I/O Stream and Binary

The standard library uses file pointers.

```
FILE * stdin, * stdout, * stderr;
fprintf(stderr, "error\n");
```

Sometimes whether they're high or low level isn't always obvious such as:

```
int printf(const char *format, ...);
```

Which prints to stdout, but it defaults to the stream version of it.

```
printf("x");
fprintf(stdout, "x");
```

These are identical, but with fprintf we're explicitly saying that the output is stdout.

```
fprintf(stderr, "x");
//usually to print to files or print an error message
```

Compared to when you look up something low-level, there are no file structure pointers. There's a file descriptor number. Which is an offset into a file descriptor table.

```
ssize_t read(int fd, void *buf, size_t count);
```

`scanf`, `gets`, and `getchar` read from stdin, which is a file pointer set up by C (FILE * stdin, * stdout, * stderr). The corresponding functions `fscanf`, `fgets`, `getc`, and `fgetc` are the same thing, except you get to pass them a file pointer structure, which is returned by the fopen function. Each of these functions read buffered text data.
`printf` and `putchar` write to stdout, which is also a file pointer set up by C. The corresponding functions `fprintf`, `putc`, and `fputc` are passed a file pointer.
`fread` and `fwrite` read/write a specified number of bytes. The data can contain binary information. These functions are also passed a FILE pointer.

Example of fprintf printing to stderr:

```
$ ccat ~/stderr.c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char ** argv) {
	printf("This is standard output\n");
	fprintf(stderr, "This is standard error\n");
	return EXIT_SUCCESS;
}
```

Given:

```
char *fgets(char *s, int size, FILE *stream);
```

When fgets tries to read a line, if it can read a line, it stores the line in the buffer (char *s) up to and including the new line character.
"Say we give int size = 100; What it'll do is, if in 99 characters of reading from the stream, if none of those is a newline character, it'll stop, it'll have read the 99 characters, and it'll set the 100th character of your buffer equal to a null byte. It just null terminates the string. It's essential because you have no idea how large the lines will be coming from this stream (stdin or whatever it happens to be). You do need a failsafe where if you are going to exceed the size of the buffer, you can't let that happen.
Which will happen with: char *gets(char *s) since there's no way to control the length."

Standard Stream I/O

Stream Output:

```
printf(formatSpecifier, value1, ...);
```

Prints output to stdout based on the formatSpecifier. The number of values specified is dependent on the number of % format codes in the formatSpecifier.

```
fprintf(file, formatSpecifier, value1, ...);
```

Similar to printf except it prints the output to the specified file (which is usually opened with an fopen).

Stream Input:

```
char * fgets(stringVariable, maxLength, file);
```

This reads from the specified file until either maxLength - 1 characters are read or until a newline character is encountered. The read text is placed in stringVariable. fgets returns NULL on EOF, or the address of stringVariable otherwise.

```
int scanf(formatSpecifier, address1, address2, ...);
```

This reads from stdin based on the formatSpecifier. fscanf reads from a specified file. After the formatSpecifier, you must specify addresses since scanf stores its input into those addresses. scanf returns the number of successful conversions from its input that are stored into those addresses.

Input from Memory

```
int sscanf(stringVariable, formatSpecifier, address1, address2, …);
```

We will usually read a line of text into a variable using fgets and then use sscanf to take the data from that variable to populate our target variables.

Professor: scanf has a ton of problems. A big one is it leaves whitespace in the input stream; So for example if you ask for a number, and they type in a number and press enter. You can read that number, but the newline stays in the input stream. So the next time you say to grab a number, the newline character is still in the newline stream which creates a mess.
The terminal is almost entirely line oriented, but scanf is not. It's token oriented. So if I ask for three numbers separated by spaces, and they enter something like "my cats name is Sam", if the user enters that- you'll get a million errors from scanf because it'll take the word "my" and try to compare that to the first integer- but again on failure it doesn't clear out the input stream. It leaves it there so now you have to parse all the other invalid tokens like "cats" "name" "is" "Sam", plus the whitespace at the end of that.

It's much more convenient to write our program line-oriented, since the terminal is line-oriented.
The return value from fgets is a bit useless. If it encounters EOF, it'll return NULL. But if it succeeds, the return value is just fgets giving you your buffer back. It's not really meant to use anything, it's just used to create the true/false split.

fgets merely acquires the string that represents the line. sscanf is just like scanf, except sscanf has a new first parameter (the string that you're parsing). This will be the same buffer that contains the string that fgets obtained for us.

The whole scanf family; the return value is of type integer. What that number will represent is how many successful conversions were made. So if my format string is something like:

```
"%d %d %d"
```

I'd expect the return value to be 3. If it's fewer than 3, then at least one of those failed.

Example:

```
#include <stdio.h>
#include <string.h>

//the ellipses indicate that this is a variadic function
//this is for when the number of arguments for a function is unknown
//it takes one fixed argument, then any num of arguments can be passed
//this one specifically adds the word "error" in front of the string, //then
parses and prints your string, then exits with a failure status.

void errExit(const char szFmt[], ...); //prototype

int main(int argc, char ** argv) {
      char szInputBuffer[100];
      char szABC123[7]; //0 - 6 for abc123, 7th for null terminator
      double dGrade1;
      double dGrade2;
      int iScanfCount;

      printf("%-6s %-6s\n", "ABC123", "Grade");

      //read a line of text until EOF
      // note that fgets returns NULL when it reaches EOF
      while(fgets(szInputBuffer, 100, stdin)) {
       //szInputBuffer[strlen(szInputButter)-1] = '\0';
       //printf("RAW INPUT: '%s'\n", szInputBuffer);

       //copy the data to the target variables
       iScanfCount = sscanf(szInputBuffer, "%s %lf %lf",
                            szABC123, &dGrade1, &dGrade2);

       if (iScanfCount < 3) {

         fprintf(stderr, "ERROR: invalid input, please try again...\n");
         continue;
        }
       //print the abc123 ID and the higher grade
       if(dGrade1 > dGrade2)
            printf("%-6s %6.2lf\n", szABC123, dGrade1);
       else
            printf("%-6s %6.2lf\n", szABC123, dGrade2);
     }
  return 0;
}
```

