Input and Output

Zeyneb Kurt

Outline

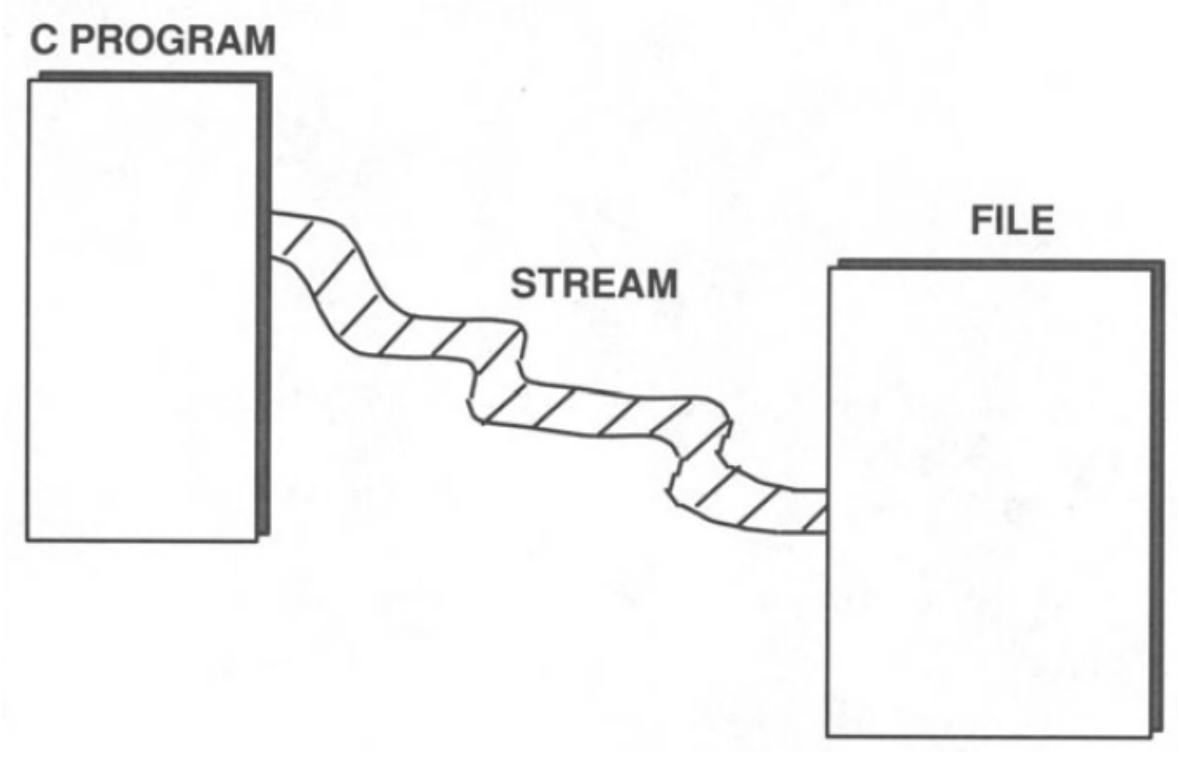
- Streams
- Standard input/output (I/O)
- File I/O
- Text and binary formats
- Error handling functions
- Buffer management and un-buffered I/O
- Random access to a file
- File management

Input and Output (I/O)

- Operating systems (OSs) vary greatly in the way they allow access to data in files and devices.
- This variation makes it extremely difficult to design I/O programs that are portable.
- The C language performs I/O through a large set of runtime routines. Some of these functions were derived from the UNIX I/O library.
- However:
  - The "C library" deals mostly with buffered I/O while the UNIX library performs unbuffered I/O.
  - The UNIX OS treats binary and text files the same. In some other OSs, the distinction is extremely important.
- ANSI Committee preserved, deleted, and modified some functions:
  - The biggest change is: elimination of unbuffered I/O functions. In the ANSI library, all I/O functions are buffered, still you can change the buffer size.
  - The ANSI I/O functions make a distinction between accessing files in binary or text mode.

