in the use of brackets ([]) when declaring a global, static or local array variable *versus* using this array notation for the parameter of a function.

With a parameter to a function, you always get a *pointer* even if you use array notation. This is true for **all** types of arrays.

1. **Dynamically-allocated string:**

Since sometimes you do not know how big a string is until run-time, you may have to resort to dynamic allocation.

The following is an example of dynamically-allocating space for a string at run-time:

#include <stdlib.h> /\* for malloc/free \*/

**...**

void SomeFunc(int length)

{

char \*str;

/\* Don't forget extra char for nul character. \*/

str = (char \*)malloc(sizeof(char) \* (length+1));

**...**

Basically, we've just asked malloc() (the allocation function) to give us back enough space for a string of the desired size. Malloc() takes the number of bytes needed as its parameter. Above, we need the size of one character times the number of characters we want (don't forget the extra +1 for the *nul character*).

We keep track of the dynamically-allocated array with a pointer and can use that pointer as we used pointers to statically-allocated arrays above (i.e., how we access individual characters, pass the string to a function, etc. are the same).

*Now, how do we get a string value into this newly-allocated array?*

1. **string.h library:**

Recall that strings are stored as arrays (allocated either statically or dynamically). Furthermore, the only way to change the contents of an array in C is to make changes to each element in the array.

In other words, we can't do the following:

label = "new value"; /\* No! \*/

label = anotherLabel; /\* Wrong! \*/

(where anotherLabel is a string variable).

**Aside:** We could do that if label was a character pointer (instead of an array); however, what would be happening is the pointer would be taking on the address of a different string, which is not the same as changing the contents of an array.

It would be annoying to have to do something like:

char name[10];

name[0] = 'R';

name[1] = 'o';

name[2] = 'b';

name[3] = '\0';

or to write *loops* all the time to do common string operations... Plus, we'd probably forget the *nul character* half the time.

The C library string.h has several common functions for dealing with strings. The following four are the most useful ones that we'll discuss:

* + strlen(str)

Returns the number of characters in the string, not including the *nul character*.

* + strcmp(str1, str2)

This function takes two strings and compares them. If the strings are equal, it returns **0**. If the first is greater than the 2nd, then it returns *some* value **greater than 0**. If the first is less than the 2nd, then it returns *some* value **less than 0**.

You might use this function as in:

#include <string.h>

char str1[] = "garden";

if (strcmp(str1, "apple") == 0)

printf("Equal\n");

else

printf("Not equal\n");

OR

if (strcmp(str1, "eden") > 0)

printf("'%s' comes after 'eden'\n", str1);

The ordering for strings is lexical order based on the ASCII value of characters. Remember that the ASCII value of 'A' and 'a' (i.e., upper/lowercase) are not the same.

An easy way to remember how to use strcmp() to compare 2 strings (let's say *a* and *b*) is to use the following mnemonics:

|  |  |
| --- | --- |
| **Want...** | **Use...** |
| a == b | strcmp(a, b) == 0 |
| a < b | strcmp(a, b) < 0 |
| a >= b | strcmp(a, b) >= 0 |
| **...** | **...** |

* + strcpy(dest, source)

Copies the contents of *source* into *dest*, as in:

#include <string.h>

char str1[10] = "initvalue";

strcpy(str1, "second");

Now, the string str1 contains the following:

-------------------------------------------

| s | e | c | o | n | d | \0 | u | e | \0 |

-------------------------------------------

and the word "initvalue" has been overwritten. Note that it is the first *nul character* (\0) that determines the end of the string.

When using strcpy(), make sure the *destination* is big enough to hold the new string.

**Aside:** An easy way to remember that the destination comes first is because the order is the same as for assignment, e.g:

dest = source

Also, strcpy() returns the destination string, but that return value is often ignored.

* + strcat(dest, source)

Copies the contents of *source* onto **the end of** *dest*, as in:

#include <string.h>

char str2[10] = "first";

strcat(str2, " one");

Now, the string str2 contains the following:

------------------------------------------

| f | i | r | s | t | | o | n | e | \0 |

------------------------------------------

When using strcat(), make sure the *destination* is big enough to hold the extra characters.

**Aside:** Function strcat() also returns the destination string, but that return value is often ignored.

*BU CAS CS - Strings as arrays, as pointers, and string.h   
Copyright © 1993-2000 by Robert I. Pitts< rip at bu dot edu>. All Rights Reserved.*

<http://www.cs.bu.edu/teaching/c/file-io/intro/>

<http://www.cs.bu.edu/teaching/c/string/intro/>

Exercises

1. Write a program to read a WAV file as input. Read the WAV file with increment of 3200 bytes until the end of file. Print the number of bytes read in each iteration. The last chunk of bytes may not be 3200 bytes. Write each chunk of bytes into another file named <input filename>\_copy.wav.

For example, the executable filename is readwav:

% ./readwav FS\_001.wav

Output is:

FS\_001\_copy.wav