Steps for above example:

1. while(fgets(szInputBuffer, 100, stdin)): We use fgets, that means our line that we read from stdin; each of my input lines, one by one, will be put into this buffer

2. So I give it the first line, it comes down here and tries to parse it: iScanfCount = sscanf(szInputBuffer, "%s %lf %lf",
szABC123, &dGrade1, &dGrade2);

3. Then we save the return value of sscanf (in iScanfCount) and check it in: if (iScanfCount < 3). If invalid we use 'continue', get sent back to fgets and prompts the user for another input line. This repeats until we get something valid.

*If you're ever printing a buffer for debugging, put quotes around it to be able to see whitespace.
When we check the value of sscanf, we originally had:
`errExit("Only received %d valid values. Found: %s\n, iScanfCount, szInputBuffer);`
Change it to put quotes around the format specifier -> '%s'\n

4. Rest of program compares the grades and prints alongside the abc123

Why do we use fgets and then sscanf instead of scanf?
With scanf when an error is encountered, we can't tell scanf to ignore the rest of the line and advance to the next line. Instead it has to keep reading and tripping over the remainder of that bad input.
With fgets when an error is encountered in sscanf (such as when sscanf doesn't return 3), we can simply do another fgets.

```
//when compiling include errExit: gcc -g example2-1.c errExit.c -o example2-1
$ ./example2-1
ABC123 Grade
abc123 56.8 98.9
abc123 98.90
xyz987 88.7 67.4
xyz987 88.70
```

code for errExit:

```
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#define ERROR_PROCESSING 99 //preprocessor directive, num doesn't
matter
/******** errExit ************************************************
  void errExit(const char szFmt[]. ...)
Purpose:
      Prints an error message defined by the printf-like szFmt and the
      corresponding arguments to that function. The number of
      arguments after szFmt varies dependent on the format codes in
      szFmt.
      It also exits the program.
Parameters:
      const char szFmt[]    This contains the message to be printed and
                            format codes (just like printf) for values
                            that we want to print.
      ...                   A variable-number of additional arguments
                            which correspond to what is needed by the
                            format codes in szFmt.
Returns:
      Exits the program with a return code of ERROR_PROCESSING (99).
Notes:
      - Requires including <stdarg.h>
*****************************************************************/

void errExit(const char szFmt[]. ...) {

      va_list args; //declare variable argument list
      va_start(args, szFmt); //va_start() to retrieve arguments
      fprintf(stderr, "ERROR: ");
      vfprintf(stderr, szFmt, args);
      va_end(args); //once all arguments are processed, let C know
                    //we're done

fprintf(stderr, "\n");
exit(ERROR_PROCESSING);
}
```

Buffered Output

Buffered output to the terminal acts differently from output to a file. When writing to the terminal, the output is sent almost immediately instead of buffering it. There are three types of buffering available with libc:

`line-based buffering` - buffers everything up to where a full line has been constructed. So once the buffer receives a newline character, the buffer is flushed. (most common)

`block-based buffering` - Used when you're dealing with a file. This could be a file where you call fopen on, or it could be a file caused by taking a regular program and redirecting, say, stdout to a file like:

```
~/stderr > out.txt
```

"So at this point, yes, stdout is redirected to a file, but by virtue of the fact that the stdout file stream is associated with the file, now its block-based buffering. So it doesn't care about newline characters any longer, it just buffers anything. Anything you write to this will be buffered. What happens next? You could just explicitly flush the buffer:

```
fflush(stdout);
```

More implicit means at which the buffer gets flushed is when the buffer reaches capacity, or fclose the filestream, that'll flush it. Also exiting the process abnormally will flush the buffer. So if you return from main or call exit, it'll cause any buffers containing outstanding data to flush."

`Third type is none, unbuffered.`

When directing output to a file, standard stream output will write to an internal buffer for efficiency. It won't actually write it to the file until there is enough data for a data block, it is asked to flush or the file is closed. Again similar to what was mentioned above, the buffer won't flush until it reaches capacity, is told to flush, or exits abnormally.

Note that the tail command in Linux prints the last portion of a file. Its most useful option is the `-f` switch which follows a stream output file. This is useful to monitor the output of a file while it is being produced (.e.g, watching a log file). `tail` will keep looking at the contents until you *CTRL-C* it.

Example:

```
$ > apache.log
$ tail -f apache.log #watching it for changes on the file

#At this point the professor goes to another server to generate
#changes to the apache.log file

$ echo "debug line 1" >> apache.log
$ for i in {2..10}; do echo "debug line $i" >> apache.log; done
#returning to where we're watching the file, the output is now
debug line 1
debug line 2
#and so on until line 10
```

Line based buffering:

Example 2-2:

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

int main(int argc, char ** argv) {
      printf("Step 1\n");
      sleep(5);
      printf("Step 2\n");
      return 0;
}
```

Output:

```
Step 1
Step 2
```

This is going to print a line, sleep for 5 seconds, then print another line. This will output to the terminal.
The terminal, because it's line-oriented, when you're tied to the terminal, you're doing line-based buffering. As I write these bytes to the internal buffer, because it's stdout and going to the terminal, it's line-based buffering. This goes to the buffer, unless/until I reach a newline.
With printf("Step 1\n"), if the newline character was not there, this would not print.
So given printf("Step 1, ")
When running this, we get nothing for 5 seconds. After 5 seconds, the output is:

```
Step 1, Step 2
```

If you took both newline characters away, it would flush anyways because immediately after the call to printf we are exiting normally via 'return 0', that will cause any buffer data still in the buffer, to flush.

Although it all looks the same, all these changes behave differently.

If we took both newline characters out, the output becomes:

```
Step 1, Step2fox01:~/cs3423/IO$
```

Let's say we have:

```
char name[64];
printf("Name: ");
scanf("%s", name);
printf("You entered: '%s'\n", name);
return 0;
```

scanf is going to block and wait, and it'll be waiting for me to type a name. Given what was just explained above, you would expect not to see "Name: " right after running the program. Yet this is our output:

```
$ ./well.c
Name:
```

Well the thing is, something did trigger it to flush. When you read from stdin, the c-standard library is smart enough to figure that if you're reading from stdin, there's a good chance you wrote a prompt to stdout. So inside scanf here, baked into this implementation, is a flush.

If you comment out the scan and replace it with 'sleep(10);'

After 10 seconds we get the output:

```
Name: You entered: '▒R@'
```

That's why we need to initialize variables in C as well, else they're filled with random garbage.

Block based buffering:

It's basically dealing with a buffer full of bytes. It's not looking for any specific character like a newline.

```
$ fopen("myfile", "w") #we can just do a file opener
```

Since FILE * file is associated with a file on the disk, it would use block based buffering.

We can also induce that ourselves by redirecting stdout to a file

```
$ ./well > out.txt
```

So if the target of the stream is a file, it'll use block based suffering. Using the original example 2-2:

```
$ ./well
Step 1
Step 2
$ ./well > out.txt
```

If we do it to a file, we can't actually see it until it's actually done. So we're going to redirect this to a file, run it in the background via '&', then simultaneously open that output file using 'tail -f' for follow mode.

```
$ ./well > out.txt & tail -f out.txt
[1] 22488
#5 seconds later
Step 1
Step 2
#you'll need to do CTRL-C to kill this
```

Despite the newlines in the print functions, it still flushed because the program terminated normally.

Fun facts:

setvbuf can be used to change the buffering; It's mode argument can be either:

_IONBF unbuffered

_IOLBF line buffered

_IOFBF fully buffered (block-based)

No buffering:

Recall:

```
$ ccat ~/stderr.c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char ** argv) {
      printf("This is standard output\n");
      fprintf(stderr, "This is standard error\n);
      return EXIT_SUCCESS;
}

#One line goes to stdout, the other goes to stderr.
```

```
$ ~/stderr
This is standard output
This is standard error
```
Professor: At the time I think I pitched the idea behind this as well this is useful because you can bifurcate the output stream such that if I want the errors to go to their own file I can:

```
$ ~/stderr 2> errors.txt
This is standard output
```

Versus if I want the errors to be on screen, but the normal output to go to a file:
```
$ ~/stderr > errors.txt
This is standard error
```

Or of course I can make both streams go to a file:
```
$ ~/stderr &> all.txt

$ cat errors.txt
This is standard error
$ cat output.txt
This is standard output
$ cat all.txt
This is standard error
This is standard output
```

There is another benefit to using stdout which is when you write to stderr, assuming it's going to the terminal (if it's going to a file it's using block-based buffering) stderr is unbuffered. That's important because, especially when you're doing debugging, If I used printf to print an error message- What if I do printf, then my program crashes, and the buffer is never flushed. So anything I was writing there never made it to the terminal. But as with stderr, the moment the call to fprintf happens, that string/data is on your screen. It could crash immediately after that call, but the end user/yourself can still see the error message.

Summary:
stderr -> unbuffered for terminal
stdout -> line-buffered for terminal
anything writing to file -> block-based buffering

```
fprintf(stderr, "Step 1, ");
$ ./well
Step 1, (waits 6 seconds) Step 2
#Because stderr is unbuffered, it went immediately to the screen.
```

binary i/o

One of the benefits of C is that it provides the ability to read and write structures. Since the data in a structure isn't just text (it could be doubles and/or integers), the files contain binary (i.e., non-printable) data. You cannot easily edit binary data with a text editor.

Instead of using scanf and printf, we use fread() and fwrite(), which do not support format codes (because we're not dealing with text).

```
typedef struct
{
   int iExemptionCnt;        // Number of exemptions
   char cFillingStatus;      // M - Married, S - Single
                            // X - married but filing as single
   double dWithholdExtra;   // extra amount to withhold
} W4;

typedef struct
{
    char szEmployeeId [10];
    char szFullName[40];
    double dHourlyRate;
    W4 w4;          //W4 struct that is defined above
} Employee;

Employee employee;
```

fopen

```
FILE *fopen (const char *pszFilename, const char *szMode)
```

- opens the specified file and returns a FILE pointer or NULL if it couldn't be opened
- The filename is the name of the file, possibly qualified with a path (e.g., "~/cs3423/employeeData.dat")
- mode is a string which tells fopen how to open the specified file

mode	description	starts at
r	• open for reading as text stream • if file doesn't exist, fopen() returns NULL	beginning
w	• open for writing as text stream • creates file if it doesn't exist • overwrites content	beginning
a	• open for writing as text stream • creates file if doesn't exist	end
r+	• open for reading and writing as text stream • if file doesn't exist, fopen() returns NULL	beginning
w+	• open for reading and writing as text stream • creates file if doesn't exist • overwrites contents	beginning
a+	• open for reading and writing as text stream • creates file if doesn't exist	end
rb	• open for reading as binary • if file doesn't exist, fopen() returns NULL	beginning
wb	• open for writing as binary • creates file if doesn't exist • overwrites contents	beginning
ab	• open for writing as binary • creates file if doesn't exist	end
rb+	• open for reading and writing as binary • if file doesn't exist, fopen() returns NULL	beginning
wb+	• open for reading and writing as binary • creates file if doesn't exist • overwrites contents	beginning
ab+	• open for reading and writing as binary • creates file if doesn't exist	end

- '+' open for [mode] and writing
- 'b' open for [mode] and binary
- 'r' returns NULL in any combo if file doesn't exist
- 'a' starts at end (appends) for any combo

'w' is very similar to the output redirect symbol in bash '>'