- The Standard C Library contains nearly 40 functions that perform I/O operations.
- They can be divided into several groups:
  - Input / output to stdin (standard input, e.g. keyboard) and stdout (standard output, e.g. monitor)
  - Input / output to files
  - Error handling functions
  - File and folder management functions (e.g. delete, create, rename a file)

Streams



- C makes no distinction between devices such as a terminal or tape drive or files located on a disk.
- In all cases, I/O is performed through *streams* that are associated with the files or devices.
- A stream consists of an ordered series of bytes (such as a one-dimensional array of characters, as shown in the Figure).
- Reading and writing to a file or device involves reading data from the stream or writing data onto the stream.
- To perform I/O operations, you must associate a stream with a file or a device.
- You do this by declaring a pointer to a structure type called *FILE*.
- *The FILE* structure, which is defined in the *stdio.h header file, contains several fields to* hold such information as the file's name, its access mode, and a pointer to the next character in the stream.
- These fields are assigned values when you open the stream and access it, but they are implementation dependent, so they vary from one system to another.

- The *FILE structures provide the OS some metadata information*, but our only chance to access to the stream is the pointer to the *FILE structure* (*called a file pointer*).
- *The file pointer, which you must declare in* your program, holds the stream identifier returned by the *fopen() function.*
- *You* use the file pointer to read from, write to, or close the stream.
- A program may have more than one stream open simultaneously, although each implementation imposes a limit on the number of concurrent streams.
- One of the fields in each *FILE structure is a file position indicator that points to* the byte where the next character will be read from or written to.
- As you read from or write to the file, the OS adjusts the file position indicator to point to the next byte.
- Although you can't directly access the file position indicator (at least not in a portable fashion), you can fetch and change its value through library functions, thus enabling you to access a stream in non-serial order.
- Do not confuse the file pointer with the file position indicator:
  - The file pointer identifies an open stream connected to a file or device.
  - The file position indicator refers to a specific byte position (i.e. next character) within a stream.

Standard Streams

- There are three streams that are automatically opened for every program:
  - *stdin,*
  - *stdout,*
  - *stderr.*
- Usually, these streams point to your terminal, but many operating systems permit you to redirect them (eg you might want error messages written to a file instead of the terminal).
- The I/O functions already introduced, eg *printf() and scanf(), use these* default streams.
- *printf() writes to stdout, and scanf() reads from stdin.*
- *You* could use these functions to perform I/O to files by making *stdin and stdout point* to files (with the *freopen() function).*
- *However an easier method is to use the* equivalent functions, *fprintf() and fscanf(), which enable you to specify a* particular stream.

Text and Binary Formats

- Data can be accessed in one of two formats: *text or binary*.
- A *text stream consists of a series of lines, where each line is* terminated by a newline ('\n') character.
- However, OSs may have other ways of storing lines on disks, so each line in a text file does not necessarily end with a newline character.
- E.g. many IBM systems, keep track of text lines through an index of pointers to the beginning of each line.
- In this scheme, the files stored on disk or tape may not contain newline characters even though they are logically composed of lines.
- However, when these lines are read into memory in *text mode*, the runtime functions automatically insert newlines into the text stream.

Text format

- When lines are written from a text stream to a storage device, the I/O functions may replace new lines in the stream with implementation-defined characters that get written to the I/O device.
- In this way, C text streams have a consistent appearance from one environment to another, even though the format of the data on the storage devices may vary.
- Despite these rules, which promote portability to some extent, you should be extremely careful when performing textual I/O.
- Programs that work on one system may not work exactly the same way on another.
- In particular, the rules described above hold true only for printable characters (e.g. tabs, form feeds, and newlines).
- If control characters (non-printable characters) appear in a text stream, they are interpreted in an implementation-defined manner.

Binary format

- In binary format, the compiler performs no interpretation of bytes. It simply reads and writes bits exactly as they appear.
- Binary streams are used primarily for non-textual data, where there is no line structure and it is important to preserve the exact contents of the file.
- If you are more interested in preserving the line structure of a file, you should use a text stream.
- E.g. the 3 standard streams *(stdin, stdout, stderr)* are all opened in text mode.
- In UNIX environments the distinction between text and binary modes is moot since UNIX treats all data as binary data.
- However, even if you are programming in a UNIX environment, you should be thinking about potential difficulties in porting your program to other systems

