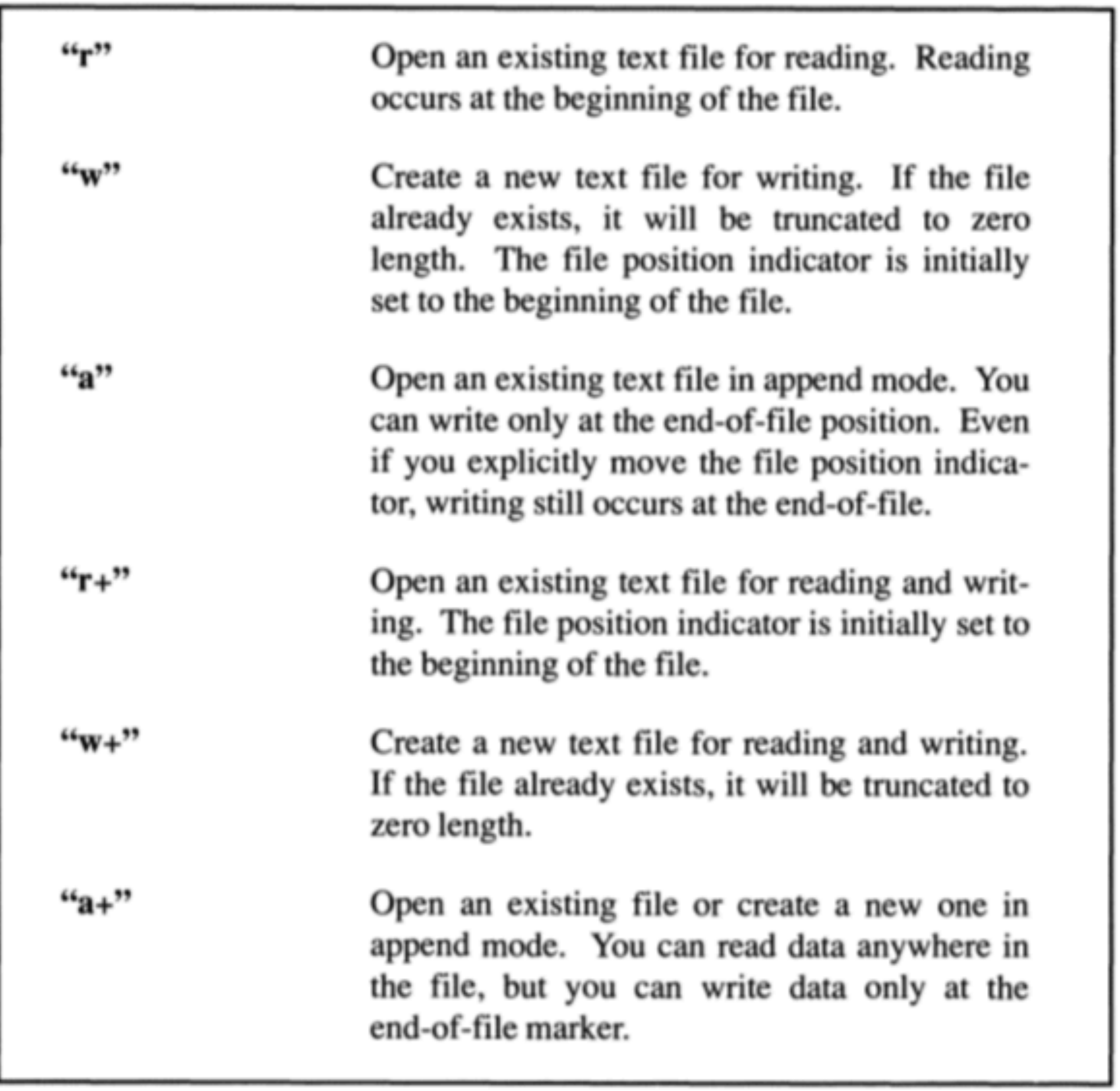
Input and Output (cont’d)

Zeyneb Kurt

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.

