In this lab, we’ll look closely at the character and string data types in C and at the standard functions (in the `ctype` and `string` libraries) for operating on values of those types.

### Pairs
- Yuyin Sun and Sanjay Sudhir
- Gemma Nash and Saung Thuya
- Eli Salm and Tyler Williams
- Sophie Gaschott and Josh Lavin
- Lilly Webster and Faizaan Ali
- Dennis Chan and Erhaan Ahmad
- Jae Hong Shin and Mattori Birnbaum
- Zachary Susag and Ella Nicolson
- Anna Blindermann and Cory McCartan
- Ying Zhang and Tanner Tufto
- Colin Greenman and Jong Hoon Bae

Lex Martin will fill in for anyone who is absent.

### Exercises

The `ctype` library provides several functions that detect properties of characters. For instance, given any character, the `islower` function determines whether it is a lower-case letter, returning a non-zero (and therefore “truish”) integer if so and the integer zero if not. Similarly, the `isspace` function determines whether its argument is a “whitespace” character (a space, form feed, newline, carriage return, horizontal tab, or vertical tab).

The same library also provides functions that convert lower-case letters into their upper-case equivalents and vice versa. For example, given any lower-case letter, the `toupper` function computes and returns the corresponding upper-case letter.

**Exercise 0000**: Using the functions in the `ctype` library, write and test an `upcase` function that takes a string (that is, a `null`-terminated array of characters) as its argument and overwrites every lower-case letter in that string with its upper-case equivalent.

**Exercise 0001**: Write and test a non-destructive version of the `upcase` function that takes two arguments, the first of which is a string and the second an array of characters equal in capacity to the first argument. The function should fill the array with an upcased copy of the given string, adding a `null` byte — the character ‘\0’ — when the `null` at the end of the given string is encountered.

**Exercise 0010**: Write and test a non-destructive `capitalize` function similar to the function you wrote in the preceding exercise except that it applies `toupper` only to the first letter in the string and to the first letter after each subsequent whitespace character.

**Exercise 0011**: Use the library function `strlen` to determine the length of the string "Hello, world!\n" and the built-in `sizeof` operator to determine how many bytes of storage that same string occupies. Account for any disparity between these results.

A string literal can be used as an initializer, but not on the right-hand side of an assignment expression. To store a copy of a string into an array of characters, one has
to assign the characters one by one into the corresponding positions in the target array. You can write your own for-loop to do this, but it’s easier just to call the library function `strncpy`, which takes three arguments: the character array into which you want to copy the string, the string you want to copy, and the number of characters you want to copy. The function returns a pointer to the target array.

Exercise 0100: How large would an array of characters need to be in order to hold a copy of the string "Hello, world!\n"? Write a program that declares such an array and invokes `strncpy` to put a copy of that string into it, then prints out that array (using the `%s` format control). Should the number of characters specified in the call include a terminating null character? Justify your answer.

The library function `strncat` is used to append a new string at the end of an existing string. It takes three arguments: the target string, the source string, and the number of characters to be appended, and it returns a pointer to the target string. Not infrequently, it is invoked several times in succession, perhaps from inside a loop, in order to collect several substrings and to make one big string out of them.

The `strncat` function works by finding the null character that terminates the target string before it is changed, and then working like `strncpy`, but starting from the position of the null. The programmer is expected to guarantee, as a precondition of the function, that the array of characters containing the target string is large enough to accommodate the entire source string, including the null character that terminates it.

Exercise 0101: Suppose that we have an array of eighty characters called `line`, in which a string of length 73 is currently stored (so that the characters of that string are stored in `line[0]` through `line[72]`, and a terminating null character is stored in `line[73]`. What is the length (as determined by `strlen`) of the longest string that `strncat` can append to `line` without overrunning the end of the array? Draw a diagram showing what the elements of `line` will look like after a string of this length has been appended.

Since `char` is an integer type in which the values can be interpreted as characters and named by character literals, you can use the arithmetic comparison operators `<`, `>`, `<=`, `>=`, and `!=` to compare character values directly. This idea won’t work for strings, though, since writing any of these operators between the names of two arrays will compare the addresses at which the respective first elements of those arrays are stored, not the characters stored in those locations and the ones the follow.

The library function `strcmp` takes any two strings (null-terminated arrays of characters) and returns an integer that is guaranteed to be negative if the first string precedes the second lexicographically, positive if the second string precedes the first, and zero if the two strings are equal. “Lexicographical precedence” is determined by comparing characters in the two strings until a difference is encountered or one of the strings runs out of characters, just as when you look a word up in the dictionary:

- "cat" precedes "dog" because its first letter, c, comes before the first letter of "dog", which is d.
- "cat" precedes "cow" because the first letters match and the second letter of "cat", which is a, comes before the second letter of "cow", which is o.
- "cat" precedes "catch" because the all of the letters of "cat", match the corresponding letters of "catch", but "catch" has additional letters at the end.

