**Buffers--** A temporary storage area.

**What is a buffer?**

A temporary storage area is called buffer. All standard input and output devices contain an input and output buffer. Streams are buffered, for example in the case of standard input , when we press the key on the keyboard, it isn’t sent to your program, rather it is buffered by operating system till the time is allotted to that program.

Even though the data might have an interpretation ( such as an array of structs with many fields), programs that read and write buffers often treat them as arrays of bytes. An array of bytes is not the same as a string, even though they are both declared char\* or char [].

1. They might not contain **ASCII characters** and they may not NULL- terminated.
2. You cannot use **strlen()** to find the length of data in a buffer (because the buffer may contain null bytes). Instead , you need to figure out the length of data by the return value from the system call (typically read) that generated the data.
3. You cannot use strcpy(), Strcat(), or related routines on bytes buffers; instead, you need to use memcpy().

**How does it affect Programming?**

On various occasions you may need to clear the unwanted buffer so as to get the next input in the desired container and not in the buffer of previous variable. For example, in the case of C after encountering “scanf()”, if we need to input buffer or else the desired input is occupied by buffer of previous variable, not by the desired container. On pressing “Enter” (carriage return) on output screen after the first input, as the buffer of previous variable was the space for new container(as we didn’t clear it), the program skips the following input of container.

You write a buffer of 123 bytes to a file using code like this:

**char \*fileName = "/tmp/foo"**

**#define BUFSIZE 4096**

**char buf[BUFSIZE]; // buffer containing at most BUFSIZE bytes**

**...**

**int outFile; // file descriptor, a small integer**

**int bytesToWrite; // number of bytes still to be written**

**char \*outPtr = buf;**

**...**

**if ((outFile = creat(fileName, 0660)) < 0) { // failure**

**// see** [**file permissions**](https://www.cs.uky.edu/~raphael/programming.html#Unix_files) **to understand 0660**

**perror(fileName); // print cause**

**exit(1); // and exit**

**}**

**bytesToWrite = 123; // initialization; 123 is just an example**

**while ((bytesWritten = write(outFile, outPtr, bytesToWrite)) < bytesToWrite) {**

**// not all bytes have been written yet**

**if (bytesWritten < 0) { // failure**

**perror("write");**

**exit(1);**

**}**

**outPtr += bytesWritten;**

**bytesToWrite -= bytesWritten;**

**}**

To get the compiler to allocate space for buffers, you must declare the buffer with a size that the compiler can compute, as in

**#define BUFSIZE 1024**

**char buf[BUFSIZE];**

If you just declare the buffer with no size:

**char buf[];**

then it has unknown size and C does not allocate any space. That's acceptable if **buf** is a formal parameter (that is, it appears in a procedure header); the actual parameter (provided by the caller) has a size. But it is not acceptable if **buf** is a variable. If you don't know the size of the buffer at compile time, you should use code like this:

**char \*buf = (char \*) malloc(bufferSize);**

where **bufferSize** is the runtime result of some computation