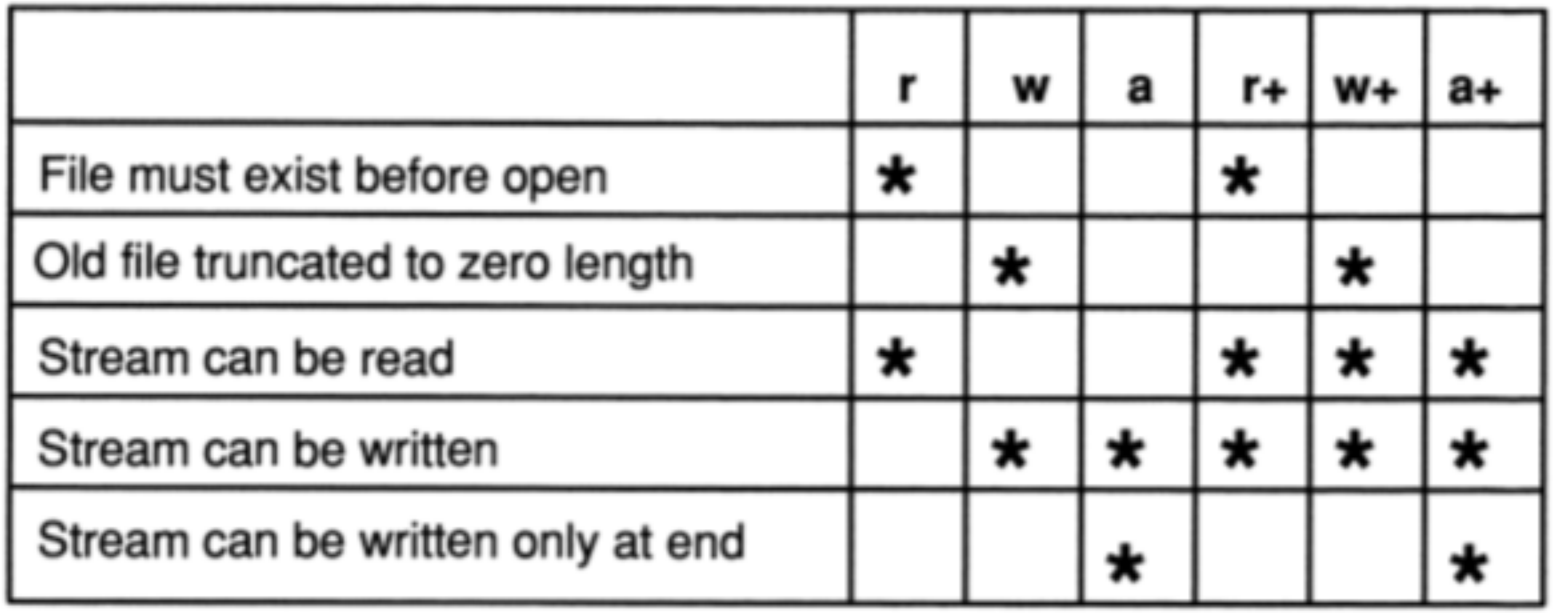
fopen() text modes



"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 "text" mode because we want to access the data line by line.
  - 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.

File Management Functions

- remove(): Deletes a file
- rename(): Renames a file
- tmpfile(): Creates a temporary binary file
- tmpnam(): Generates a string that can be used as the name of a temporary file.

Random Access - *fseek() / ftell()*

- So far we accessed files sequentially, beginning with the 1st byte and accessing each successive byte in order.
- For some applications this can be reasonable.
- However, for some applications, you need to access particular bytes in the middle of the file.
- In this case, we use 2 random access functions: *fseek() and ftell().*
- The *fseek()* moves the file position indicator to a specified character in a stream:

  int fseek(FILE *stream, long int offset, int whence);
- The arguments are:
  - *stream*: A file pointer
  - *offset*: An offset measured in characters (can be positive or negative).

