CS 214 / 2021-02-17
==========

arraylist example

```
	int x;
	arraylist_t list;  // local variable, but will refer to heap data
	al_init(&list, 10);
		// we passed a pointer, so al_init can change the actual fields
		// of this local variable

	al_remove(&list, &x);  // remove last item, write value to x

	al_remove(&list, NULL); // remove last item, discard value
```

Why do we pass the arraylist struct by reference?
	i.e., why do we pass a pointer?

	-> because we want our changes to persist after the function returns
	-> passing a copy of the struct would mean that we can't change used

---

What is the operating system for?
-> provides an interface between programs and the hardware
	-> abstracts details of the hardware
-> isolates programs and users
	-> multiple programs can run without interfering with each other

therefore, program IO almost always happens via OS services ("system calls" "syscalls")

	-> instead of talking directly to the disk or display hardware,
	 we ask the OS to send byte sequences appropriately

How system calls happen is different depending on the HW and OS
	-> C provides functions that wrap the syscall
	-> the functions themselves are HW/OS specific and may be written in assembly

Posix standardizes system calls for Unix-like operating systems
	there used to many, many proprietary Unix variants
	having a standard made it easier to write portable code
	-> nowadays, Unix is pretty much always Linux or BSD (including MacOS)
		-> plus a bunch of other software (eg. GNU utilities)

The Posix functions for file IO are open, close, read, write, &c.
-> central interface is the file descriptor
	-> this is a number that indicates which file you are talking about

Each process has a table of open files; the file descriptor is an index into this table
-> remember: files can be any stream of bytes
	-> e.g., we could read from a file on disk, or a terminal interface, or the network
	 or other processes

```
#include <fcntl.h>   // "file control"
	provides open and close

int open(char *filename, int flags)

	filename is the name of a file or a path to the file
	flags is a bit-vector that says what features we want

	Must start with one of these
		O_RDONLY  - open a file in read mode
		O_WRONLY  - open a file in write mode
		O_RDWD    - open a file in read/write mode

	Use bit-wise or to add additional features

		O_APPEND  - start with file pointer at the end of a file
		O_TRUNC   - delete contents of file
		O_CREAT   - create file if it does not exist (*special)

to open a file in append mode, we would say

	int fd = open("log.txt", O_WRONLY|O_APPEND);

Use "man 2 open" to see a list of possible flags


most of time we will open in read mode

	int fd = open("my_input.txt", O_RDONLY);

open returns -1 if it could not open the file
	file descriptors are always non-negative
	when open fails, it will set the global variable errno

traditionally, file descriptors 0, 1, and 2 are already open when your program starts
	0 is standard input
	1 is standard output
	2 is standard error
	you don't open these yourself


Using O_CREAT adds a third parameter to open, so we can specify its permissions

int open(char *filename, int flags, mode_t mode)
	mode indicates what permissions the file should have
		e.g., user/group/world readable/writeable/executable
	we will come back to this

Use close to close a file

	int fd = open("my_file", O_RDONLY);
	...
	close(fd);


Reading from a file

#include <unistd.h>

size_t read(int file_descriptor, void *buffer, size_t buffer_size)

	-> reads from the specified file
	-> writes bytes starting at the address specified in buffer
	-> reads up to buffer_size bytes
	-> returns the number of bytes read
		-> returns 0 at end-of-file
		-> returns -1 and sets errno if something went wrong

// read from stdin

int bytes_read;
char buf[256];

bytes_read = read(0, buf, 256);
	// read UP TO 256 bytes from standard input (0)
		// we might read fewer than 256 bytes, but we definitely won't read more
	// write bytes to buf (an array of chars)
	// return the number of bytes successfully read (assigned to "bytes_read")
		// if we ask for more bytes than there are available, we get all of them
		// if only 100 bytes are available, then bytes_read will be 100


read does not read strings! If we want a terminator, we have to add it ourself

// writing to a file

ssize_t write(int file_descriptor, void *buffer, size_t count)
	-> writes to the specified file
	-> bytes are taken starting at address specified (buffer)
	-> number of bytes to write is specified (count)
	-> returns the number of bytes successfully written
		-> returns -1 and sets errno if something went wrong
	-> note that the buffer can be any type!


// write to stdout

int bytes_written;
char buf[256] = "Hello!\n";

bytes_written = write(1, buf, strlen(buf));
	// write to stdout (file descriptor 1)
	// take bytes from buf
	// only write the bytes in the string, not including terminator
	// bytes_written will be assigned how much we were able to write

Q. why didn't we do this?

	write(1, buf, 256);
		-> this would write all the garbage data after the \0

	write(1, buf, strlen(buf) + 1);
		-> this would write the terminator
		-> ... but we don't normally want to do this

Remember: terminators are used in C strings, but files are not strings
	files can contain many \0 bytes, or none
	file length is tracked by file system


Using read and write with binary files
-> read and write work with any data
	-> we take bytes directly from memory without interpretation

Writing an array of ints to a file

int a[] = {1,2,3,4};

int fd = open("my_data", O_WRONLY|O_TRUNC|O_CREAT, S_IRUSR);

write(fd, a, sizeof(int) * 4);
	// writes the array of ints to the file directly
	// does not print as text! does no interpretation!
	// the literal bytes from memory are written to the file


the file will be (say) 16 bytes,

on a little-endian machine (Intel) with 4-byte ints, it will contain (in hex):
	01 00 00 00 02 00 00 00 03 00 00 00 04 00 00 00

on a big-endian machine (ARM) with 4-byte ints, it will contain (in hex):
	00 00 00 01 00 00 00 02 00 00 00 03 00 00 00 04

this is why file transfer between different hardware can be challenging!


We can use read in the same way

int a[4];
int fd = open("my_data", O_RDONLY);
read(fd, a, 4 * sizeof(int));

	// reads 16 bytes from the file and writes the bytes to memory in a
	// no interpretation!


Typical pattern when reading a text file
	two loops
	outer loop calls read and writes to a char buffer
		inner loop iterates through the bytes in the buffer
	outer loop ends when read returns 0 (EOF) or -1 (error)

// int bytes_read, length;
// char *buffer or char buffer[...]

bytes_read = read(my_file_desc, buffer, length);
while (bytes_read > 0) {
	for (i = 0; i < bytes_read; ++i) {
		// do something with buffer[i]
	}
	bytes_read = read(my_file_desc, buffer, length);
}

	// note that we have to keep reading until we get 0
	// don't want to assume how big the file is
	// allocating a giant buffer is wasteful


Some people prefer not to duplicate the call to read

while ((bytes_read = read(my_file_desc, buffer, length)) > 0) {
	for (i = 0; i < bytes_read; ++i) {
		// do something with buffer[i]
	}
}

Recall that assignment is an expression that evaluates to the value assigned
	(x = foo()) == y
		// calls foo(), assigns return value to x, and compares with y

Some people feel that putting the call to read in the while loop condition is
confusing and makes it hard to see what is happening


Reading:
	man pages for open, close, read, write
	these are in section 2 of the manual, so use "man 2 read", etc. to make sure
		you get the right one
```