CS 214 / 2021-03-03
===================

project 1

Three scenarios
1. reading from a file and writing to standard output
2. reading from standard input and writing to standard output
3. reading from a file and writing a different file

Scenarios 1 and 2 are just special cases of 3
	standard input is file descriptor 0
	standard output is file descriptor 1

you could create a single function that reads from a file descriptor and writes its output to a file descriptor

You could easily write this function and have it handle all three cases

	int wrap(unsigned width, int input_fd, int output_fd);

(e.g., return value indicates whether we exceeded the page width)

	int error = wrap(page_width, 0, 1); // scenario 2

when we read/write from a file, we obtain a file descriptor from open

Aside: since the width is always zero, why not use assert to check for bugs?

	assert(width > 0);

What is the idea of part 1?
What do I mean by efficient? How can we handle arbitrarily long files?

Question: how much memory does our program need to use? What in our input affects our memory requirements?
	How much do we need to keep track of at any given moment?

Answer: we never need to have more than one word in memory at a time
	as soon as we have complete word, we can decide whether to print it on the current line, or start a new line
	once we have printed a word, we don't need it any more

	so our program's memory requirement is O(n), where n is the length of the longest word
		we can still have additional memory use for variables, buffers, etc., but these can be O(1) size

Important: you do not need to read an entire line at once
	-> there is no easy way to do this! do not even try!

- you do not need to build the line in memory and print it all at once
	-> this is inefficient and unnecessary

But what about buffering?
	we can call read with a 1-character buffer, but this adds overhead to our program
		- each call to read involves a system call, which requires passing control to the OS (a "context switch")
	if we have a buffer of 128 characters, then we only need to call read every 128 characters (ideally)

Idea: choose a buffer size (can be big or small, doesn't matter)

	each time we call read, we try to fill the entire buffer
		e.g., if our buffer is 10 chars, we might get
				[Hello worl]

	we iterate through the buffer
		- return result from read tells us how much is in the buffer

	we examine each character and look for the start and end of words
		start of a word is a whitespace character followed by a non-WS char
		end of a word is a non-WS char followed by a WS char

	When we are in a word, we remember where it started (in the buffer)
	Once we reach the end of the word, we know its length and which bytes in the buffer make up the word

		[Hello worl]
		 ^ first char in word
		 ^ last char in word
		word is 5 chars long

		at this point, we can just send that to write
			-> remember, we can pass any pointer + size to write

			write(out_fd, &buf[word_start], word_len);

	But what happens when we reach the end of the buffer?

		[Hello worl]
		 ^ first char in word
		 ^ buffer ended, possibly inside a word

		-> we know how big the partial word is
		-> stash it somewhere
		-> call read again (refresh the buffer)
			-> overwrite the data in the buffer
				-> we don't need it anymore

		if the new buffer starts with WS, then the stashed text is
			a complete word
		otherwise, the stashed text is part of the word we are reading
		-> how you manage this is up to you

		[d!]

			-> "d!" is the end of the word we stashed
			-> so the complete word was "world!"

	Beware words that are too large for the buffer!
		if a word is too big for the buffer, just keep stashing it away
		use any handy data structure for this

If your code is written correctly, it shouldn't matter how big the buffer is; the same logic applies for a 1-byte buffer or a 1000-byte buffer

What is "whitespace"?
	Characters that don't have visible pixels on screen
	' ' space
	'\n' newline
	'\t' tab
	'\r' carriage return
	'\v' vertical tab
	etc.

	Use isspace() from ctype.h
		if isspace returns 1, the char is whitespace
		otherwise, it is not

Note: certain characters are "control codes"; these don't normally occur in text files, but are not considered whitespace
	'\a' bell
	'\b' backspace
	etc.

buffering and syscalls

As we said, syscalls are more expensive than regular function calls
we require a "context switch" in order pass control to the OS
	-> changes permissions on the processor (user mode to supervisor mode)
	-> programs run in virtual memory, OS does not
	-> etc.