```
$ cat words.txt
one
two
and so on until ten…
$ echo "test" > words.txt
$ cat words.txt
test
```

With 'a+':
With append, no matter where you seek in the file (you can seek to whatever position you want), if I attempt to write something there, the OS automatically resets the cursor position to the EOF.
Even if you seek somewhere else, you can only write to the end. With that said, you can read wherever you want.

ASCII/UTF-8: character encoding
mapping from a byte to a character

0011 0001 -> '1'

Linux/Unix/BSD/etc. -> LF ('\n') -> 0x0A (byte value)
MacOS >=X -> LF ('\n') -> 0x0A
MacOS <X -> CR ('\r') -> 0x0D
DOS/Windows -> CRLF ('\r\n') -> 0x0D 0x0A
When you're reading and writing in binary mode, it doesn't matter what the bytes are. This is just a buffer full of bytes/it's treated as a stream of bytes.
In the text mode, conversions can take place.
Text or Automatic -> transfer a text file -> convert the line endings from the remove host to the local host

e.g. If I was uploading a Windows file to Linux, it'd change the CRLF to just LF.
- unix2dos can convert to windows format vice versa

Example 2: use fopen to open a file for binary input and another for binary output

```
//FILE -> file struct
//pfileEmployee -> pointer var that holds the address of the file stream
FILE * pfileEmployee;
#define EMPLOYEE_FILE "employeeData.dat"

// Open binary file for read
//fopen() opens a file stream, 1st argument is employeeData.dat, 2nd is rb
//rb -> read mode, binary mode
pfileEmployee = fopen(EMPLOYEE_FILE, "rb");
//checks if file opening successful, errExit() if failed
if (pfileEmployee == NULL)
    errExit("could not find employee file: '%s'\n", EMPLOYEE_FILE);
//another file pointer for output, used for writing to new file
FILE *pfileNewEmployee;
#define NEW_EMPLOYEE_FILE "employeeDataNew.dat"

// Open binary file for write
pfileNewEmployee = fopen(NEW_EMPLOYEE_FILE, "wb");
//error handling
if (pfileNewEmployee == NULL)
    errExit("could not open new employee file: '%s'\n", NEW_EMPLOYEE_FILE);
```

fread

```
long fread (void *psbBuf, long lSizeOneRec, long lNumberOfRec, FILE *pFile)
```

- Uses a record oriented system, treats everything as a sequence of a record.

```
void *psbBuf -> Doesn't specify its own limit so you need to keep
                track to not overflow the buffer.
lSizeOneRec  -> The size of a record.
lNumberOfRec -> Number of records you'd like to read into the buffer.
```

- A void * pointer gets its data type from the invoking parameter, but it must be an address.
- Beginning with the current file position, fread reads >=1 binary records into the specified buffer which is typically a binary structure. Returns the number of records successfully read as its functional value. If one record is read successfully, 1 is returned. Returns 0 if none are read.

Example 2-4: use fread to read one employee record

```
//&employee -> address to store record data
//sizeof(Employee) -> size of one record
//1L -> number of records as a long
iNumRecRead = fread (&employee, sizeof(Employee), 1L, pfileEmployee);

if (iNumRecRead != 1) {
     errExit("Expected to get an Employee record, %s", "but fread returned no records");
}
```

Both fread/fwrite have a return value of number of records read.

fwrite

```
long fwrite (void *psbBuf, long lSizeOneRec, long lNumberOfRec, FILE *pFile)
```

Same thing as fread for all parameters, but with writing to a file.
Although both are shown as type 'long', the manual lists them as size_t (still 8 byte or 64bit number). Any time you have a numerical literal in your code, it's assumed to be an integer. When it compiles into assembly it's gonna be treated as a 32-bit integer, but depending on how your compiler is configured you may get a warning about a width issue between those two types because I'm giving a 4 byte/32 bit value whereas a 64 bit one is expected.

It's just going to sign extend this up to 8 bytes, but to avoid that- you can put an L here that states this literal is a long. 1UL is an unsigned long.
Example 2-4: read and write each binary employee record.

```
int iRecCount = 0;
// copy the employee file to create a new one
while (fread (&employee, sizeof(Employee), 1L, pfileEmployee)==1) {
      iRecCount++;
      rc = fwrite(&employee, sizeof(Employee), 1L, pfileNewEmployee);
      if (rc != 1) {
        errExit("Error writing record %d to the new employee file", iRecCount);
      }
fclose(pileEmployee);
fclose(pfileNewEmployee);
#condition of loop -> call fread
#Does the file copy one record at a time
#Just use copy command to copy employeeData, not applicable in real world
```

Although fread is fairly safe, you can overwrite memory if you specify the wrong sizeOfRecord or lNumber of Records.

fclose

```
void fclose (FILE *pFile)
```

Closes the specified file, completing the I/O and freeing internal memory used for the file.

Sequential Files & Direct Access Files

We're used to dealing with direct access files/random access.
The files contents are one long sequence of the same struct, for example: If I want to access the nth employee record I'd just do

```
(n-1)*sizeof(Employee)
```

You can just fseek to this location:

```
fseek(myfile, (n-1)*sizeof(Employee), SEEK_SET);
```

This would put the cursor position at the very first one, or if set at 100 -> going to be at 99th employee.

A sequential file contains multiple records which are arranged sequentially (i.e., one record after the other).

They're best with the data: ASCII text files, "memory dumps", and stream data. When they read data from a file in a sequential fashion:
- No direct access
- Very difficult to insert a new record anywhere other than at the end of the file.
- Updates of a record are fairly easy if the new record is of the same size.
- Many conventional application systems which use sequential files have a Master File and a Differential File (describes changes to the master).
- A Master File contains the latest Master copy of the data sorted by a key.

Basically:
Sequential-access files -> you have to read or write the file in the order of the data.

A direct access file is defined to allow access at particular positions directly (without having to read the file sequentially).

fseek can be used to set the current position based on a relative byte offset.
fread or fwrite can then be used to read or write the data at that position.
Sequential access is also provided by simply using fread or fwrite from the beginning of the file.

b-tree (balanced tree not binary tree) and hashed files are frequently stored/manipulated as direct access files.

A sequential file is like a cassette, and a random-access file is like a DVD/Blu-ray.

Example: Opening a file to write to

