File I/O

Programming with C

CSCI 112, Spring 2015

with additions by Mary Ann Cummings, 9/2020

Patrick Donnelly

Montana State University

MONTANA STATE UNIVERSITY

Input/Output Methods

There are two approaches to achieve Input/Output in C.

Redirection

- Achieved using the OS.
- Simple way to change how a program reads and writes data.
- Often used to quickly test a program by simulating a user.

Pointers to Files

- Achieved using C.
- Considered the more "proper" way to access data in a file.
- Used when a program relies on data in a file in order to run.

Redirection - How It Works

Standard Input/Output

- printf() sends characters to standard output (stdout).
- scanf() reads characters from standard input (stdin).

By default, stdout and stdin both use the console.

Redirection

Within linux, we can redirect stdout and stdin to point elsewhere.

We specify this when typing the command to start the program.

- If we redirect stdout to use a file, printf() sends characters to the file instead of the console.
- If we redirect stdin to use a file, scanf() reads characters from a file rather than the console.

Redirection - Usage

Redirecting stdout

Redirect stdout by adding the '>' character, followed by the name of the file to redirect to.

```
$ ./program >output_file.txt
```

Redirecting stdin

Redirect stdin by adding the '<' character, followed by the name of the file to redirect from.

```
$ ./program <input_file.txt
```

We can also do both:

```
$ ./program <input_file.txt >output_file.txt
```

Redirection - EOF

Unlike with user input, files eventually come to an end.

To see if a file has ended, check the return value of `scanf()`.

scanf() return value

- Until this point, we have ignored its return.
- scanf() returns the number of items it was able to read
- If the end of the file has been reached, scanf() returns a negative value
  - This value represents the end of file constant (EOF)

```
int item;
while (1 == fscanf( inp, "%d", &item ))
    printf("Read %d\n", item);
printf("DONE\n");
```

Pointers to Files - Declaration

File pointers allow us to explicitly name a file to use *from within* C.

We achieve this using a pointer to the type FILE (a FILE pointer).

Declaration

```
FILE *inp;    // pointer 'inp' points to a file
FILE *outp;   // pointer 'outp' points to a file
```

This declares space on the stack for a pointer to a FILE. Initially it points at "nothing".

Pointers to Files - Declaration

To point a file pointer at a particular file, we must initialize it.

Use `fopen()` to point to get the address of a FILE object.
The `fopen()` function is found in the header file “stdio.h”.

Initialization

```
inp =fopen (" input_ file . txt", "r");
outp =fopen("output_file.txt", "w");
```

- The first argument is the file we wish to access.
- The second argument is *how* we want to access the file.
  - ”r” - read from the file
  - ”w” - write to the file (erase any previous content)
  - ”a” - append to the file (do not erase)

Pointers to Files - Usage

To scan from or print to a file, use the `fscanf()` and `fprintf()` functions.

These work just like `scanf()` and `printf()`, except that there is an additional first argument (the FILE pointer).

Usage

```
fscanf( inp , "%lf", &item ); fprintf(
outp, "Value: %.2f\n", item);
```

Again, `fscanf()` returns a negative value at the end of the file.
The same format codes apply to these functions as well.
For more information, see lecture notes on `scanf()` and `printf()`.

Pointers to Files - Moving the Pointer

The FILE pointer moves forward automatically as you read.
What if we want to move the pointer ourselves?

We can use `fgetpos()` and `fsetpos()` to move the pointer manually.

Usage

- The position in a file is represented in C by the type 'fpos_t'.
- `fgetpos(inp, &pos)` stores the position for `inp` in `pos`.
  - The first argument is the file pointer.
  - The second argument is the fpos_t variable.
- `fsetpos(inp, &pos)` sets the position for `inp` to `pos`.
  - The first argument is the file pointer.
  - The second argument is the fpos_t variable.

Pointers to Files - Moving the Pointer

Example

```
FILE *inp = fopen("in.txt", "r");
fpos_t start_pos;
fgetpos(inp, &start_pos);
// do something ...
fsetpos(inp, &start_pos);
```

Note that we can also move the pointer back to the beginning of the file at any time using the `rewind()` function.

- This has one argument: the FILE pointer.

```
rewind(inp);
```

Pointers to Files - Cleanup

When you are finished using a file, you must "close" them.
This frees the file up to be used by another program.

This is done using the `fclose()` function.

Usage

```
fclose( inp );
fclose(outp);
```

The argument is the pointer to the file you wish to close.

Similar to freeing memory, all FILE pointers will be closed when your program ends, but it is still good practice to close as you go.

Pointers to Files - Example

```c
#include <stdio.h>

int main( void) {

    double item;

    // declare an initialize FILE pointers  FILE
    *inp =fopen("input_file.txt", "r");  FILE
    *outp =fopen("outdata.txt", "w");

    // read until EOF, and write results to another file
    while (1 ==fscanf( inp , "%lf", &item ))
        fprintf(outp, "Value: %.2lf\n", item);

    // close files
    fclose( inp );
    fclose(outp);

    return 0;
}
```

Special Characters

We have already talked about the EOF character, and how `scanf()` returns the negative EOF value when it is reached. There are also other special characters, or escape sequences.

Common Escape Sequences

- '`\n`' - new line
- '`\t`' - tab
- '`\f`' - form feed (new page)
- '`\r`' - return (go to beginning of line)
- '`\b`' - backspace

These are all considered whitespace, and are therefore delimiters when using `scanf()`.

Each escape sequence is actually a character however, and can therefore be read in using the character (%c) format code with `fscanf()`.

fgets()

By default, `scanf()` reads special characters as white space.
To get around this, we can use the `fgets()` function instead.

fgets()

```
char *str =malloc( size );
fgets(str, size, stdin);
```

- 1st argument: Character array that should store input.
- 2nd argument: The maximum number of characters to store.
- 3rd argument: The input that should be used.

This will read one line of text (until the user presses enter).
In this case, spaces and special characters will be included.

Another Example

This program will count all the exclamation points and tabs.

```
int main( void ){

    FILE *inp =fopen("input_file.txt", "r");
    int points =0, tabs =0;
    char c;

    while ( EOF!= fscanf( inp , "%c", &c)) {
        if ( c =='!') points ++;
        if ( c =='\t') tabs ++;
    }

    fclose(inp);

    return 0;
}
```

Conclusions

Things To Remember

- Two methods of using files.
  - Redirection - Accomplished by OS
    Uses `scanf()` and `printf()` as before
  - FILE pointers - Accomplished by C
    Uses `fscanf()` and `fprintf()` with file pointers

Next Time

Data Structures

Binary Files

- To read large files in and print them, your program has expend alot of effort to convert it to binary then convert it back to ints, etc to print them.

- If there is no need for us to have to read these files, then we can
- make them binary files.
    - then we don't have to do this translation.

- A binary file has the computer’s internal representation of each data element in the file.

- Using binary files has the disadvantage of increasing the file size.
    - You can do a compression on the file for storage, which you can't do for text files.

- Other disadvantages - can't be read by a human, is not portable.

Binary File Function Calls

fopen
fclose
fread
fwrite

Advantages

Can read/write whole arrays/data structures with one call

Table 11.5 in the book