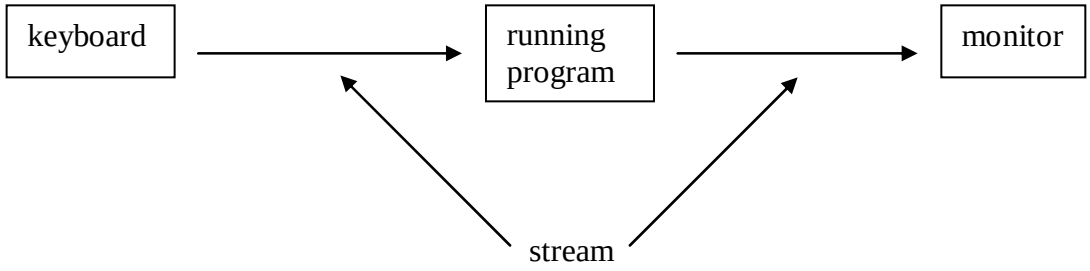
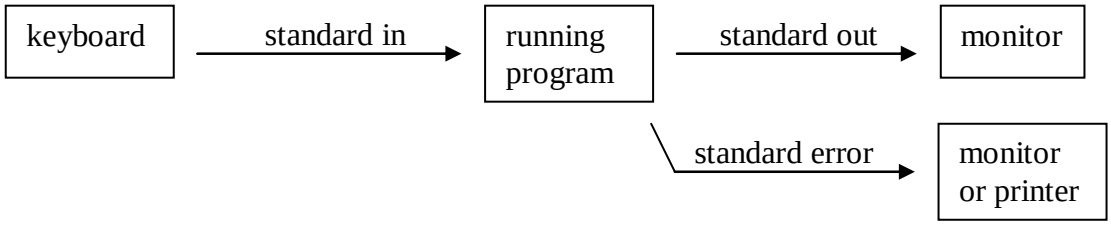
ECE 222 System Programming Concepts
Lecture notes – Streams, Buffers, Pipes

When a program is running, it is "connected" to the keyboard and to the monitor (and maybe to additional devices, but we will come to that later). Each connection is said to be a stream, representing a flow of data.



Every time a program is started, three streams are automatically created by the O/S.



The standard in stream carries bytes from the keyboard to the program, the standard out carries bytes from the program to the monitor (or the shell or window in which the program is running), and the standard error stream carries bytes from the program to either the same monitor, or perhaps a backup device like a printer.

In C, the scanf () function is actually a special version of the more generic fscanf () function, which can send receive bytes from any stream. For example:

```
#include <stdio.h>
char s[80];
fscanf(stdin,"%s",s);
```

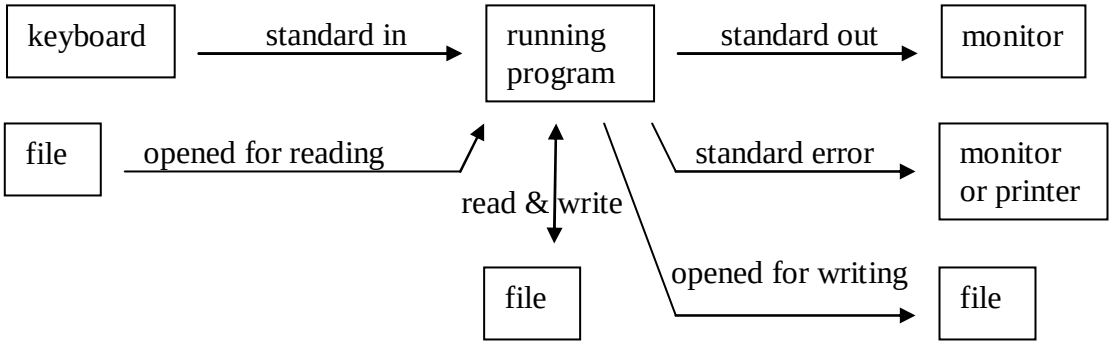
The same is true with regards to printf() and fprintf(), the latter is the generic version:

```
fprintf(stdout,"%s\n",s);
fprintf(stderr,"Hello error stream\n");
```

Note that these look a lot like how we access files. In fact, this is exactly how we access files, except that there is a stdin, stdout, or stderr. What are they?