Buffering

- Compared to memory, secondary storage devices such as disk drives and tape drives are extremely slow.
- For programs that involve I/O, the time taken to access these devices overshadows the time the CPU takes to perform operations.
- It is extremely important, therefore, to reduce the number of physical read and write operations as much as possible.
- Buffering is the simplest way to do this. A *buffer is an area where data is temporarily stored before being sent to its* ultimate destination.
- Buffering provides more efficient data transfer because it enables the OS to minimize accesses to I/O devices.
- All OSs use buffers to read from and write to I/O devices.
- The OS accesses I/O devices only in fixed-size chunks, called *blocks. (a block is 512 or 1024 bytes mostly)*
- *This means that even if you* want to read only one character from a file, the operating system reads the entire block on which the character is located.
- For a single read operation, this isn't very efficient, but suppose you want to read 1000 characters from a file: If I/O were unbuffered, the system would perform 1000 disk seek and read operations.
- With buffered I/O, the system reads an entire block into memory and then fetches each character from memory when necessary, which saves 999 I/O operations.

- The C runtime library contains an additional layer of buffering, which comes in two forms: *line buffering* and *block buffering.*
- In line buffering, the system stores characters until a newline character is encountered, or until the buffer is filled, and then sends the entire line to the OS to be processed.
- This is what happens, for example, when you read data from the terminal. The data is saved in a buffer until you enter a newline character. Then, the entire line is sent to the program.
- In block buffering, the system stores characters until a block is filled and then passes the entire block to the operating system.
- The size of a block is defined by the OS (typically 512 or 1024 bytes).
- By default, all I/O streams that point to a file are block buffered.
- Streams that point to your terminal *(stdin and stdout) are either line buffered or unbuffered, depending on* the implementation.

Buffer manager

- The C library standard I/O package includes a *buffer manager* that keeps buffers in memory as long as possible.
- So if you access the same portion of a stream more than once, there is a good chance that the system can avoid accessing the I/O device multiple times.
- Note: this can create problems if the file is being shared by more than one process.
- For inter-process synchronization: write your own assembly language functions or use system functions supplied by the OS.
- In both line buffering and block buffering, you can explicitly direct the system to flush the buffer at any time (*fflush() function), sending whatever data is* in the buffer *to its destination.*
- Although line buffering and block buffering are more efficient than processing each character individually, they are unsatisfactory if you want each character to be processed as soon as it is input or output.
- For example, you may want to process characters as they are typed rather than waiting for a newline to be entered.
- C allows you to tune the buffering mechanism by changing the default size of the buffer.
- In most systems, you can set the size to 0 to turn buffering off entirely (for unbuffered I/O operations)

The *<stdio.h> Header File*

- To use any of the I/O functions, include the *stdio.h*, which contains:
  - Prototype declarations for all the I/O functions.
  - Declaration of the *FILE structure*.
  - Several useful macro constants, including *stdin*, *stdout*, *stderr*, *EOF*, *and NULL*.
- ***EOF***: *the value returned by many functions* when the system reaches the end-of-file marker.
- ***NULL***: *the name for a null pointer*. It can be defined in another header file called *stddef.h*.
- To use NULL, you must either include stdio.h or stddef.h

Error Handling

- Each I/O function returns a special value if an error occurs.
- This error value varies from one function to another.
- Some functions return 0 for an error, others return a non-0 value, and some return *EOF.*
- There are also two members of the FILE structure that record whether an error or end-of-file has occurred for each open stream.
- End-of-file conditions are represented differently on different systems. Some systems have a special character that denotes the end of a file, while others use some method of counting characters to determine when the end of a file has been reached.
- In either case, an attempt to read data past the end-of-file marker will cause an end-of-file condition.
- *I/O function* returns the same value for an end-of-file condition as it does for an error condition. In these cases, you need to check one of the flags to see which event actually occurred. A stream's *end-of-file* and *error* flags can be checked via *the feof() and ferror() functions, respectively.*
- The *clearerr()* function sets both flags equal to zero.
- You must explicitly reset the flags with *clearerr( )-they are not automatically reset when you read them, nor are they* automatically reset to 0 by the next I/O call.
- They are initialized to 0 when the stream is opened, but the only way to reset them to 0 is with *clearerr(). – check the upcoming example codes !!!*