```
#include <stdio.h>
main() {
  FILE *fptr; //defines a file pointer
  fptr = fopen("c:\cprograms/cdata.txt", "w"); //writing to file
  //rest of code
  fclose(fptr); //always close files you open
}
```

Example: Sequential Files

```
/* program takes book info and writes said info to a file named
   bookinfo.txt */
#include "bookinfo.h"
#include <stdio.h>
#include <stdlib.h>

FILE * fptr;

main() {
  int ctr;
  struct bookInfo books[3]; //array of 3 struct variables

  //get info about each book from user
  for (ctr = 0; ctr < 3; ctr++) {
    printf("What's the name of the book #%d?\n", (ctr+1));
    gets(books[ctr].title);

    puts("Who is the author? ");
    gets(books[ctr].author);

    puts("How much did the book cost? ");
    scanf(" $%f", &books[ctr].price);

    puts("How many pages in the book?");
    scanf(" %d", &books[ctr].pages);
    getchar(); //clears last newline for next loop
   }

 if ((fptr = fopen("/tmp/BookInfo.txt","w")) == 0) {
       printf("Error: File couldn't be opened.\n");
       exit(1);
 }
 else {
   fprintf(fptr, "\n\n Book collection: \n");
   for (ctr = 0; ctr < 3; ctr++) {
       fprintf(fptr, "#%d: %s by %s\n", (ctr+1), books[ctr].title, books[ctr].author);
       fprintf(fptr, "\n Pages: %d, Cost: $%.2f", books[ctr].pages, books[ctr].price);
 }
 fclose(fptr);

return 0;
}
```

<header><payload>
header of a known size, variable of a payload size
in this, payload could be another header
<ip header>tcp header> <tcp payload><ip payload> etc...
If I wanna access the 100th packet, I don't know the relative byte address.
Every packet is a different size. But the header is the same size and
contains the length of the payload, so I have to read the header first.
So I have to:
read header first -> fseek past specified payload length -> tells me the size
of the payload -> seek past that then after that the next header -> next
payload, and so on.

Direct Positioning in a File

```
      int fseek (FILE *pFile, long lRelativeByteAddress, int SeekMode)
//Sets the position in the file based on an offset and iSeekMode.
//Returns 0 if successful.
```

Values of SeekMode:

SEEK_SET -> Set the position relative to the beginning of the file
The byte address specified is relative to the file.

```
      fseek(myfile, 10, SEEK_SET)
    //places cursor 10 bytes into file from the beginning
```

SEEK_CUR -> Advances cursor 10 more bytes into the file wherever it is.

SEEK_END -> Places cursor 10 bytes past end of file

When using SEEK_CUR and SEEK_END, negative values for lRelativeByteAddress are allowed. You can do -10 to go back 10 bytes.

If you fseek past the end of the file, it is not an error.
It's legal to seek past the end of the file. The "hole" is backfilled with zeros.

->writing is legal past the end of the file

if you seek far enough that there's empty space, the hole is filled with zeros.
fwrite - it will write a record at that location. It also pads with records containing all zeros up to that new record.

->reading is not legal past the end of the file

illegal to seek prior to the beginning of a file.
fread - it will not find a record
- lRelativeByteAddress is a byte offset (relative to zero).
- pFIle is a FILE pointer returned by fopen.

Important: If the file is open in append mode, the position will be reset to the end of the file before every write.

Example: direct.c

```
FILE *pFileDirect;
FILE *pFileInputTxt;
void processCommandSwitches(int argc, char *argv[], char **ppszDirectFileName,
                            char **ppszInputTxtFileName);
void printFile(char *pszDirectFileName, char szTitle[]);
void processInputCommands(char *pszInputTxtFileName, char *pszNewDirectFileName);

int main(int argc, char *argv[]) {
      char * pszDirectFileName = NULL;
      char * pszInputTxtFileName = NULL;
      int rc;
      int iCommandType;

      processCommandSwitches(argc, argv, &pszDirectFileName, &pszInputTxtFileName);
      processInputCommands(pszInputTxtFileName, pszDirectFileName);
      return 0;
}

void processInputCommands(char *pszInputTxtFileName, char *pszDirectFileName) {
      Employee employee;  //employee data
      char szInputBuffer[100];
      char cCommand;
      long lRBA, lRecNum;
      char szRemaining[100];
      int iScanfCnt, rcFseek, rc;
      int iWriteNew; // 1 - wrote 1 record, 0 - error

//open the txt file for read
if ((pFileInputTxt = fopen(pszInputTxtFileName, "r") == NULL) errExit(ERR_INPUT_TXT_FILENAME,
pszInputTxtFileName);

 //open the New Direct data file for write binary. If it already exists, we simply update it
if ((pFileDirect = fopen(pszDirectFileName, "wb+") == NULL) errExit(ERR_UPDATE_FILENAME,
pszDirectFileName);
//what does wb+ mean: if it exists we open for read/write and delete the contents
//if it doesn't exist, we create it and open it for read/write, "binaryreadwrite"
 /* get commands until EOF fgets returns null when EOF is reached. */
while (fgets(szInputBuffer, 100, pFileInputTxt) != NULL) {
        if (szInputBuffer[0] == '\n') continue;
      printf("> %s", szInputBuffer);
      iScanfCnt = sscanf(szInputBuffer, "%c %ld %99[^\n]\n", &lRecNum, szRemaining);
      if (iScanfCnt < 2) {
            printf("Error: Expected command and RBA, found: %s\n", szInputBuffer);
            continue;
        }
      switch(cCommand) {
            case 'W':
            iScanfCnt = sscanf(szRemaining, "%6s %lf %d %c %lf %40[^\n]\n",
            employee.szEmployeeId, &employee.dHourlyRate, &employee.w4.iExemptionCnt,
            &employee.w4.cFillingStatus, &employee.w4.dWithholdExtra, employee.szFullName);