Exercise 0110: Write and execute a C program that tests whether "cat" and "CAT" are equal as strings (as determined by `strcmp`) and, if not, which of them lexicographically precedes the other.
The library function `strchr` takes two arguments, a string and a character, and returns the address of the first array element in the string that matches the character, or a special pointer value called NULL if the given character does not occur in the given string at all. (The name NULL is defined in the header file for the `stddef` library.)

The main purposes of the NULL pointer value are (a) to serve as a sentinel indicating the end of some data structure that has an unpredictable size and (b), as in this case, to signal the failure of a search or of some precondition for further processing. It is an error to apply the indirection operator * to a pointer variable whose value is NULL.

Exercise 0111: Using the library function `strchr`, write and test a C function that takes a string and prints out a space-separated list of the positions in that string that are occupied by colons. For instance, given the string "::alpha::beta:gamma:", the function should print out 0 6 7 12 18.

Hints: (a) Position numbers can be computed by pointer subtraction. (b) In a loop, the pointer returned by one call to `strchr` can be the argument to `strchr` in the next iteration. How will you know when to exit from the loop?

The `stdio` standard library offers a bewildering variety of functions for input and output, some of which are well-known troublemakers that often result in buffer-overflow errors. We’ll look at just a few of the most useful and least dangerous ones.

Many traditional C programs want to interact with users by displaying their output in a terminal window and reading their input from the keyboard as the user types it into that window. This seems a bit primitive in some ways, but it provides unexpected flexibility in a lot of cases, since (with the cooperation of the shell) a program that expects input from the keyboard in this way can instead be invoked in such a way that it takes input from a file that has been prepared in advance or even from the output from another program.

The `getchar` function takes no arguments. It reads in and returns one character from “standard input” (the keyboard in a terminal window, or some file or program that is pretending to be the keyboard with the connivance of the shell). The user can press <Ctrl/D> to signal that no more input will be forthcoming; in this case, `getchar` returns a negative integer value defined by the constant `EOF` (“end of file”). The signature for `getchar` specifies that it returns an `int` rather than a `char` so as to make this error indication possible.

Similarly, the `putchar` function takes one argument, which should be a character, and outputs that character to “standard output” (normally, the terminal window). If it is able to complete this operation, it returns the character that it wrote; otherwise, it returns `EOF`.

Exercise 1000: Write and run a C program that uses `getchar` and `putchar` to echo standard input to standard output, replacing each lower-case letter that it reads with the corresponding upper-case letter in the output. Stop echoing when the user of the program presses <Ctrl/D>.

Exercise 1001: You can tell the shell to connect an existing file (rather than the keyboard) to a program’s standard input port by writing a less-than symbol, followed by the name of the file, at the end of the command line. This is called input redirection. Run the program you wrote for the preceding exercise again, this time using input redirection so that it takes its input from the `.c` file containing the program’s source code.

In many applications, it is convenient to read an entire line of input at a time. The author of our textbook provides a useful function called `read_line` for this purpose:
int read_line(char str[], int n) {
    int ch;
    int i = 0;
    while ((ch = getchar()) != '
')
        if (i < n)
            str[i++] = ch;
    str[i] = '\0';
    return i;
}

Exercise 1010: Suppose that we want to be able to read in and store as many as 511 characters from standard input before seeing the first newline character. How large an array of characters would we have to allocate for \texttt{read\_line} to fill up? What is the largest value that we could then provide as the second argument in the call to \texttt{read\_line}? (Note that \texttt{read\_line} reads the newline character and, in effect, removes it from the input buffer, but does not store it in \texttt{str}.)

Exercise 1011: The C standard \texttt{stdio} library provides a somewhat similar function called \texttt{gets}. Read the manual page for this function (type \texttt{man 3 gets} in a terminal window) and then explain why it's better to use \texttt{read\_line}.

Exercise 1100: One defect of the \texttt{read\_line} function shown above is that it does not handle the end-of-file signal gracefully. Revise the function so that, if the call to \texttt{getchar} ever returns \texttt{EOF}, \texttt{read\_line} terminates the string it is currently collecting with a null character and then immediately returns \texttt{EOF} rather than the number of characters read.

Exercise 1101: Revise the program you wrote for exercise 1000 so that it uses \texttt{read\_line}, \texttt{upcase} and the library procedure \texttt{puts} to echo and transform the input line by line rather than character by character.

The library function \texttt{scanf} can be used to extract values from formatted input, in a way that is roughly analogous to the way in which \texttt{printf} inserts values into formatted output.

Exercise 1110: Write a program that uses \texttt{scanf} inside a \texttt{for}-loop to allow the user to supply interactively the elements of an array of fourteen \texttt{int} values.