Chapter 6

Streams and File/Device I/O

What is in This Chapter ?

This chapter discusses **Streams** and how to do Input/Output (i.e., **I/O**) through these streams. It mainly focusses on how to **open/close/read/write** *files*, although the concepts carry over to other streams as well, such as reading and writing to various hardware devices. The difference between **binary-based** and **text-based** files is discussed along with examples. The idea of **Random Access files** is mentioned, along with a discussion of how to access/modify binary files quicker than **Sequential Access files**. There is a discussion of **buffers**, and how they relate to I/O in regard to flushing the data. Finally, a discussion of **pipes** is presented and how they can be used in the Linux shell to redirect program input and output.

6.1 Streams and File I/O

By now, you should know that **I/O** is short for **Input**/**Output**.

> ***Input*** *is the transferring of bytes from a data source (e.g., keyboard, mouse, game controllers, files, network, other programs) to a program.*

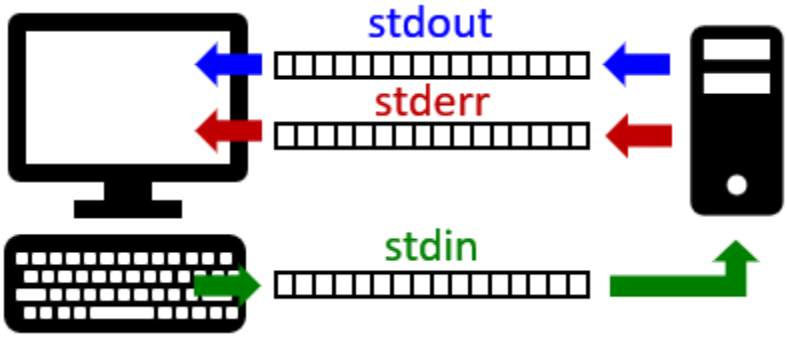
> ***Output*** *is the transferring of bytes from a program to a data sink (e.g., monitor, files, network, printer, other programs).*

In C, I/O operations are supported by standard library functions and system calls as well as Unix shell functions.

When I/O is described as the transferring of bytes, we tend to use the term ***stream***. You should all be familiar with streamed data since we see this often over the internet (e.g., Netflix is used to stream data to your home).

In C, there are standard streams set up already:

- **stdin** = standard input == 0 (default is keyboard)
- **stdout** = standard output == 1 (default is display)
- **stderr** = standard error == 2 (default is display)



Standard streams are defaulted to sources/sinks as shown above, but they can be ***re-directed*** to/from other programs, files, devices, data sources and sinks (more on this later).

Streams can contain data in two types of formats:

1. formatted **text** data
   - stored in ASCII text representation

2. unformatted **binary** data
   - stored in machine representation

There are advantages and disadvantages to both formats …

Text:

- ✓ Readable (we can easily read what is stored)
- ✓ Data can be recovered easily/naturally, no need to remember exact format
- ✓ High compression ratio (e.g., smaller when zipped)
- ✗ Can require more storage space
- ✗ Slow to access – sequential access only
- ✗ Every field of data must be written, cannot write **struct** in one shot

Binary:

- ✓ Fast to access – random access to data
- ✓ Can require less storage space
- ✓ Read/write whole **structs**
- ✗ Low compression ratio (e.g., larger when zipped)
- ✗ Hard to recover data, need to remember format it was written in

A stream ends with a special marker called an **end-of-file** (a.k.a. **EOF**) marker that follows the last byte of the file. It is similar to the idea of putting `'\0'` at the end of a string to identify the end of the string. In C, the end-of-file marker is pre-defined as **EOF** and it has a value of **-1** in our system (although this value is OS dependent). Some of the C library functions will return this end-of-file marker.

What are some of the stream-related library functions? The general stream-related system calls that we can use are not the same as library functions. They are operating-system-dependent … which means that if you use them, your code will not be as portable as you might like. Here they are:

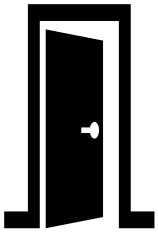
```
int      open(const char *path, int flags);
int      close(int fileDescriptor);
ssize_t  write(int fileDescriptor, const void *buffer, size_t count);
ssize_t  read(int fileDescriptor, void *buffer, size_t count);
```

These can be used to read in data from (and send data to) various I/O devices. So, as it turns out, we can read/write to various hardware devices in a similar manner to reading/writing files by using these more generalized functions. However, we will not discus these any further in this course.

Instead, we will discus the library functions that relate to **file streams** … which, for example, can be used to read/write files to/from your hard drive, USB drive, etc…

Before a stream (file or otherwise) can be used to send data, it must first be *opened*. Afterwards it must be *closed*.

The **`fopen()`** function is used to open a stream and it has this format:

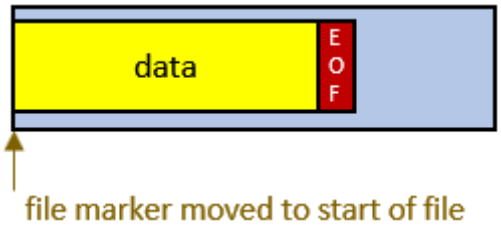


```
FILE *  fopen(const char *filename, const char *mode)
```

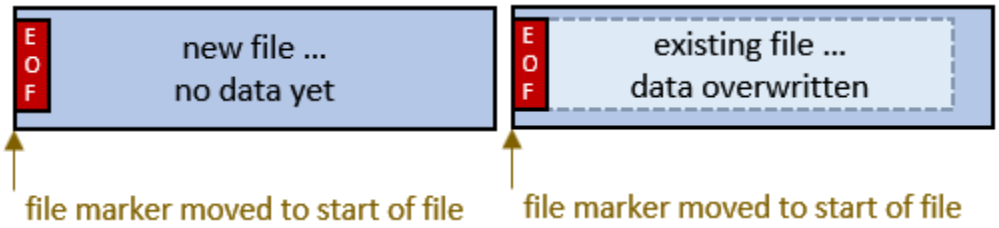
The first parameter is used to specify the name of the file to be opened.

