CS 214 / 2021-03-01
==========

Regarding project 1:

- yes, you will need to use opendir/readdir/closedir for part II
- for this assignment, you must use the POSIX file IO functions
        open, close, read, write
        Do not use the buffered functions: fopen, fclose, putchar, etc.

        Anything that uses a file descriptor is okay
        Anything that uses a FILE* is not allowed

- lseek is not forbidden, but you won't be able to use it
        - lseek does not work for standard input, when standard input is a
          terminal (TTY)
        - better to write a solution that does not rely on random file access
                -> treat the input as a stream of bytes

- how do we deal with a word that is is split between two or more calls to
        read?
        - e.g, we call read with some buffer size, and we get part of a word
        -> that is what you need to figure out
        -> use variables and heap objects to store data between calls

- do not care about where data from standard input comes from
        - just read the byte stream
        - if the stream contains \n, then that's a newline

        - if you want to test reading from stdin, just use redirection or a pipe

                ./ww 80 < some_text.txt
                echo 'this is a bunch of hooey' | ./ww 10

- read returns 0 at end of file

- treat all whitespace characters uniformly
        -> but do look for whitespace runs that include two or more '\n'
- treat all non-whitespace characters uniformly
        use isspace() from <ctype.h>
                -> returns true for ' ', '\n', '\t', '\r', '\v', a few others
        all non-whitespace characters are part of a word

- you can assume that your input will not include '\0'

assert

Do your programs check assumptions?

        If I have a function argument that should only be positive, do I check that?
        -> it's a good idea to do this!

        "impossible" things should never happen, so if they do happen you want
        to know about it
                -> it must be a bug, so you will want to fix it

rather than write a bunch of code, you can use assert to check for things
that should never happen

#include <assert.h>

{
        assert(x > 0); // looks like a function
                // if its argument is false (0), it prints an error message and
                // halts your program

}

For example: a.out: assert.c:6: main: Assertion `1 == 0' failed

-> note that the error message includes the source file and line number
    and the actual source code for the condition
                -> handy!

-> assert() is actually a macro
        it obtains the file and line number using __FILE__ and __LINE__
        it obtains the source code by "stringifying" its argument

#define ensure(X) if (!(X)) { puts("PANIC! " #X); abort(); }

        using #X says we want X as a string literal

ensure(x > 0)

->

if (!(x > 0)) { puts("PANIC! " "x > 0"); abort(); }

Remember that the compiler will concatenate adjacent string literals

Remember to check for errors! Even impossible errors!
        close only fails in unusual circumstances,
                but it might indicate a bug in your program, like closing a file twice
                if it fails, you want to know about it

        malloc almost never fails due to low memory,
                but it will definitely fail if you accidentally give it a negative argument

                malloc(-1) is the same as malloc(some enormous positive integer)

        int *p = malloc(....);
        assert(p);
                // some would argue this is inappropriate because we don't print the
                // specific reason malloc failed

Don't use assert for possible failures
        e.g., there are many reasons why open might fail other than bugs
        you should explicitly check for this

If you are using a function that sets errno, you can use perror to print an error
        message

        int fd = open(argv[1], O_RDONLY);
        if (fd == -1) {
                perror(argv[1]); // will print argv[1] followed by a text description of the problem
                exit(EXIT_FAILURE);
        }

#define strict(X) do { \
                if ((X) == -1) { \
                        perror(#X); \
                        exit(EXIT_FAILURE); \
                } \
        } while (0)

we use a few tricks
        -> \ at the end of the line continues the macro (convenient for readability)
        -> we use do { ... } while (0) to swallow a semicolon

        strict(close(fd));
->
        do { ... } while(0);
                -> compiler will realize that loop always runs exactly once and eliminate it

getting the line number as a string literal is tricky because of how macros are
expanded

#define xstr(X) #X
#define str(X) xstr(X)

#define strict(X) do { \
                if ((X) == -1) { \
                        perror(#X " @" __FILE__ ":" str(__LINE__)); \
                        exit(EXIT_FAILURE); \
                } \
        } while (0)

Working with directories

What is a directory?
-> it is a special file that contains directory entries
    -> "special" because its inode has a flag saying "this is a directory"
    -> a directory entry links a name to an inode
            -> each entry includes a name and an inode ID
            -> the inode ID is the "true name" of the file