Not a huge burden, but it is slower than a regular function call

-> this is why read and write work with a buffer
	-> do a lot of work with a single system call

The major difference between the Posix file functions (that use file descriptors) and the C file functions (that use FILE*), is that the C functions are buffered

when we call fopen, we get back a FILE *

what is FILE?
	this is a struct defined in stdio.h
	contents vary between compilers, but it generally includes

	- a file descriptor
	- a buffer (char array)
	- index of current byte in buffer
	- other info (e.g., are we at EOF)

On most Posix systems, fopen calls open, fread/fscanf/etc. use the buffer in FILE
	and call read when they need to refresh the buffer

	-> this is why fgetc is more efficient than calling read with a 1-char buffer

		fgetc reads a single character from a file
			-> but it only makes a syscall when the buffer is empty or completely read

	while ((ch = fgetc(my_file)) != EOF) {
		// do something with ch
	}

	makes fewer system calls than

	while ((bytes_read = read(my_fd, &ch, 1)) > 0) {
		// do something with ch
	}

On a Posix system, we can actually conver between file descriptors and FILE*

	fdopen -> create a FILE * for a file descriptor (add buffering!)

	fileno -> return the file descriptor used by a FILE *

Why is this useful?
	file descriptors can be used to access many things, not just files on disk
		- standard input/output/error
		- reading from devices in general
		- communicating with other processes (e.g., pipes)
		- communicating over a network (e.g., sockets)

The reason project 1 requires read/write is to give you a sense of why buffers are convenient
	- also, it gives you practice for working with sockets

Read/write give us more control about when data is obtained from/sent to the OS

How big should your buffer be?
	the tradeoff is memory use vs how often you call into the OS

	- smaller buffer uses less space
	- bigger buffer requires fewer syscalls

Common sizes: 64, 128, 256, 1024
	but this is arbitrary

using make to run tests

Remember: we can have our recipe run any program

	test: ww
		./ww 80 input.txt > output.txt

Now "make test" will (a) recompile ww if necessary, (b) wrap input.txt, and (c) save the output to "output.txt"

	-> this still requires us to examine output.txt after the test

We can create a file with the expected output, e.g., "reference.txt"

We can use cmp or diff to compare the content of two files
	cmp will find the first different byte between two files
	diff will print out a detailed description of the differences

	test: ww
		./ww 80 input.txt > output.txt
		cmp reference.text output.txt

	test2: ww
		./ww 80 input.txt > output.txt
		diff reference.text output.txt

	test3: ww
		./ww 80 input.txt | cmp reference.text

test3 sends the output of ww directly to cmp without the need for an output file

cmp and diff use exit status to signal whether the files were the same or different
make always checks the exit status and reports recipe failure

	if cmp gets one argument, it compares against standard input

	use - as an argument to diff to have it read standard input

		./program | diff reference -

		pipe output from program to diff and compare against "reference"

We can have multiple tests in our Makefile and have a target that runs all of them

	tests: test1 test2 test3 test4

	test1: ww
		./ww 80 input1.txt | diff reference1.txt -

exit, _exit, abort

Our program ends when we return from main, value returned from main is the exit status

```
int main(...)_
{
	...
	return EXIT_SUCCESS;   // end process with exit status 0
}
```

But we have a few ways to terminate a process early

	exit - terminates our process "normally"
		exit status given by argument to exit
		exit(EXIT_SUCCESS)
		exit(EXIT_FAILURE)

		- this will close files & flush FILE * buffers
			- e.g., anything we have written using FILE * that hasn't been sent to
				OS yet gets sent to OS
			- that is, make sure any pending printfs get to finish
		- this will call any "exit handlers" we have registered

		when we return from main, the run-time system calls exit next

	int atexit(void (*function)(void));
		- register a function as an "exit handler"
		- when we call exit/return from main, the run-time will call any registered
			exit handlers

	void cleanup();

	...
	{
		...
		atexit(cleanup); // register cleanup as an exit handler
			// should check return result to see if it succeeded
	}

	we can use this to allow our program to gracefully close network connections, etc., when it exits

	there can be many exit handlers (up to ATEXIT_MAX, which is at least 32)

_exit() is just like exit(), but it does not call the exit handlers and might not flush buffers

	void _exit(int status);

	this is "immediate exit"
	doesn't call our cleanup functions
	argument sets the exit status

abort() is for "abnormal termination"

	void abort();

	essentially crashes your program with the SIGABRT signal
		(more on signals later)

	reports a non-zero exit status

	does not do any cleanup
		lets OS close files, etc.

	does not call exit handlers
		-> can be caught if we install a "signal handler" for SIGABRT

	if something has gone very wrong, we maybe can't trust the contents of memory, so trying to clean up might cause further problems
		-> so just abandon the process and let someone else clean up

For this class, exit() is sufficient, but _exit() and abort() are okay to use -> be aware of the differences between them

next time: fork and exec