The second parameter specifies the mode, which can be one of the following:

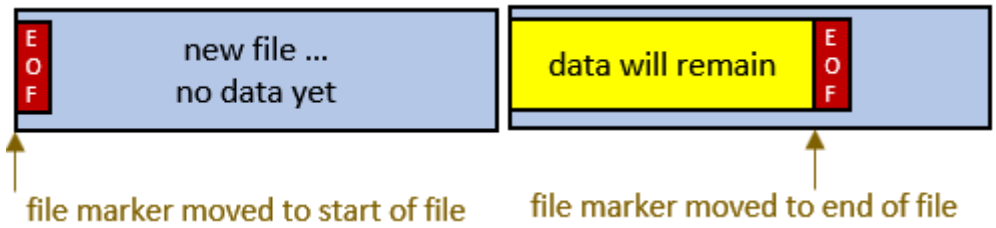
- **"r"** – for reading the file, which must already exist.



- **"w"** – for writing to a file. If the file does not yet exist, a new one with the given name is created. If the file already exists, it is completely overwritten (i.e., old file contents are erased !!).



- **"a"** – for appending to a file. If the file does not yet exist, a new one with the given name is created. If the file already exists, then any new information sent to the file stream is appended to the end (i.e., old file contents remains intact).



- **"r+"** – opens a file for reading & writing (file marker begins at start of file).
- **"w+"** – creates a new file for reading & writing (file marker begins at start of file).
- **"a+"** – opens a new file for reading & appending (file marker begins at end of file).

The function returns a **FILE *** which is a file handle/descriptor that uniquely identifies the file stream. It will return **0** (i.e., the **NULL** pointer) if an error occurred. There could be many reasons (at least 20) why a file could not be opened. You can see the documentation for details. Here is how to open a file for writing:

```
FILE  *fd;

fd = fopen("sampleFile.dat", "w");
if (!fd) {
   printf("ERROR: Could not open file\n");
   exit(1);
}
```

Here is slightly more compact way to do it:

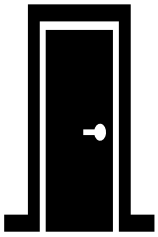
```
FILE  *fd;

if ((fd = fopen("sampleFile.dat", "w")) == NULL) {
   printf("ERROR: Could not open file\n");
   exit(1);
}
```

Often, we want to check for **NULL** when opening for reading. This will allow us to inform the user that the file does not exist and ask if the file should be created.

```
FILE  *fd;

if ((fd = fopen("sampleFile.dat", "r")) == NULL) {
   // file does not exist, maybe ask user if it should be created
}
else {
   // continue with the opened file and read it
}
```

Once the stream has been opened, it will later need to be closed using the **fclose()** function:

```
int  fclose(FILE *fd)
```



The function takes the **FILE *** file descriptor that was returned from the call to **fopen()**. It will close the file. If there were any contents in the buffer waiting to be sent to the file, it is flushed (i.e., written to the file) before the file is closed. If there is content in the input buffer that was not read in yet, it is discarded. The function returns **0** if all went well, and returns the **EOF** character if an error occurred (at least 10 kinds of errors … see documentation). Once closed, the stream can no longer be used.

Here is how to check for an error upon closing:

```
if (fclose(fd) == EOF) {
  printf("ERROR: Could not close file\n");
  exit(1);
}
```

There are some other useful functions that can be used when dealing with file errors as you read/write from/to a file:

- **ferror**(FILE *fd) - checks if the error flag is set for the file
- **clearerr**(FILE *fd) - clears the error flag associated with the file
- **feof**(FILE *fd) - checks if **EOF** was reached

For example, after trying to read to a file, we could do this:

```
if (ferror(fd)) {
  printf("Error reading from file\n");
  clearerr(fd);
}
```

6.2 Binary File I/O

Once the file is opened, we can read or write **binary** (i.e., non-ASCII) bytes to the file by using the **`fread()`** and **`fwrite()`** functions, respectively. Let's look at the **`fwrite()`** function first:

```
size_t fwrite(const void *dataPtr, size_t size, size_t n, FILE *fd);
```

Here, **`fd`** is the file descriptor that was returned from the **`fopen()`** function. The **`dataPtr`** pointer points to the block of memory that contains the data to be written to the file. The **`size`** parameter indicates the number of bytes of each item to be written, and the **`n`** parameter indicates the number of items to be written. If the function is successful, it returns the count of the number of items that were successfully written to the file. If an error occurred, then it returns a number less than **`n`**. In most systems, the size_t type is equivalent to an **unsigned int**.