    Binary: # of bytes. Text: Either 0, or a value returned by ftell().
  - *whence*: The starting position from which to count the offset.
- There are 3 choices for the *whence* argument, all of which are defined in *stdio.h:*
  - *SEEK_SET:* The beginning of the file.
  - *SEEK_CUR:* The current position of the file position indicator
  - *SEEK_END:* The end-of-file position.

- For example: stat = fseek(fp, 10, SEEK_SET);
- We move the file position indicator to character 10 of the stream. This will be the next character read or written.
- Note: streams, like arrays, start at the 0-th position, so character 10 is actually the 11-th character in the stream.
- The value returned by *fseek()* is 0 if the request is legal.
- If the request is illegal, *fseek()* returns a non-0 value.
- This can happen for a variety of reasons, the following is illegal if *fp* is opened for read-only access because it attempts to move the file position indicator beyond the end-of-file position: stat = fseek(fp, 1, SEEK_END)
- If *SEEK_END* is used with read-only files, the offset value must be less than or equal to 0.
- Similarly, if *SEEK_SET* is used, the offset value must be greater than or equal to 0.

- For binary streams, the *offset* argument can be any + or - integer value that does not push the file position indicator out of the file.
- For text streams, the *offset* argument must be either zero or a value returned by *ftell()*.
- The *ftell()* takes just one argument, which is a file pointer, and returns the current position of the file position indicator.
- *ftell()* is used to return to a specified file position after performing one or more *I/0* operations.
- For example, in most text editor programs, there is a command that allows the user to search for a specified character string.
- If the search fails, the cursor (and file position indicator) should return to its position prior to the search. This can be implemented as:

```
cur_pos = ftell(fp);
if (search (string) == FAIL)
        fseek(fp, cur_pas, SEEK_SET);
```

- Note: the position returned by *ftell()* is measured from the beginning of the file:
  - For binary streams, the value returned by *ftell()* represents the actual number of characters from the beginning of the file.
  - For text streams, the value returned by *ftell()* represents an implementation-defined value that has meaning only when used as an offset to *an fseek()* call.

Printing a File in Sorted Order - using structures and random access functions

- Suppose you have a large data file composed of records.
- Let's assume that the file contains 1000 records, where each record is a *PERSONALSTAT* structure, as declared earlier weeks.
- Suppose that the records are arranged randomly, but we want to print them alphabetically by the *name* field. First, you need to sort the records.
- There are two ways to sort records in a file: One is to actually rearrange the records in alphabetical order. However, there are several drawbacks to this method:
    - You need to read the entire file into memory, sort the records, and then write the file back to the storage device. This requires a great deal of I/O power.
    - It also requires a great deal of memory since the entire file must be in memory at once.
    - There are ways to sort a file in parts, but they are complex and require even more I/O processing.
    - Another drawback is that if you add records in the future, you need to repeat the entire process.

- The other sorting solution is to read only the part of the record that you want to sort (called the *key*, e.g. the name field*)* and pair each key with a file pointer (called an *index)* that points to the entire record in the file.
- Sorting the key elements involves less data than sorting the entire records. This is called an *index sort.*
- In the indexing sort method you don't need to rearrange the actual records themselves. You need only sort the index, which is usually a smaller task (in our example, the records are so short that there isn't much difference between sorting the records themselves and sorting the entries in the index file). To figure out the alphabetical order, though, you need to read in the *name* field of each record.

Suppose that the first five records have the following values.

Jordan, Larry	043-12-7895	5-11-1954
Bird, Michael	012-45-4721	3-24-1952
Erving, Isiah	065-23-5553	11-01-1960
Thomas, Earvin	041-92-1298	1-21-1949
Johnson, Julius	012-22-3365	7-15-1957

The key/index pairs would be

Check the example code!

index	key
0	Jordan, Larry
1	Bird, Michael
2	Erving, Isiah
3	Thomas, Earvin
4	Johnson, Julius

Instead of physically sorting the entire records, we can sort the key/index pairs by index value:

1	Bird, Michael
2	Erving, Isiah
4	Johnson, Julius
0	Jordan, Larry
3	Thomas, Earvin