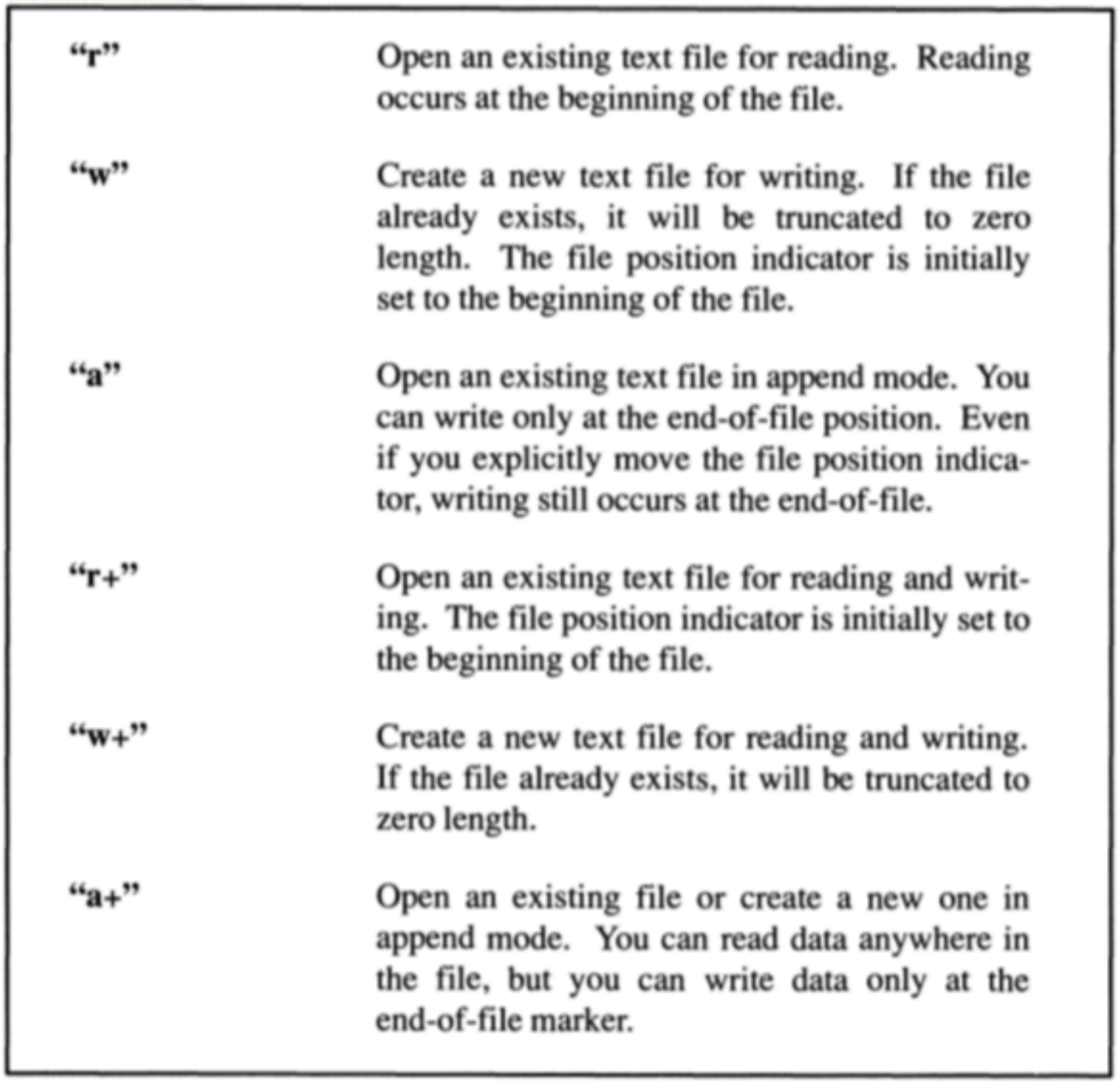
The *errno* variable

- In addition to the end-of-file and error flags, there is a global variable called *errno* that is used to record errors.
- In UNIX, an old version of *errno* is an integer variable declared in the *errno.h* header file.
- The *errno* variable is primarily used for math functions; very few of the I/O functions make use of *errno*.

Opening a File

- Before you can read from or write to a file, you must open it with the *fopen()* function.
- *fopen()* takes 2 arguments:
  - The file name and
  - The access mode.
- There are two sets of access modes:
  - One for text streams and
  - One for binary streams.