Here is how we might write out a **float** value to a file stream:

```
float f = 100.13;
fwrite(&f, sizeof(float), 1, fd);
```

Notice how we pass in a pointer to the **float** and also the size of the **float** as well as the number of floats to be written. In addition to simple primitive types, we can output strings, arrays and structs. Here is a full example that shows this:

Code from **fwriteExample.c**

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

struct person {
  char  name[10];
  int   age;
  float weight;
};

int main() {
  FILE  *fd;

  // Open a new file for writing only
  fd = fopen("sampleFile.dat", "w");
  if (!fd) {
     printf("ERROR: Could not open file\n");
     exit(1);
  }

  // Write out a float
  float  data = 1078.24;
  fwrite(&data, sizeof(float), 1, fd);

  // Write out a string of 30 chars
  char  sentence[30] = "This is a sentence.";
  fwrite(sentence, sizeof(char), 30, fd);
```

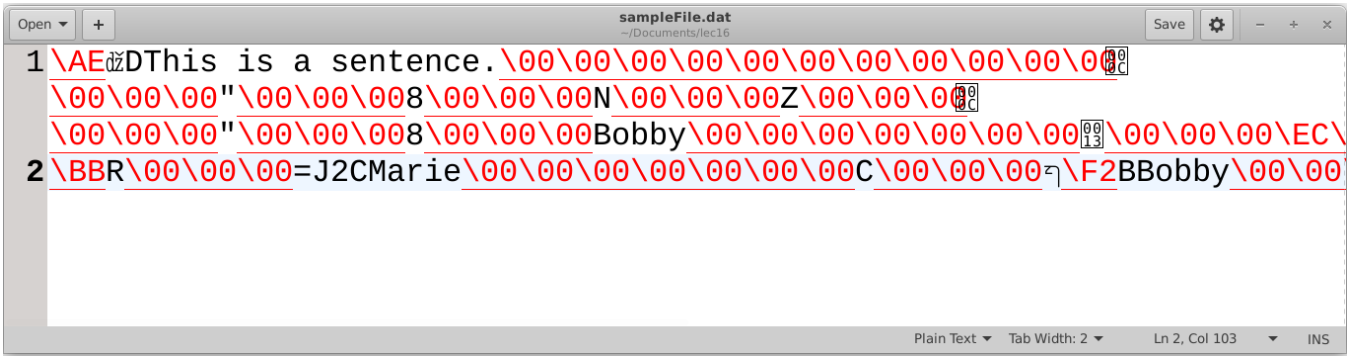
It is important to have a fixed-size string here because when reading back in, we need to know how many bytes belong to the string. It makes things more difficult if we have variable-sized strings.

```
  // Write out a whole array of ints
  int intArray[5] = {12, 34, 56, 78, 90};
  fwrite(intArray, sizeof(intArray), 1, fd);

  // Write out first three elements of the array
  fwrite(intArray, sizeof(int), 3, fd);

  // Write out an array of structs
  struct person friends[4] = {{"Bobby", 19, 143.57}, {"Jenny", 20, 110.32},
                              {"Fredy", 82, 178.29}, {"Marie", 67, 121.32}};
  fwrite(friends, sizeof(friends), 1, fd);

  // Write out a single friend, Bobby
  fwrite(friends, sizeof(struct person), 1, fd);

  // All done ... close the file
  fclose(fd);
}
```

When running this program, there is not much to see, as there is no output. However, the **sampleFile.dat** file is produced. But it itself contains binary data, so it is mostly unreadable:

```
student@COMPBase:~$ ./fwriteExample
student@COMPBase:~$ more sampleFile.dat
␀džDThis is a sentence.
student@COMPBase:~$
```

Opening the file in **gedit** will give an error and show a little bit more, but it will still look weird:



In order to know whether or not everything was saved properly to the file, we will need to read the information back in to compare. To read a file back in, we use the **`fread()`** function which has this format:

```
size_t   fread(void *dataPtr, size_t size, size_t n, FILE *fd);
```

The **`dataPtr`** is the starting address of the memory block where data will be stored after reading from the file. The function reads **`n`** items from the file where each item occupies the number of bytes specified by **`size`**. On success, it reads **`n`** items from the file and returns **`n`**. On error or end of file, it returns a number less than **`n`**.

Here is how to read in a **float** … notice how it is basically the same structure as writing a float:

```
float f;

fread(&f, sizeof(float), 1, fd);
```

To read in our **sampleFile.dat** from the previous **fwriteExample.c** program, here is a complete program:

Code from **freadExample.c**

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

struct person {
  char  name[10];
  int   age;
  float weight;
};

int main() {
  FILE  *fd;

  // Open the file for reading only
  fd = fopen("sampleFile.dat", "r");
  if (!fd) {
     printf("ERROR: Could not open file\n");
     exit(1);
  }

  // Read in a float
  float  data;
  fread(&data, sizeof(float), 1, fd);
  printf("Here is the float: %f\n", data);

  // Read in a string of 30 chars
  char  sentence[30];
  fread(sentence, sizeof(char), 30, fd);
  printf("Here is the sentence: \"%s\"\n", sentence);

  // Read in a whole array of ints
  int intArray[5];
  fread(intArray, sizeof(intArray), 1, fd);
  printf("\nHere is the array: {");

  for (int i=0; i<5; i++)
    printf("%d,", intArray[i]);
  printf("}\n");

  // Read in first three elements of the array
  fread(intArray, sizeof(int), 3, fd);
  printf("\nHere are the first three elements of the array: {");
  for (int i=0; i<3; i++)
    printf("%d,", intArray[i]);
  printf("}\n");
```

```
  // Read in an array of structs
  struct person friends[4];
  fread(friends, sizeof(friends), 1, fd);
  printf("\nHere are the friends:\n");
  for (int i=0; i<4; i++)
    printf("  Name: %s,  Age: %d,  Weight: %f\n",
           friends[i].name, friends[i].age, friends[i].weight);

  // Read in a single friend
  struct person friend;
  fread(&friend, sizeof(struct person), 1, fd);
  printf("\nHere is the first friend:\n");
  printf("  Name: %s,  Age: %d,  Weight: %f\n", friend.name,
         friend.age, friend.weight);
  printf("\n");

  fclose(fd);  // All done ... close the file
}
```