//Check for bad input.mscanf returns the number of valid conversions
            if (iScanfCnt < 6) {
```

```
                errExit(ERR_INVALID_EMPLOYEE_DATA, szInputBuffer);
                lRBA = lRecNum*sizeof(Employee);
                rcFseek = fseek(pFileDirect, lRBA, SEEK_SET);
                assert(rcFseek == 0);
               //write it to the direct file
                iWriteNew = fwrite(&employee, sizeof(Employee), 1L, pFileDirect);
                assert(iWriteNew == 1);
                break;
               case 'R':
                lRBA = lRecNum*sizeof(Employee);
                rcFseek = fseek(pFileDirect, lRBA, SEEK_SET);
                assert(rcFseek == 0);
               // print the information at the RBA
                rc = fread(&employee, sizeof(Employee), 1L, pFileDirect);
               if (rc == 1) {
               //we didnt do assert b/c we dont want it to die if the return
               //value isn't 1, just print an error message. but if it is 1 we print it out
                printf("%-7s  %8.2lf  %5d %c %8.2lf %-40s\n", employee.szEmployeeId,
                employee.dHourlyRate, employee.w4.iExemptionCnt,
                employee.w4.cFillingStatus, employee.w4.dWithholdExtra,
                employee.szFullName);
                }
               else {
               printf("Record number %ld not found for RBA %ld\n", lRecNum, lRB  A);
               break;
               }
               default:
               printf("unknown command\n");
          }
}
// close the files
fclose(pFileInputTxt);
fclose(pFileDirect);
- - -
void printFile(char *pszDirectFileName, char szTitle[]) {
       FILE *pfileEmployee;
       Employee employee;
       printf("%s for file %s\n%-7s  %8s  %6s  %6s  %10s  %-40s\n", szTitle, pszDirectFileName,
       "Employee", "Pay Rate", "Exempt", "Status", "With Extra", "Full Name");
       //open the file
       pfileEmployee = fopen(pszDirectFileName, "rb");
     if (pfileEmployee == NULL) {
       errExit(ERR_MASTER_FILENAME, pszDirectFileName);
       //read the Employee data until EOF
       while (fread(&employee, sizeof(Employee), 1L, pfileEmployee) == 1) {
        printf("%-7s  %8.2lf  %5d  %c %8.2lf %-40s\n", employee.szEmployeeId,
        employee.dHourlyRate, employee.w4.iExemptionCnt, employee.w4.cFillingStatus,
        employee.w4.dWithholdExtra, employee.szFullName);
         }
fclose(pfileEmployee);
}
```

```
void processCommandSwitches(int argc, char *argv[], char **ppszDirectFileName, char
**ppszInputTxtFileName) {
       int i;
       //If there aren't any arguments, show the usage
       if (argc <= 1) exitUsage(0, "No arguments", "");

       //Examine each of the command arguments other than the name of the program.
       for (i = 1; i < argc; i++) {
       //check for a switch
        if (argv[i][0] != '-') exitUsage(i, ERR_EXPECTED_SWITCH, argv[i]);
       //determine which switch it is
       switch (argv[i][1]) {
              case '?':
               exitUsage(USAGE_ONLY, "", "");
               break;
              case 'o':    // Direct File Name
               if (++i >= argc) //check for too long of a file name
                exitUsage(i, ERR_MISSING_ARGUMENT, argv[i - 1]);
               else
                *ppszDirectFileName = argv[i];
                 break;
              default:
                 exitUsage(i, ERR_EXPECTED_SWITCH, argv[i]);
       }
    }
}
- - -
void errExit(const char szFmt[], ... ) {
  va_list args;            //This is the standard C variable argument list type
  va_start(args, szFmt);   //Tells the compiler where the variable arguments begin.
  printf("ERROR: ");       //They begin after szFmt.
  vprintf(szFmt, args);    //vprintf receives a printf format string and a
                           //va_list argument
  va_end(args);           //let the C environment know we are finished with the
                          //va_list argument
  printf("\n");
  exit(ERROR_PROCESSING);
}
- - -
void exitUsage(int iArg, char *pszMessage, char *pszDiagnosticInfo) {
       if (iArg > 1)
       fprintf(stderr, "Error: bad argument #%d.  %s %s\n", iArg, pszMessage,
       pszDiagnosticInfo);
       fprintf(stderr, "direct -i inputTxtFile -o directFile\n");
       if (iArg >= 0)
              exit(ERR_COMMAND_LINE_SYNTAX);
       else
              exit(USAGE_ONLY);
}
```