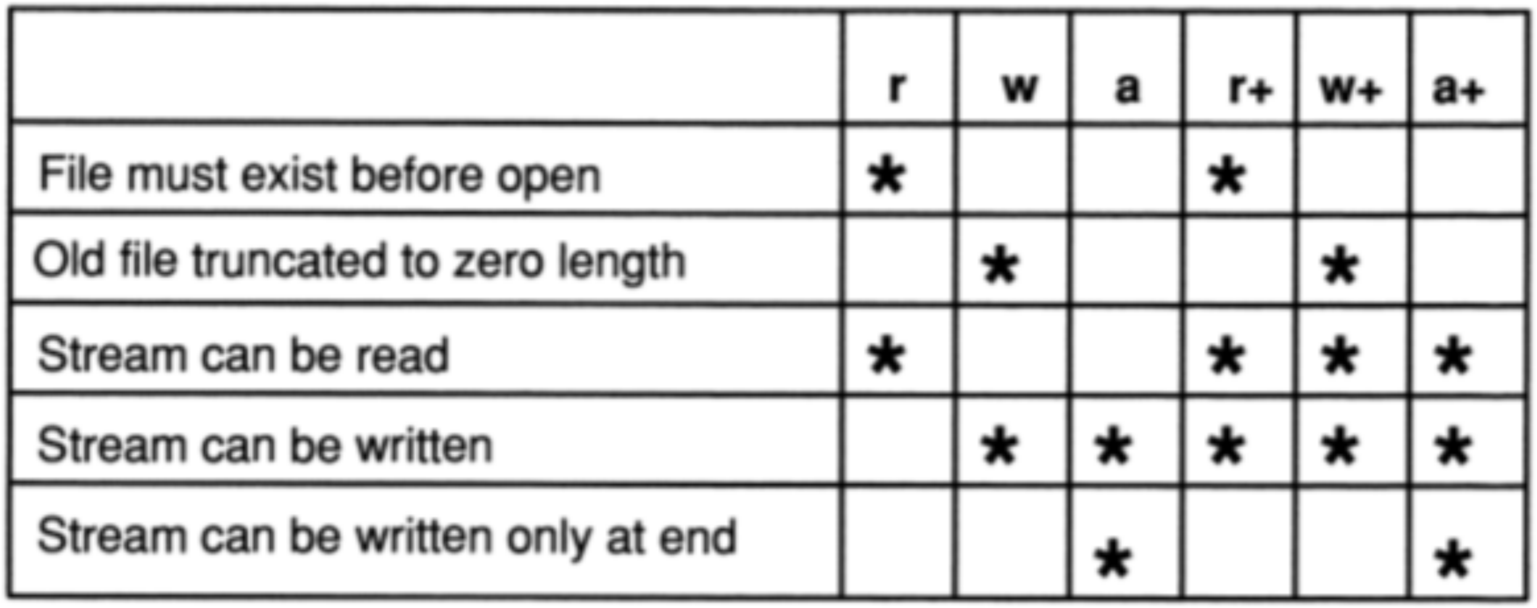
fopen() text modes



"r"	Open an existing text file for reading. Reading occurs at the beginning of the file.
"w"	Create a new text file for writing. If the file already exists, it will be truncated to zero length. The file position indicator is initially set to the beginning of the file.
"a"	Open an existing text file in append mode. You can write only at the end-of-file position. Even if you explicitly move the file position indicator, writing still occurs at the end-of-file.
"r+"	Open an existing text file for reading and writing. The file position indicator is initially set to the beginning of the file.
"w+"	Create a new text file for reading and writing. If the file already exists, it will be truncated to zero length.
"a+"	Open an existing file or create a new one in append mode. You can read data anywhere in the file, but you can write data only at the end-of-file marker.

- The binary modes are exactly the same, except that they have a "b" appended to the mode name.
- For example to open a binary file with read access you would use "rb".

File and Stream properties of fopen() modes



	r	w	a	r+	w+	a+
File must exist before open	*			*		
Old file truncated to zero length		*			*	
Stream can be read	*			*	*	*
Stream can be written		*	*	*	*	*
Stream can be written only at end			*			*

- *fopen()* returns a file pointer that you can use to access the file later in the program (check the example code).
- Note how the file pointer *fp* is declared as a pointer to *FILE.*
- *fopen()* returns a null pointer *(NULL)* if an error occurs.
- If *successful, fopen()* returns a non-zero file pointer.
- *fprintf()* is exactly like *printf(),* except that it takes an extra argument indicating which stream the output should be sent to.
- In this code, we send the message to the standard I/O stream *stderr.* By default, this stream usually points to your terminal.