When we run the code, we should see the values that were written to the file:

```
Here is the float: 1078.239990
Here is the sentence: "This is a sentence."

Here is the array: {12,34,56,78,90,}

Here are the first three elements of the array: {12,34,56,}

Here are the friends:
  Name: Bobby,  Age: 19,  Weight: 143.570007
  Name: Jenny,  Age: 20,  Weight: 110.320000
  Name: Fredy,  Age: 82,  Weight: 178.289993
  Name: Marie,  Age: 67,  Weight: 121.320000

Here is the first friend:
  Name: Bobby,  Age: 19,  Weight: 143.570007
```

6.3 Text File I/O

There are two functions that can be used to read in formatted text data:

- **fprintf()**
  - writes to file from program variables according to format string
- **fscanf()**
  - reads from file into program variables according to format string

These work very similarly to **printf()** and **scanf()** that we have used previously, by writing/reading things out/in as text, as opposed to binary. Here is how to write/read a **float**:

```
float f = 100.13;
float g;

fprintf(fd, "%f", f);
fscanf(fd, "%f", &g);
```

When using these functions, you need to be careful to make sure that the data is readable later. For example, if we decide to output three integers like this:

```
int  x=10, y=4, z=78;

fprintf(fd, "%d%d%d", x, y, z);
```

… then you will have a problem reading them in again since they will all appear as a single integer (i.e., **10478**) when read back in again.

If you place a space in between, then all is ok:

```
fprintf(fd, "%d %d %d", x, y, z);
```

A more serious concern is when printing strings, since the size of the string is not known when reading in again. Consider, for example, if you do this:

```
char  sentence[30] = "This is a sentence.";
fprintf(fd, "%s", sentence);
```

The string will be properly stored in the file. However, when trying to read this with a simple string read from the **`fscanf()`** function, it will not work:

```
char sentence[30];
fscanf(fd, "%s", sentence);              // reads in only "This"
```

The problem is that the **`"%s"`** format string in **`fscanf()`** will only read in characters up to a **space** or **tab** or **newline**, then it will stop.

Alternatively, you can read in a fixed number of characters like this:

```
char sentence[30];
fscanf(fd, "%30c", sentence);    // reads in exactly 30 chars
```

This will read in exactly **30** characters, which would be fine if we outputted **30** characters. We could force **30** characters to be printed when outputting by adding spaces afterwards by using **-30** as the width like this:

```
char  sentence[30] = "This is a sentence.";
fprintf(fd, "%-30s", sentence);
```

Then we could read in the **30** characters and all would be good, except that the string will now have extra spaces after it:

```
"This is a sentence.           "
```

We could write some code to remove the trailing spaces. Another option would be simply to indicate, in the file, the number of characters to be read:

```
char  sentence[30] = "This is a sentence.";
fprintf(fd, "%d %s", strlen(sentence), sentence);
```

Here is what would appear in the file:

```
19 This is a sentence.
```

Then reading would be simpler. We read in the size, then one character at a time like this:

```
char  sentence[30];
int   size;

fscanf(fd, "%d ", &size);
for (int i=0; i<size; i++)
  fscanf(fd, "%c", &sentence[i]);
```

This will do just fine. It would likely be simpler, however, to just use **`fwrite()`** and **`fread()`** to write/read strings as their maximum number of chars, even if garbage data follows the `'\0'`:

```
fwrite(sentence, sizeof(char), 30, fd);
fread(sentence, sizeof(char), 30, fd);
```

Keep in mind, however, that there can always be problems if you make assumptions. For example, we might assume that the string does not start with digits. If it does, it could interfere with things, if for example, we were to write out an integer just before the string:

```
int  d = 76;
fprintf(fd, "%d", d);

char  sentence[30] = "52 cards in a deck";
fwrite(sentence, sizeof(char), 30, fd);
```

This will output as follows:

```
7652 cards in a deck
```

And the problem when reading like this:

```
int  d;
fscanf(fd, "%d",&d);
printf("Here is the int: %d\n", d);

char  sentence[30];
fread(sentence, sizeof(char), 30, fd);
printf("Here is the sentence: \"%s\"\n", sentence);
```

… is that we will get this result:

```
Here is the int: 7652
Here is the sentence: " cards in a deck"
```

Therefore, it is important to be careful and to not assume certain things about your data. Try to think of all the things that could go wrong and you will save yourself a lot of debugging pain/time.

Here is an adjusted version of our **fwriteExample.c** code to use **fprintf()** instead:

Code from **fprintfExample.c**

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

struct person {
  char  name[10];
  int   age;
  float weight;
};

int main() {
  FILE  *fd;

  // Open the file for writing only
  fd = fopen("sampleFile.txt", "w");
  if (!fd) {
     printf("ERROR: Could not open file\n");
     exit(1);
  }

  // Write out a float
  float  data = 1078.24;
  fprintf(fd, "%f\n", data);

  // Write out a string of 30 chars
  char  sentence[30] = "This is a sentence.";
  fprintf(fd, "%-30s\n", sentence);

  // Write out a whole array of ints
  int intArray[5] = {12, 34, 56, 78, 90};
  for (int i=0; i<5; i++)
    fprintf(fd, "%d ", intArray[i]);
  fprintf(fd, "\n");

  // Write out first three elements of the array
  for (int i=0; i<3; i++)
    fprintf(fd, "%d ", intArray[i]);
  fprintf(fd, "\n");

  // Write out an array of structs
  struct person friends[4] = {{"Bobby", 19, 143.57}, {"Jenny", 20, 110.32},
                              {"Fredy", 82, 178.29}, {"Marie", 67, 121.32}};
  for (int i=0; i<4; i++)
    fprintf(fd, "%s %d %f\n", friends[i].name,friends[i].age,friends[i].weight);

  // Write out a single friend
  fprintf(fd, "%s %d %f\n", friends[0].name,friends[0].age,friends[0].weight);

  // All done ... close the file
  fclose(fd);
}
```

The resulting **samplefile.txt** file would look like this:

```
1078.239990
This is a sentence.
12 34 56 78 90
12 34 56
Bobby 19 143.570007
Jenny 20 110.320000
Fredy 82 178.289993
Marie 67 121.320000
Bobby 19 143.570007
```

Here is the text-based version to read it back in using **fscanf()**:

Code from **fscanfExample.c**

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

struct person {
  char  name[10];
  int   age;
  float weight;
};

int main() {
  FILE  *fd;

  // Open the file for reading only
  fd = fopen("sampleFile.txt", "r");
  if (!fd) {
     printf("ERROR: Could not open file\n");
     exit(1);
  }

  // Read in a float
  float  data;
  fscanf(fd, "%f\n", &data);
  printf("Here is the float: %f\n", data);

  // Read in a string of 30 chars
  char  sentence[30];
  fscanf(fd, "%30c\n", sentence);
  printf("Here is the sentence: \"%s\"\n", sentence);

  // Read in a whole array of ints
  int intArray[5];
  printf("\nHere is the array: {");
  for (int i=0; i<5; i++) {
    fscanf(fd, "%d", &intArray[i]);
    printf("%d,", intArray[i]);
  }
  printf("}\n");
  fscanf(fd, "\n");
```

```
  // Read in first three elements of the array
  printf("\nHere are the first three elements of the array: {");
  for (int i=0; i<3; i++) {
    fscanf(fd, "%d", &intArray[i]);
    printf("%d,", intArray[i]);
  }
  printf("}\n");
  fscanf(fd, "\n");

  // Read in an array of structs
  struct person friends[4];
  printf("\nHere are the friends:\n");
  for (int i=0; i<4; i++) {
    fscanf(fd, "%s %d %f\n", friends[i].name,&friends[i].age,&friends[i].weight);
    printf("  Name: %s,  Age: %d,  Weight: %f\n", friends[i].name,
           friends[i].age, friends[i].weight);
  }

  // Read in a single friend
  struct person friend;
  fscanf(fd, "%s %d %f\n", friend.name, &friend.age, &friend.weight);
  printf("\nHere is the first friend:\n");
  printf("  Name: %s,  Age: %d,  Weight: %f\n", friend.name,
         friend.age, friend.weight);
  printf("\n");

  // All done ... close the file
  fclose(fd);
}
```

The output is the same as before:

```
Here is the float: 1078.239990
Here is the sentence: "This is a sentence.           "

Here is the array: {12,34,56,78,90,}

Here are the first three elements of the array: {12,34,56,}

Here are the friends:
  Name: Bobby,  Age: 19,  Weight: 143.570007
  Name: Jenny,  Age: 20,  Weight: 110.320000
  Name: Fredy,  Age: 82,  Weight: 178.289993
  Name: Marie,  Age: 67,  Weight: 121.320000

Here is the first friend:
  Name: Bobby,  Age: 19,  Weight: 143.570007
```

6.4 File Navigation/Positioning

Suppose there is a large file with a lot of binary data in it. For example, there may be a large database file with many records. If we wanted to read/update one of the records, it would be a slow process to always start at the beginning of a file and traverse through a bunch of records just to get to the one that we want to read/update. This process is called ***sequential access***, and it is slow, tedious and annoying. It would be better to be able to jump to any position in the file and start reading or updating. The idea of not having to start at the beginning, but to be able to jump anywhere in the file is called ***random access***.

Random access to data in a binary file is easy. It is similar to simply offsetting an array pointer by a certain known amount. Recall a previous example that incremented array pointers:

```
int  intArr[8] = {23, 54, 67, 88, 43, 12, 83, 46};

printf("First  int: %d \n", *intArr);       // displays 23
printf("Second int: %d \n", *(intArr + 1)); // displays 54
printf("Fourth int: %d \n", *(intArr + 3)); // displays 88
```

As you can see, we can access the 4th element in the array simply by offsetting the pointer by 3 “ints” (i.e., **12** bytes) from the start of the array in memory.

Each file has a ***file marker*** (a.k.a., ***file pointer***) that indicates where in the stream the next byte is to be read from or written to. It is similar to the idea of using your index finger to keep track of where you are. Each time we read/write, the file marker is incremented. You can ask **where** the file marker is at any time by using the **`ftell()`** function. You can also **move** the file marker to a particular location by using the **`fseek()`** function. Finally, you can **reset** the file marker back to the start of the file by using the **`rewind()`** function.

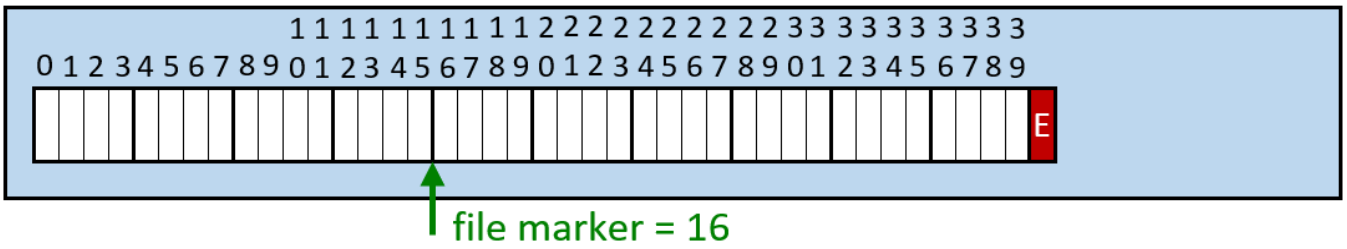
Consider creating a file with **10** integers stored in it and then reading the integers back in. Notice the program below which shows the file marker moving as it reads each integer:

Code from **fileMarker.c**

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

int main() {
  FILE  *fd;

