What did I learn?

Different header files contain different functions

```
#include <stdio.h> // for printf()
#include <fcntl.h> // for open(), close()
#include <unistd.h> // for unix read, write
```

Command line arguments

```
int main (int argc, char * argv[]) {

    Return 0;

}
```

When writing the code, these are the two command line parameters: argc, argv.

*argv is a pointer to an array of strings(which are also pointers to their respective first elements).

Each string is an argument passed when running the executable.

In C, a string and an array are pointers to the first element.

So in the second parameter to the main function, is *argv[]. This is a pointer to an array of strings(and strings are pointers to their first elements). So argv is a pointer to an array of pointer.

File Descriptor

A file descriptor is a non-negative integer that acts as an abstract handle to an input/output resource, in this case a file.

Offset

An offset (or file position) is a byte-count integer that tracks the current location within a file stream relative to a starting point (usually the beginning of the file).

If the file descriptor is the ticket that lets you into the library, the offset is the finger pointing to the exact character on the page where you last stopped reading.

When read() returns 0, it means the offset has reached the very end of the file (the "End of File" or EOF), and there's nothing left to point to.

What is a process?

A process is an active execution environment managed by the kernel. It consists of the program code, its own private address space in memory, and a set of system resources like file descriptors and environment variables. Unlike a thread, which shares memory with others, a process is isolated, meaning one process crashing won't typically take down another.

In Linux, everything is a file. What does it mean?

In Linux, a stream of bytes are called a file. This stream of bytes could be coming from anywhere, from the keyboard, from a network socket etc.

The 'everything is a file' philosophy means that the Linux kernel provides a unified abstraction for interacting with nearly all system resources. Whether you are talking to a hardware device (like a mouse), a network socket, a process, or a plain text document, the interface remains the same.

What is Unsigned

```
unsigned char buffer[16];
// why not char buffer[16]
// why unsigned
```

Why use unsigned ?

So an integer uses 4 bytes of memory, or 32 bits of memory.

In memory, in the metal, everything is stored as binary. 11111111 could be read as -1(if signed), or 255(if unsigned).

Since we are building a hexdump-utility whose purpose to see the exact bytes as they are, and we do not want the compiler to make assumptions for us that we may be dealing with negative number, we use the unsigned keyword.

What assumption is the compiler making here?

```c
#include <stdio.h>


int main () {
    char sc = 0xff;
    unsigned char uc = 0xff;
    printf("\nsc =  %d, %0X\nuc = %d, %0X\n", sc, sc, uc, uc);
    return 0;
}
```

output

sc = -1, FFFFFFFF

uc = 255, FF

===

There is something called integer extension, where additional zeros or ones are added on the left of any binary number to make it 32-bits. In the CPU, maybe for performance reasons, every integer is converted to a full 32-bit integer. So if we didn’t use the ‘unsigned’ keyword, our cpu may think of

11111111 as

11111111 - 11111111 - 11111111 - 11111111 = -1

By using the ‘unsigned’ keyword, the cpu thought of

11111111 as

00000000 - 00000000 - 00000000 - 11111111 = 255

We do not want the CPU making any assumptions, hence we use the ‘unsigned’ keyword.

What is ssize_t?

```
ssize_t bytesRead;
```

size_t is a datatype to store really large values. Its size is the word size of the computer(32-bit, or 64-bit).

Why not just use int?

Because the largest int value (in a 32-bit or a 64-bit) computer will be (2^32) - 1.

Why?

Because, the size of an int is 4 bytes.

On the other hand sizet on a 64-bit machine, has a size of 64 bits, which means that the largest value it can store is (2^64) - 1.

Now for representing the size of a file, there is always a chance it can exceed 4gb. If we used int, then there would be the overflow problem, where instead of seeing a positive number we see a negative number.

But in case of sizet in a 64-bit machine, the largest value that can be stored is a few exabytes. No file in the world is so large.

Understood. So why ssize_t?

Because there is always a chance for the read function (bytesRead = read();) to return -1.

This can happen in situations like the USB drive being removed abruptly.

Hence we needed to have a data type which can store really really large values and also have the ability to be negative. So ssize_t. The largest value that can be stored is ((2^64) - 1)/2.