Closing a File

- To close a file, you need to use the *fclose()* function: fclose(fp);
- Closing a file frees up the *FILE* structure that *fp* points to so, the OS can use the structure for a different file.
- It also flushes any buffers associated with the stream.
- Most OSs have a limit on the number of streams that can be open at once, so it's a good idea to close files when you're done with them.
- In any event, all open streams are automatically closed when the program terminates normally.
- Most OSs will close open files even when a program aborts abnormally, but you can't depend on this behavior.

Reading and Writing Data

- Once you have opened a file, you use the file pointer to perform read and write operations.
- There are three ways to perform I/O operations on three different sizes of objects:
  - **One character at a time**
  - **One line at a time**
  - **One block at a time**
- Each of these methods has some pros and cons.
- One rule that applies to all levels of I/O is:
- You cannot read from a stream and then write to it without an intervening call to *fseek()*, *rewind()*, or *fflush()*.
- The same rule holds for switching from write mode to read mode.
- These three functions are the only I/O functions that flush the buffers.

One Character at a Time

- Four functions that read and write one character to a stream:
  - *getc()* A macro that reads one character from a stream.
  - *fgetc()* Same as *getc(),* but implemented as a function.
  - *putc()* A macro that writes one character to a stream.
  - *fputc()* Same as *putc(),* but implemented as a function.
- **Note**: *getc()* and *putc()* are usually implemented as macros whereas *fgetc()* and *fputc()* are guaranteed to be functions.
- Because *putc* and *getc* are implemented as macros, they usually run much faster. They are almost twice as fast as *fgetc* and *fputc*
- However since they are macros, they are susceptible to side effect problem e.g. this is a dangerous call that may not work as expected: putc(’x', fp[j++]);
- If an argument contains side effect operators, you should use *fgetc()* or *fputc(),* which are guaranteed to be implemented as functions.
- Check the example code

One Line at a Time

- There are two line-oriented *I/0 functions-fgets()* and *fputs()*.
- The prototype for *fgets()* is: char *fgets(char *s, int n, FILE *stream);
- The three arguments of fgets():
  - *s* A pointer to the 1st element of an array to which characters will be written.
  - *n* An integer representing the max number of characters to read.
  - *stream* The stream from which to read.
- *fgets()* reads characters until it reaches a newline, or the end-of-file, or the maximum number of characters specified.
- *fgets()* automatically inserts a null character after the last character written to the array.
- So, we specify the "n" parameter to be one less than the "s" array #.
- *fgets()* returns *NULL* when it reaches the end-of-file.
- Otherwise, it returns the first argument ("s" string).
- The prototype for *fputs()* is: fputs(char *s, FILE *stream)
- *fputs()* writes the array identified by the 1st argument to the stream identified by the 2nd argument.

fgets() vs gets()

- *gets()* is the function that reads lines from *stdin.*
- Both functions append a null character ('\0') after the last character written.
- However, *gets()* does not write the terminating newline character to the input array. *fgets()* does include the terminating newline character (or an EOF if it just got the last line of the file).
- Also, *fgets()* allows you to specify a maximum number of characters to read, whereas *gets()* reads characters indefinitely until it encounters a newline or end-of-file.

- Check the example code to copy a file to another one, line by line.
- Note: we should open the files in "<u>text</u>" mode because we want to access the data <u>line by line</u>.
  - If we open the files in binary mode, *fgets()* might not work correctly because it would look explicitly for a newline character.
  - The file itself may or may not include newline characters.
  - If the file was written in text mode, it will contain newline characters.
- You might think that the *copyfile()* version that reads and writes lines would be faster than the version that reads and writes characters because it requires fewer function calls.
- Actually, the version using *getc()* and *putc()* is significantly faster.
- This is because most compilers implement *fgets()* and *fputs()* using *fputc()* and *fgetc()*. Since these are functions rather than macros, they tend to run more slowly.

One Block at a Time

- We can also access data in lumps called *blocks*.
- A block is like an array.
- When you read or write a block, you need to specify the number of elements in the block and the size of each element.
- The two block I/O functions are: *fread()* and *fwrite()*.
- The prototype for *fread()* is:

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

- The prototype for fwrite() is:

void fwrite(void *ptr, size_t size, size_t nmemb, FILE *stream);

- size_t is an integer *type* defined in stdio.h and the arguments are:
  - *ptr* A pointer to an array (mostly char array), in which to store data.
  - *size* The size of each element in the array.
  - *nmemb* The number of elements to read.
  - *stream* The file pointer.
- *read()* returns the number of elements actually read, which should be the same as the 3rd argument unless an error occurs or an EOF condition is encountered.
- The *fwrite()* is the mirror of *fread()*, takes the same arguments, but instead of reading elements from the stream to the array, it writes elements from the array to the stream.