  // Write out 10 random integers
  fd = fopen("integers.dat", "w+"); // read & write

  for (int i=0; i<10; i++) {
    int data = (int)(rand()/(double)RAND_MAX*100);
    fwrite(&data, sizeof(int), 1, fd);
  }
  rewind(fd);  // Reset to the beginning so we can read it in
```

```
  // Read in the 10 integers and show the file marker moving
  int    data;
  printf("File Marker starts at: %lu\n", ftell(fd));

  while ((fread(&data, sizeof(int), 1, fd)) > 0) {
    printf("Read in value: %d, File Marker moved to: %lu\n", data, ftell(fd));
  }

  printf("fread() returned -1, EOF reached\n");
  fclose(fd);
}
```

Here is what the file marker would look like after just **4** integers were read in:



The code above would print out this:

```
File Marker starts at: 0
Read in value: 84, File Marker moved to: 4
Read in value: 39, File Marker moved to: 8
Read in value: 78, File Marker moved to: 12
Read in value: 79, File Marker moved to: 16
Read in value: 91, File Marker moved to: 20
Read in value: 19, File Marker moved to: 24
Read in value: 33, File Marker moved to: 28
Read in value: 76, File Marker moved to: 32
Read in value: 27, File Marker moved to: 36
Read in value: 55, File Marker moved to: 40
fread() returned -1, EOF reached
```

Now that we understand that the integers are stored in order and that the file marker keeps track of where we are … let us see how we could alter one of the integers in the file by adjusting the file marker to that integer's location in the file (i.e., offset it by the appropriate amount) and then write to the file again.

In order to jump to any location within a file, we simply add the appropriate offset to the file marker by using the **`fseek()`** function. The key is to know how much to offset by. The **`fseek()`** has this format:

```
int  fseek(FILE *fd, long int offset, int whereFrom);
```

The offset is the number of bytes to skip over (i.e., offset from) depending on the `whereFrom` value … which can be one of the following:

SEEK_SET = **0** = the offset is with respect to the beginning of the file
SEEK_CUR = **1** = the offset is with respect to the current position in the file
SEEK_END = **2** = the offset is with respect to the end of the file

Here are some examples:

`fseek(fd, 0, SEEK_SET)`	moves file marker to **beginning** of file.
`fseek(fd, N, SEEK_SET)`	moves file marker to byte (**N + 1**) in file.
`fseek(fd, N, SEEK_CUR)`	moves file marker **N** bytes forward from current position in file.
`fseek(fd, -N, SEEK_CUR)`	moves file marker **N** bytes backward from current position in file.
`fseek(fd, 0, SEEK_END)`	moves file marker to **end** of file.
`fseek(fd, -N, SEEK_END)`	moves file marker **N** bytes backward from end of file.

Consider our previous example. Assume that we want to alter the **8th** integer in the file, perhaps by increasing it by **10**. We would have to position the file marker to the **8th** integer using **`fseek()`**, read the value, increase it and re-write it as follows:

Code from **randomAccess.c**

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

int main() {
  FILE  *fd;
  int    data;

  // Read in the file with the 10 integers
  fd = fopen("integers.dat", "r+");
  printf("Original file values:\n");
  while ((fread(&data, sizeof(int), 1, fd)) > 0) {
    printf("%d, ", data);
  }

  // Position the file marker and read in the 8th integer
  fseek(fd, sizeof(int)*7, SEEK_SET); // 7 since first integer starts at 0
  fread(&data, sizeof(int), 1, fd);

  // Increase the value and re-write
  data = data + 10;
  fseek(fd, sizeof(int)*7, SEEK_SET);
  fwrite(&data, sizeof(int), 1, fd);

  // Position back to  the start
  fseek(fd, 0, SEEK_SET);

  // Display the values of the file again
  printf("Modified file values:\n");
  while ((fread(&data, sizeof(int), 1, fd)) > 0) {
    printf("%d, ", data);
  }
  fclose(fd);
}
```

Here is the output, assuming the **integers.dat** file has already been written:

```
student@COMPBase:~$ ./randomAccess
Original file values:
84, 39, 78, 79, 91, 19, 33, 76, 27, 55,
Modified file values:
84, 39, 78, 79, 91, 19, 33, 86, 27, 55,
student@COMPBase:~$
```

Of course, running it over and over again will cause the 8th integer to keep increasing:

```
student@COMPBase:~$ ./randomAccess
Original file values:
84, 39, 78, 79, 91, 19, 33, 86, 27, 55,
Modified file values:
84, 39, 78, 79, 91, 19, 33, 96, 27, 55,
student@COMPBase:~$ ./randomAccess
Original file values:
84, 39, 78, 79, 91, 19, 33, 96, 27, 55,
Modified file values:
84, 39, 78, 79, 91, 19, 33, 106, 27, 55,
student@COMPBase:~$ ./randomAccess
Original file values:
84, 39, 78, 79, 91, 19, 33, 106, 27, 55,
Modified file values:
84, 39, 78, 79, 91, 19, 33, 116, 27, 55,
student@COMPBase:~$
```

Here are a couple of more useful functions:

```
remove(char *fileName)
```
- deletes the file with the given name. Returns **0** if it was deleted ok.

```
rename(char *oldFileName, char *newFileName)
```
- renames the file to the new name. Returns **0** if no error.

6.5 Buffers

When devices do I/O, both input and output data is often buffered.

> *A **Buffer** is an area of memory that is used to temporarily store data while it is being moved from one place to another.*

There are two main advantages of using buffering:

1. **To *regulate* the flow of data**. This can be handy if the receiver of the data is not ready. In that case, we can allow the data to collect (like a lineup forming) until such time as the receiver is ready to accept the incoming data. It may also be the case that the data is coming in too fast. In that case, only a little bit can be read at a time … data is buffered so that the sender is not slowed down needlessly.