```
first do write record 1:
> W 1 11111 10.00 1 S 0.00 Highwater, Helen
> R 1
11111 10.00 1 S 0.00 Highwater, Helen
> W 2 22222 12.00 1 S 20.00 Flood, T. Rential
> W 3 33333 25.00 2 M 100.00 Tall, Jerry
> W 4 44444 40.00 1 S 200.00 Yuss, Jean E.
> W 5 55555 8.00 1 X 0.00 Absent, Marcus
> R 4
44444 40.00 1 S 200.00 Yuss, Jean E.
> W 8 88888 88.00 1 S 0.00 Moss, Pete
> R 8
88888 88.00 1 S 0.00 Moss, Pete
> R 7
0.00 0 0.00
> R 9
#internally this program is using a 0 based record number
#when we use 1 we're already skipping the very first record ">W1 1111…"
#this will have done an fseek to 1 * sizeofemployee
> W 8 #jumping over 6 and 7 to write to 8
> R 8
#read to 8 and we get no error
>R 7
0.00 0 0.00
#doesnt produce an error; hole is filled with zeroes. the floats, integers, etc are
#filled with 0. theyre all filled with null/0 bytes /null string
> R 9
#will give record num not found error
#1 byte past EOF
#writing is fine, but when we read it it fails
> R 50
record num not found
#R 0 would be filled with zeroes bc we skipped over it, so the hole is filled w/ 0
```

Notice that record number 0 and 7 weren't written, but they are all zero. When reading record numbers 9 and 50, we get a not found error since those are past the last written record (record number 8).
Remember: sscanf family return value is equal to successful conversions its made
We ask for 6 and if we don't, we err out and exit, otherwise: fseek returns 0 on success!!

Compiling and Executing direct.c

```
$ gcc -o direct direct.c
$ ./direct -i inputDirect.txt -o direct.dat
#the generated output was shown earlier
```

empMerge.c example:

```
#define _CRT_SECURE_NO_WARNINGS 1
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <stdlib.h>
#include <stdarg.h>
#include "empMerge.h"

/* files */
FILE * pfileMaster;
FILE * pfileUpdate;
FILE * pfileNewMaster;
FILE * pfileInputTxt;
/*prototypes*/
void processCommandSwitches(int argc, char * argv[], char **ppszMasterFileName,
char **ppszUpdateFileName, char **ppszNewMasterFileName, char **ppszInputTxtFileName,
int * piComandType);

void merge(char * pszMasterFileName, char * pszUpdateFileName, char * pszNewMasterFileName);
void printFile(char * pszMasterFileName, char szTitle[]);
void readTxtWriteMaster(char * pszInputTxtFileName, char * pszNewMasterFileName);

int main(int argc, char *argv[]) {
	char *pszMasterFileName     = NULL;
	char * pszUpdateFileName    = NULL;
	char * pszNewMasterFileName = NULL;
	char * pszInputTxtFileName  = NULL;
	int rc;
	int iCommandtype;
	/*get the filenames and command type from the command switches*/
	processCommandSwitches(argc, argv, &pszMasterFileName, &pszUpdateFileName,
			&pszNewMasterFileName, &pszInputTxtFileName, &iCommandType);
	switch(iCommandType) {
		case COMMAND_MERGE: //merge 2 files producing new master
		  if(pszMasterFileName == NULL) errExit(ERR_MISSING_SWITCH,"-m");
		  if(pszUpdateFileName == NULL) errExit(ERR_MISSING_SWITCH,"-u");
		  if(pszNewMasterFileName == NULL) errExit(ERR_MISSING_SWITCH,"-n");
		  printFile(pszMasterFileName, "Master File");
		  printFile(pszMasterFileName, "Updates File");
		printFile(pszNeMasterFilename, "Results for Merge");
		break;
		case COMMAND_CREATE: //create a master from txt
		  if(pszNewMasterFileName == NULL) errExit(ERR_MISSING_SWITCH, "-n");
		  if(pszInputTxtFilename == NULL) errExit(ERR_MISSING_SWITCH, "-i");
		readTxtWriteMaster(pszInputTxtFileName, pszNewMasterFileName);
		printFile(psznewmasterFileName, "Results for Create");
		break;

		case COMMAND_PRINT: //only print a data file
		  if(pszMasterFileName == NULL) errExit(ERR_MISSING_SWITCH, "-p");
		printFile(pszMasterFileName, "Results for -p Print");
		break;
	}
	return 0;
}
```

```
void merge(char * pszMasterFileName, char * pszUpdateFileName, char * pszNewMasterFileName) {
       Employee employeeMaster;
       Employee employeeUpdate;
       int iGotMaster;
       int iWriteNew;

       pfileMaster = fopen(pszMasterFileName, "rb"); //open master
       if (pfileMaster == NULL) errExit(ERR_MASTER_FILENAME, pszMasterFileName);
        pfileMaster = fopen(pszUpdatefileName, "rb"); //open the update data file
       if (pfileMaster == NULL) errExit(ERR_MASTER_FILENAME, pszUpdateFileName);
        pfileMaster = fopen(pszNewMasterFileName, "wb"); //open the new master file
       if (pfileMaster == NULL) errExit(ERR_MASTER_FILENAME, pszNewMasterFileName);

       //read the first record from each file
       iGotMaster = fread(&employeeMaster, sizeof(Employee), 1L, pfileMaster);
       iGotUpdate = fread(&employeeUpdate, sizeof(Employee), 1L, pfileUpdate);

       //continue in loop until one of the files reaches EOF
   while (iGotMaster == 1 && iGotUpdate == 1) {
     if (strcmp(employeeMaster.szEmployeeId, employeeUpdate.szEmployeeId) < 0) {
        iWriteNew = fwrite(&employeeMaster, sizeof(Employee), 1L, pfileNewMaster);
        assert(iWriteNew == 1);
        iGotMaster = fread(&employeeMaster, sizeof(Employee), 1L, pfileMaster);
     }
     else if (strcmp(employeeMaster.szEmployeeId, employeeUpdate.szEmployeeId) > 0) {
        iWriteNew = fwrite(&employeeUpdate, sizeof(Employee), 1L, pfileNewMaster);
        assert(iWriteNew == 1);
        iGotUpdate = fread(&employeeUpdate, sizeof(Employee), 1L, pfileUpdate);
     }
     else {
       iWriteNew = fwrite(&employeeUpdate, sizeof(Employee), 1L, pfileNewMaster);
       assert(iWriteNew == 1);
       iGotUpdate = fread(&employeeUpdate, sizeof(Employee), 1L, pfileUpdate);
       iGotMaster = fread(&employeeMaster, sizeof(Employee), 1L, pfileMaster);
     }
  }