- Like *fputs()* and *fgets()*, the block I/O functions are usually implemented using *fputc()* and *fgetc()* functions, so they are not as efficient as the macros *putc()* and *getc()*.
- Note also that these block sizes are independent of the blocks used for buffering.
- The buffer size, for instance, might be 1024 bytes. If the block size specified in a read operation is only 512 bytes, the OS will still fetch 1024 bytes from the disk and store them in memory.
- But, only the first 512 bytes will be available to the *fread()*.
- On the next *fread()* call, the OS will fetch the remaining 512 bytes from memory rather than performing another disk access.
- So, the block sizes in *fread()* and *fwrite()*, do not affect the number of device I/O operations performed.
- Check the example code!

Selecting an I/O Method

- Choosing the best I/O method is a matter of weighing pros and cons, paying special attention to *simplicity*, *efficiency*, and *portability*.
- From an *efficiency* standpoint, the macros *putc()* and *getc()* are usually fastest.
- However, most OSs can perform very fast block I/O operations that can be even faster than *putc()* and *getc()*.
- These capabilities are often *not* available through the C runtime library. You may need to write assembly code or call OS services.
- E.g. UNIX systems provide routines called *read()* and *write()*, which perform efficient block I/O transfers.
- If you think you may want to use OS's block I/O operations in the future, it is probably a good idea to write the original C routines using *fread()* and *fwrite()* since it will be easier to adapt these routines if they are already block oriented.
- Though efficiency is important, it is not the only consideration.
- Sometimes the choice of an I/O method requires *simplicity*.
- For example, *fgets()* and *fputs()* are relatively slow functions, but it is worth sacrificing some speed if you need to process entire lines.

- Consider a function that counts the number of lines in a file. By using *fgets*() and *fputs(),* we can write this program very *simply*.
- We could also write this function using character or block I/O, but the function would be more complex.
- If execution speed is not important, therefore, using the *fgets*() and *fputs*() is the best.
- The last consideration in choosing an I/O method is *portability*.
- In terms of deciding between character, line, or block, I/O *portability* doesn't really play a role.
- But, *portability* is a major concern in *choosing between text and binary mode.*
- If the file contains textual data, such as source code files and documents, we should open it in text mode and access it line by line.
- This will help us to avoid many pitfalls if we port the program to a different machine.
- If the data is numeric and does not have a clear line structure, it is best to open it in binary mode and access it either character by character or block by block.

Unbuffered I/0

- Although the C runtime library provides us to change the buffer size, we should use the capability with care.
- Compiler developers have chose a default buffer size that is optimal for the OS under which the program will be run.
- If you change it, you may experience a loss of I/O speed.
- When you want to turn off buffering, you can change the buffer size.
- Mostly, this happens when we want user input to be processed immediately
- Normally, the *stdin* stream is line-buffered, requiring the user to enter a newline character before the input is sent to the program. Sometimes, this is unsatisfactory.
- Consider a text editor program: The user may type characters as part of the text or enter commands. User could press an up-arrow key to move the cursor to another line. The I/O functions must be capable of processing each character as it is input, without waiting for a terminating newline char.

- To turn buffering off, you can use either *setbuf()* or *setvbuf()* function.
- The *setbuf()* takes 2 arguments. The 1st is file pointer, and the 2nd one is a pointer to a character array which is serve as the new buffer.
- If the array pointer is a null pointer, buffering is turned off:

  setbuf(stdin, NULL); // *setbuf()* does not return a value.
- The *setvbuf()* is similar to *setbuf(),* but takes 2 additional arguments that enable you to specify the type of buffering (line, block, or no buffering) and the size of the array to be used as the buffer.
- The buffer type should be one of 3 symbols (defined in *stdio.h):*
  - *_IOFBF:* block buffering
  - *_IOLBF:* line buffering
  - *_IONBF:* no buffering
- To turn buffering off, therefore, write:

  stat = setvbuf(stdin, NULL, _IONBF, 0);
- The *setvbuf()* function returns a non-0 value if it is successful.
- If it cannot fulfill the request, it returns 0.