2. **To *optimize* the flow of data**. Sometimes it is costly to perform an operation (e.g., there may be overhead in setting up things to receive some data). It is usually best to minimize the setup/overhead time by reducing such costly operations. This is often the case when accessing storage (e.g., disk drives, dvd drives, etc..), which can be slow, as mechanics may be involved to move the disk or disk's head. The idea is similar to lights at an intersection. Cars flow in one direction for a certain amount of time, then stop to allow flow in another direction. While one direction of flow is stopped, cars come along and are buffered while waiting to flow again.

Data cannot build up in a buffer indefinitely, as there are limits to buffer sizes (just as there are physical limits to how many cars can line up on a street or how many people could stand in a lineup. Eventually, the data must be released (i.e., allowed in/out). We use the term ***flush*** to indicate that we are sending the data out by emptying the buffer of its contents. It is analogous to the situation in which a traffic officer decides to let the traffic go in one direction until there is no more lineup.

When doing output, we should always flush a buffer once we are done with it … to ensure that all of our buffered data gets sent through (or pushed into) the stream so that the buffer is emptied. When we close a stream, this is done automatically. However, there are times that we want to flush a file buffer manually. We do so by calling the `fflush()` function.

We don't always need to buffer our data. If the data is coming in slowly then there is no need to have a buffer. We won't have, for example, a lineup of customers if a business receives only one brief-visiting customer every hour. Similarly, when receiving keyboard input, the keys are pressed so slowly (when compared to the CPU speed) that there is no need to buffer since we can get such input in real time. Hence each keystroke is flushed as it is pressed. However, the operating system does provide a means of buffering data from the keyboard.

In our programs, we need to decide when to flush. This could be done when our buffer gets full according to some predetermined limit, in which case it could be automatically flushed. This is called ***block buffering***. It is good to use when there are large data transfers.

Alternatively, we could use ***line buffering***. In which bytes accumulate until a newline character is encountered, at which point the buffer is automatically flushed. This is used, for example, when we enter shell commands in Unix/Linux. Even though the data is buffered, it can be changed/edited until the newline character is entered.

Here is a simple program it looks at buffering on the system console (i.e., **stdout**). Hence, the flushing is done when the `'\n'` character appears. First, **5** numbers are displayed, but only after the **5** are collected as a block of data. Second, each number is flushed with a newline character. Third, the numbers are flushed right away so that they appear on the same line.

Code from **flushExample.c**

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

int main() {

  // "Block" buffering
  printf("Here is an example of block buffering:\n");
  for (int i=1; i<=5; i++) {
    printf("%d ", i);
    sleep(1);
  }
  printf("\n");

  // Line buffering
  printf("Here is an example of line buffering:\n");
  for (int i=1; i<=5; i++) {
    printf("%d\n", i);
    sleep(1);
  }

  // No/unblocked buffering
  printf("Here is an example of no/unblocked buffering:\n");
  for (int i=1; i<=5; i++) {
    printf("%d ", i);
    fflush(stdout);
    sleep(1);
  }
  printf("\n");
}
```

The output is as follows:

```
Here is an example of block buffering:
1 2 3 4 5
Here is an example of line buffering:
1
2
3
4
5
Here is an example of no/unblocked buffering:
1 2 3 4 5
```

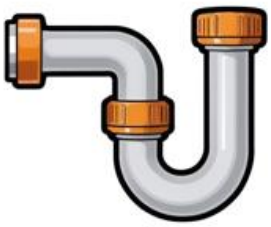
These all appear at once after 5 sec.

These appear one at a time each sec.

These appear one at a time each sec.

To have a better feel for ***block*** buffering, we would need a bigger example with many buffered objects that are flushed at a specific buffer limit.

6.6 Sources, Sinks and Pipes



As mentioned, files can be a ***source*** or a ***sink***. That is, data can be ***read in from*** a file, or it can be ***sent to*** a file. Files are essentially just a big one-dimensional array of bytes that happens to be stored in a non-volatile format (i.e., the data does not disappear when the power goes off). We can actually re-direct sources and sinks in our programs by using a pipe:

> *A **pipe** is a system call that creates a one way communication link between two file descriptors (i.e., between alternate sources and/or sinks).*

One of the most common uses of pipelining is for program testing.

By using it, we can:

- redirect input to come into our program from a file instead of the keyboard
- compare a program's actual output to an expected output
- store both input and expected output into files

Pipelining is the action of re-directing streams. We do this on the shell command line. We can actually *chain* multiple pipes together. Here are the pipelining symbols that we can use:

< - uses a specified file as **stdin** to the program, instead of the keyboard

> - redirects **stdout** of the program to a specified file, instead of terminal

| - redirects **stdout** from one program to **stdin** of another program

Consider this program that prompts for numbers, and then prints out the maximum:

Code from **redirect.c**

```c
#include <stdio.h>

#define MAX_NUMS 100

int main() {
  int arr[MAX_NUMS];
  int num, currMax;
  int count = 0;

  while (1) {
    scanf("%d", &num);
    if (num < 0)
      break;
    arr[count++] = num;
    if (count == MAX_NUMS)
      break;
  }

  currMax = 0;


  for (int i=0; i<count; i++) {
    if (arr[i] > currMax)
      currMax = arr[i];
  }
  printf("Max=%d\n", currMax);
}
```

If we were to run the program, the output is simple to determine:

```
student@COMPBase:~$ ./redirect
23
54
76
63
-1
Max=76
student@COMPBase:~$
```

Now consider using a **stdout** pipe redirect (i.e., **>**) to a file called **redirect.out** as follows:

```
student@COMPBase:~$ ./redirect > results.out
23
54
76
63
-1
student@COMPBase:~$ more results.out
Max=76
student@COMPBase:~$
```

Notice that a file called **results.out** was created and it contained the output from the redirect program. The output for the program did not appear in the shell window, because it was redirected to

the specified file. The input values (entered by the user) still appear, as these were not printed to **stdout** using **printf()**, they were the values echoed from the user as they were entered from the keyboard.

We can also have the input values brought into the program from a file instead of from the keyboard, by using the **<** pipe symbol. Imagine a file called **numbers.txt** that contains some integers, with a **-1** at the end. We can use this in the program:

```
student@COMPBase:~$ more numbers.txt
23
54
75
77
98
32
45
61
-1
student@COMPBase:~$ ./redirect < numbers.txt
Max=98
student@COMPBase:~$
```

As you can see, when the program runs, it does not ask the user for numbers, because it gets them from the file instead, due to the **<** pipe on **stdin**.

We can actually combine these to by chaining them together:

```
student@COMPBase:~$ ./redirect < numbers.txt > results.out
student@COMPBase:~$ more results.out
Max=98
student@COMPBase:~$
```

The order of the pipes is not important. Notice how the program just runs and produces output without the user being involved nor any results being shown.

The **|** pipe is interesting since it allows us to pass the results of a program to another program. Recall the **userInput** program that we wrote a long time ago:

```
student@COMPBase:~$ ./userInput
What is your name ?
Mark
Hello, Mark
student@COMPBase:~$
```

Let us run our **redirect** program, using the **numbers.txt** file and then pass the results to the **userInput** program instead of sending it to a file:

```
student@COMPBase:~$ ./redirect < numbers.txt | ./userInput
What is your name ?
Hello, Max=98
student@COMPBase:~$
```

Notice that the **./** is required for the **userInput** program so that the program could be found. Notice that the **Max=98** output is entered into the **userInput** program as if it was the name typed in.

This pipelining is very useful when running large files overnight, allowing you to write batch files with multiple input files and producing multiple output files.

We can also use pipelining in the Linux shell window.

Recall that **ps aux** printed out a lot of processes. Whenever we have a lot of things to display in a window, we can pipe the results into the **more** shell command. That will allow us to view one page at a time of the output.

```
student@COMPBase:~$ ps -aux | more
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.4 167676 12884 ?        Ss   Feb25   0:08 /sbin/init
root           2  0.0  0.0      0     0 ?        S    Feb25   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        I<   Feb25   0:00 [rcu_gp]
--More--
```

Now, assume that we have a text file called **names.txt** that contains the following names:

```
student@COMPBase:~$ more names.txt
Michael
Jessica
Ashley
Daniel
Robert
Brandon
Sarah
William
Nicole
Daniel
Laura
Brandon
Heather
Jessica
Eric
Megan
Laura
Kevin
Thomas
Jessica
Ashley
Robert
Christina
Laura
Daniel
Robert
student@COMPBase:~$
```

We can sort these names by using the **sort** command:

```
student@COMPBase:~$ sort names.txt
Ashley
Ashley
Brandon
Brandon
Christina
Daniel
Daniel
Daniel
Eric
Heather
Jessica
Jessica
Jessica
Kevin
Laura
Laura
Laura
Megan
Michael
Nicole
Robert
Robert
Robert
Sarah
Thomas
William
student@COMPBase:~$
```

Unfortunately, the names are displayed sorted but the file remains the same. We could save the sorted names by piping to a file like this:

```
student@COMPBase:~$ sort names.txt > namesSorted.txt

student@COMPBase:~$
```

Nothing is displayed, but the list of sorted names will be saved into **namesSorted.txt**. By using the **uniq** command, we can also display the unique names from the sorted list, which will get rid of duplicates:

```
student@COMPBase:~$ uniq namesSorted.txt
Ashley
Brandon
Christina
Daniel
Eric
Heather
Jessica
Kevin
Laura
Megan
Michael
Nicole
Robert
```

```
Sarah
Thomas
William
student@COMPBase:~$
```

If the names are not yet sorted, we could first **sort** them, then pipe to **uniq** and then even save the results to a file of sorted/unique names as follows:

```
student@COMPBase:~$ sort names.txt | uniq > uniqueNames.txt

student@COMPBase:~$
```

The resulting **uniqueNames.txt** file will contain the unique names sorted. The **cat** (i.e., concatenate) command can be used to extract portions of a file by allowing lines to be selected. We can pipe the results of a **cat** command to a **head** command which allows us to grab the first few lines of a file:

```
student@COMPBase:~$ cat names.txt | head -10
Michael
Jessica
Ashley
Daniel
Robert
Brandon
Sarah
William
Nicole
Daniel
student@COMPBase:~$
```

From that resulting list, we can take the last **5** names by using the **tail** command:

```
student@COMPBase:~$ cat names.txt | head -10 | tail -5
Brandon
Sarah
William
Nicole
Daniel
student@COMPBase:~$
```

The **grep** command allows us to search and extract names that match some kind of input string. Here is how to extract all names with an **"a"** in them:

```
student@COMPBase:~$ grep "a" names.txt
Michael
Jessica
Daniel
Brandon
Sarah
William
Daniel
Laura
Brandon
Heather
```

```
Jessica
Megan
Laura
Thomas
Jessica
Christina
Laura
Daniel
student@COMPBase:~$
```

Finally, here is how we can extract the top **5** names with the letter **"a"** in it from a sorted list of unique names and save it to a file:

```
student@COMPBase:~$ sort names.txt | grep "a" | uniq | head -5 > top5ANames.txt

student@COMPBase:~$ more top5ANames.txt
Brandon
Christina
Daniel
Heather
Jessica

student@COMPBase:~$
```

As you become a Linux expert, you will do well at this.