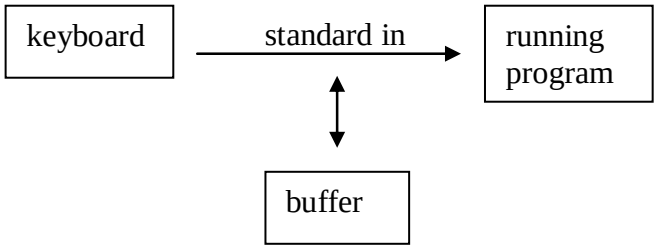
[Look inside /usr/include/stdio.h and search for them.]

They are in fact addresses, maintained by the O/S, of places to send and receive bytes. When you open a file, you get the same thing – an address. (The address is stored in a structure type-defined as FILE.) In other words, pieces of hardware are treated similarly to how files are treated; they are both accessed as streams from a given address.



While opening a file, a program indicates the direction of the stream. For a read/write stream, the program must be aware of two addresses, one at which to receive bytes and one at which to send bytes.

Each address is at a memory location controlled by the O/S. The O/S implements a buffer there. A buffer is a temporary storage to help manage the flow of bytes.



For example, what if the sender puts bytes into the stream faster than the receiver can handle? Or what if the program is in the middle of a calculation, and is not prepared to receive any bytes? The buffer can store up the bytes until the program is able to handle them, receiving them at the reduced rate, or when it is ready for them.

The keyboard and the running program do not need to know everything about how the buffer works, for example they do not know the address of the block of memory where bytes are temporarily stored. It is however important to know when the buffer "flushes". Flushing is the act of emptying out the temporary storage, sending all the bytes in the buffer on down the stream to the receiver. In general, a buffer is set up to flush in one of three modes:

block buffering – flushes when an entire "block" is full, such as 1 KB, or 4 KB, etc.
line buffering – usually indicates an ASCII-based stream, flushes when a CR is seen
unbuffered – flushes on every byte

You can see the effect of buffering through the following C code:

```
int i;
for (i=0; i<5; i++)
  {
  printf("i=%d  ",i);
  sleep(1);
  }
```

Running it as written above, you see no output until the program ends. This means that the stdout stream is buffered, either block or line. We can determine that it is line buffered by adding a newline at the end of the printf(). This time, every second we see a new line of output.

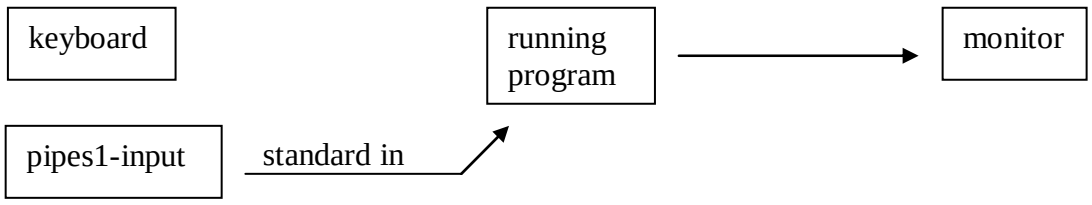
We can force the buffer to flush using the fflush(stdout) function call, without having to print a newline each time. Note that fflush() can be used on any stream.

Recall that the O/S automatically opens 3 streams for every running program. Because of this, the shell uses special symbols to "reconnect" those streams to other blocks. Using these symbols is called pipelining, or piping.

<	standard in comes from the given file
>	standard out goes to the given file
|	standard out from the first program goes to standard in for the second program

For example:

```
shell> pipes1 < pipes1-input.txt
```



We can do the same thing with the standard out stream:

```
shell> pipes1 < pipes1-input.txt > pipes1-output.txt
```

We can even connect the output from one program to the input to another program:

```
shell> pipes1 < pipes1-input.txt | pipes2
```

All of these reconnections, or redirections of input and output, can be chained together repeatedly. This allows us to write programs that perform single, simple operations, and

link them together into complex chains in order to accomplish tasks. This is where the phrase pipelining (or piping) comes from. For example:

```
shell> ls –al /usr/lib | grep libc | sort –r > results.txt
```

A nice set of standard programs has been built up over the years, following this methodology. Most unix/linux systems come with these programs installed. There are a few of which you should always be aware; more will become known and useful as one becomes more invested in system programming.

grep	search for the given string
sort	sorting
wc	count lines, words, bytes (chars)
more	interactive program to pause lengthy display
diff	compare two files

Note that even these simple programs have lots of options, controlled by command line arguments, to affect how they operate.

[Do in-class exercise depunct-inst.txt.]