We use opendir to open a directory file and read its contents

#include <sys/types.h>
#include <dirent.h>

DIR *opendir(char *path)

        -> returns a pointer to an abstract DIR structure
        -> returns NULL and sets errno on failure

int closedir(DIR *directory_pointer);

        -> close the directory & deallocate struct
        -> returns -1 and sets errno on failure

struct dirent *readdir(DIR *directory_pointer);

        -> obtain the next entry from the directory
        -> returns NULL if there are no further entries
        -> each time we call it, we get a pointer to a struct containing data about
           the next directory entry
                -> this is managed by the library; do not attempt to free this pointer
                -> it is allowed for readdir to rewrite the data and return the same
                        pointer each time
                        in other words, we can only use the pointer we got back from readdir
                        until the next call to readdir or closedir

        Copied from man page for readdir:
          struct dirent {
               ino_t   d_ino;   /* Inode number */
               off_t   d_off;   /* Not an offset; see below */
               unsigned short d_reclen;   /* Length of this record */
               unsigned char d_type;   /* Type of file; not supported
                                              by all filesystem types */
               char   d_name[256]; /* Null-terminated filename */
          };

        -> illustrative, but some tricks are played (d_name may be longer than 256)

        -> d_name is the name within the directory
                -> need to combine with directory name to get a complete path

                                <dir_name>/<file_name>

        -> d_type is a char, but its actual values are implementation specific
                compare values to predefined macros

                        DT_REG - type of regular files
                        DT_DIR - type of directory files

directory entries are not the same as file descriptors
- directory entry is information about a file on disk (name, inode number)
- file descriptor is information about an open file
        (where it comes from, current position in file)

file descriptors are specific to a process (running program)
directory entries are part of the file system

Note that there are always directory entries . and ..
        . - another entry for the directory itself
        .. - another entry for the parent directory

        Paths on Unix don't do any special magic for . and ..
                -> we literally just use the directory entries for them that always exist!

                ../../foo

                this path follows the .. entry to get to the parent,
                        then follows its .. entry to get to the grandparent,
                        and then looks up the foo entry

                if we think of the file system as a graph, then it is always cyclic
                if it is a tree, then it is a tree where we can always follow links up
                        towards the root

        Note that even the root, /, has a .. entry
                it points back to the root

        (Again, this is specific to Posix. Windows has its own rules.)

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <dirent.h>

int main()
{
    DIR *dirp = opendir("."); // open the current directory
    struct dirent *de;

    while ((de = readdir(dirp))) {
        //puts(de->d_name);
        printf("%lu %d %s\n",
               de->d_ino,
               de->d_type,
               de->d_name);
    }

    closedir(dirp); // should check for failure

    return EXIT_SUCCESS;
}

Exercise: recursively list all the children of a directory and its subdirectories
        -> watch out for . and .. !

Blocking

Certain IO functions may not return immediately
        -> if we are reading from a file, we may have to wait for data from the disk
        -> if we are reading from a TTY, we have to wait until the user types something

This is called "blocking"
        -> our program calls out to the OS for input, and the OS puts us on hold until
           the data becomes available

We expect functions like read and write to block in most circumstances

When does read block?
        If we ask for data, and none currently available, but the file has not closed
        If any data is available, read will return immediately, even if not all the
                bytes we asked for are present

        read(fd, buf, 128);
                <- if data is available, will copy up to 128 bytes to buf
                <- if we are at the end of the file, returns 0
                <- otherwise, it blocks until data becomes available

        When we read from a file, we will usually get as much data as we asked for,
        except for the last read
                -> This is because the OS caches the file system in memory, so data is usually
                   waiting for us

        When we read from a pipe, we will get data that is available, or block if we
                need to wait for the other program to write more

        When we read from a TTY, we block until the user hits return/enter or ^D
                this returns control to us and we get the data that was entered

                A TTY will store the text you type in a "line buffer"
                        Hitting enter adds \n to the end of the buffer and sends it to the program
                        Hitting ^D sends the buffer to the program without adding any characters
                                -> thus, if we enter ^D at the start of the line, read will receive 0
                                   bytes and return 0

                TTYs are weird, because we can still get more data even after EOF
                        -> but relying on this can lead to unexpected behavior if stdin is not
                                a TTY

                We should assume that read only returns 0 at EOF and stop reading