 while (iGotMaster == 1) {
       iWriteNew = fwrite(&employeeMaster, sizeof(Employee), 1L, pfileNewMaster);
       assert(iWriteNew == 1);
       iGotMaster = fread(&employeeMaster, sizeof(Employee), 1L, pfileMaster);
 }

while (iGotUpdate == 1) {
       iWriteNew = fwrite(&employeeUpdate, sizeof(Employee), 1L, pfileNewMaster);
       assert(iWriteNew == 1);
       iGotUpdate = fread(&employeeUpdate, sizeof(Employee, 1L, pfileUpdate);
 }

fclose(pfileMaster);
fclose(pfileUpdate);
fclose(pfileNewMaster);
}
```

```
void readTxtWriteMaster(char *pszInputTxtFileName, char *pszNewMasterFileName) {
   Employee employee;
   char szInputBuffer[100];
   char *pszGetsResult;
   int iScanfCnt;
   int iWriteNew;

  pfileInputTxt = fopen(pszInputTxtFileName, "r");
  if (pfileInputTxt == NULL) errExit(ERR_INPUT_TXT_FILENAME, pszInputTxtFileName);
  pfileNewMaster = fopen(pszNewMasterFileName, "wb");
  if (pfileNewMaster == NULL) errExit(ERR_UPDATE_FILENAME, pszNewMasterFileName);

   pszGetsResult = fgets(szInputBuffer, 100, pfileInputTxt);
  while (pszGetsResult != NULL) {
   iScanfCnt = sscanf(szInputBuffer, "%6s %lf %d %c %lf %40[^\n]\n",
   employee.szEmployeeId, &employee.dHourlyRate, &employee.w4.iExemptionCnt,
   &employee.w4.cFillingStatus, &employee.w4.dWithholdExtra, employee.szFullName);

  if (iScanfCnt < 6) errExit(ERR_INVALID_EMPLOYEE_DATA, szInputBuffer);

  iWriteNew = fwrite(&employee, sizeof(Employee), 1L, pfileNewMaster);
  assert(iWriteNew == 1);
  pszGetsResult = fgets(szInputBuffer, 100, pfileInputTxt);
 }
 fclose(pfileInputTxt);
 fclose(pfileNewMaster);
}

void printFile(char *pszMasterFileName, char szTitle[]) {
       FILE * pfileEmployee;
       Employee employee;
       printf("%s for file %s\n%-7s %8s %6s %10s %-40s\n", szTitle, pszMasterFileName,
              "Employee", "Pay Rate", "Exempt", "Status", "With Extra", "Full Name");

       pfileEmployee = fopen(pszMasterFileName, "rb");
       if (pfileEmployee == NULL) errExit(ERR_MASTER_FILENAME, pszMasterFileName);

       while (fread(&employee, sizeof(Employee), 1L, pfileEmployee) == 1) {
         printf("%-7s %8.2lf %5d %c %8.2lf %-40s\n", employee.szEmployeeId,
                employee.dHourlyRate, employee.w4.iExemptionCnt,
                employee.w4.cFillingStatus, employee.w4.dWithholdExtra,
                char **ppszUpdateFileName, char ** ppszNewMasterFileName,
                char ** ppszInputTxtFileName, int * piComandType);
 }
```

```
void processCommandSwitches(int argc, char *argv[], char ** ppszMasterFileName,
              char ** ppszUpdateFileName, char ** ppszNewMasterFileName,
              char ** ppszInputTxtFileName, int * piComandType) {
       int i;
       *piComandType = COMMAND_MERGE;
       if (argc <= 1) exitUsage(0, "No arguments", "");

       for (i = 1; i < argc; i++)
              if (argv[i][0] != '-') exitUsage(i, ERR_EXPECTED_SWITCH, argv[i]);
switch (argv[i][1]) {
       case 'm':  // Master File Name
       case 'p':
        if (++i >= argc)
        exitUsage(i, ERR_MISSING_ARGUMENT, argv[i - 1]);
        //check for too long of a filename
        else
        *ppszMasterFileName = argv[i];
         if (argv[i - 1][1] == 'p') *piComandType = COMMAND_PRINT;
       break;
       case 'u': // Update File Name
         if (++i >= argc)
         exitUsage(i, ERR_MISSING_ARGUMENT, argv[i - 1]);
         else
         *ppszUpdateFileName = argv[i];
         break;
       case '?':
       exitUsage(USAGE_ONLY, "", "");
         break;
       case 'n':  // New Master File Name
         if (++i >= argc)
         exitUsage(i, ERR_MISSING_ARGUMENT, argv[i - 1]);
         // check for too long of a file anme
         else
         *ppszNewMasterFileName = argv[i];
         break;
       case 'i':   // Input Txt File Name
         if (++i >= argc)
         exitUsage(i, ERR_MISSING_ARGUMENT, argv[i - 1]);
         // check for too long of a file anme
         else
         *ppszInputTxtFileName = argv[i];
         *piComandType = COMMAND_CREATE;
         break;
       default:
         exitUsage(i, ERR_EXPECTED_SWITCH, argv[i]);
    }
  }
}
```

```
void exitUsage(int iArg, char *pszMessage, char *pszDiagnosticInfo)
{
  if (iArg > 1)
       fprintf(stderr, "Error: bad argument #%d.  %s %s\n", iArg, pszMessage,
               pszDiagnosticInfo);
       fprintf(stderr, "empMerge -m masterFile -u updateFile -n newMasterFile\n ");
       fprintf(stderr, "empMerge -i inputTxtFile -n newMasterFile\n");
       fprintf(stderr, "empMerge -p masterFile\n");
  if (iArg >= 0)
       exit(ERR_COMMAND_LINE_SYNTAX);
  else
       exit(USAGE_ONLY);
}
```