The primary goal of this section is to be able to use strings.
Strings

There is no built-in C `string` type. The “convention” is that a C string is an **array of characters**, terminated by a **null character**.

```c
char my_string[4] = {'c', 'a', 't', '\0'};
```

The **null character**, also known as a null **terminator**, is a `char` with a value of zero. It is often written as `'\0'` instead of just `0` to improve communication and indicate that a null character is intended.

`'\0'` is equivalent to `0`. That is different from `'0'`, which is equivalent to `48` (the ASCII character for the symbol zero).
String initialization

char arrays also support a double quote (") initialization syntax. When combined with the automatic length declaration ([ ]), the length includes the null terminator.

The following definitions create equivalent 4-character arrays:

```c
char a[] = {'c', 'a', 't', '\0'};
char b[] = {'c', 'a', 't', 0};
char c[4] = {'c', 'a', 't'};
char d[] = { 99, 97, 116, 0};
char e[4] = "cat";
char f[] = "cat";
```

This array initialization notation is different than the double quote notation used in expressions (e.g., in printf("string")).
Null termination

With null terminated strings, we do not need to pass the length to functions. It is determined by the location of the \'\0\'.

```c
// e_count(s) counts the # of e's and E's in string s

int e_count(const char s[]) {
    int count = 0;
    int i = 0;
    while (s[i]) { // not the null terminator
        if ((s[i] == 'e') || (s[i] == 'E')) {
            ++count;
        }
        ++i;
    }
    return count;
}
```

It is good style to have `const` parameters to communicate that no changes (mutation) occurs to the string.
**strlen**

The **string** library (**#include <string.h>**) provides many useful functions for processing strings (more on this library later).

The **strlen** function returns the length of the **string**, not necessarily the length of the **array**. It does **not include** the null character.

```c
int my_strlen(const char s[]) {
    int len = 0;
    while (s[len]) {
        ++len;
    }
    return len;
}
```
Here is an alternative implementation of `my_strlen` that uses pointer arithmetic.

```c
int my_strlen(const char *s) {
    const char *p = s;
    while (*p) {
        ++p;
    }
    return (p-s);
}
```

In practice, pointer notation is often used with strings as it is slightly faster. Using array index notation (`s[i]`) performs an extra addition in the loop.
Lexicographical order

Characters can be easily compared ($c_1 < c_2$) as they are numbers, so the character order is determined by the ASCII table.

If we try to compare two strings ($s_1 < s_2$), C compares their addresses (pointers), which is not helpful.

To compare strings we are typically interested in using a lexicographical order.

Strings require us to be more careful with our terminology, as “smaller than” and “greater than” are ambiguous: are we considering just the length of the string? To avoid this problem we use precedes ("before") and follows ("after").
To compare two strings using a lexicographical order, we first compare the first character of each string. If they are different, the string with the smaller first character precedes the other string. Otherwise (the first characters are the same), the second characters are compared, and so on.

If the end of one string is encountered, it precedes the other string. Two strings are equal (the same) if they are the same length and all of their characters are identical.

The following strings are in lexicographical order:

"" "a" "az" "c" "cab" "cabin" "cat" "catastrophe"
The `<string.h>` library function `strcmp` uses lexicographical ordering.

`strcmp(s1, s2)` returns zero if the strings are identical. If `s1` precedes `s2`, it returns a negative integer. Otherwise (`s1` follows `s2`) it returns a positive integer.

```c
int my_strcmp(const char s1[], const char s2[]) {
    int i = 0;
    while (s1[i] == s2[i]) {
        if ((s1[i] == '\0') && (s2[i] == '\0')) return 0;
        ++i;
    }
    if (s1[i] < s2[i]) return -1;
    return 1;
}
```
To compare if two strings are *equal* (identical), use the `strcmp` function and check for zero (false).

```c
char a[] = "the same?";
char b[] = "the same?";
char c[] = "different";

trace_bool(strcmp(a, b) == 0);
trace_bool(!strcmp(a, b));
trace_bool(!strcmp(a, c));
```

`strcmp(a, b) == 0` => true

`!strcmp(a, b)` => true

`!strcmp(a, c)` => false

Never use the equality operator (==) to compare strings. It compares the *addresses* of the strings, not their contents.
Lexicographical orders can be used to compare (and sort) any sequence of elements (arrays, lists, ...) and not just strings.

The following Racket function lexicographically compares two lists of numbers:

```racket
(define (lon<=? lon1 lon2)
  (cond [(empty? lon1) #t]
       [(empty? lon2) #f]
       [(< (first lon1) (first lon2)) #t]
       [(< (first lon2) (first lon1)) #f]
       [else (lon<=? (rest lon1) (rest lon2))])))

(lon<=? '(4 9 1 2 1) '(4 5 9)) ; => #f
(lon<=? '(4 3) '(4 3 2)) ; => #t
```
String I/O

The `printf` placeholder for strings is `%s`.

```c
char a[] = "cat";
printf("the %s in the hat\n", a);
```

`printf` prints out characters until the null character is encountered.

`printf` does not print out the null character.
When using `\%s` with `scanf`, it stops reading the string when a “white space” character is encountered (e.g., a space or `\n`).

`scanf("\%s", ...)` is useful for reading in one “word” at a time.

```c
char name[81];
printf("What is your first name?\n");
scanf("\%s", name);
```

You must be very careful to reserve enough space for the string to be read in, and **do not forget the null character**.

`scanf("\%s", ...)` automatically adds the null character.
example: understanding scanf

char name[10] = {0};
while (scanf("%s", name) == 1) {
    printf("Hello, %s!\n", name);
}

The input:

Samantha Bob [EOF]

Generates the following output:

Hello, Samantha!
Hello, Bob!

Afterward, what is stored in the name array?

B o b \0 n t h a \0 \0
In the following example, the name array is 81 characters and can accommodate first names with a length of up to 80 characters.

```c
char name[81];
printf("What is your first name?\n");
scanf("%s", name);
```

What if someone has a really long first name?
int main(void) {
    char command[8];
    int balance = 0;
    while (1) {
        printf("Command? ('balance', 'deposit', or 'q' to quit): ");
        scanf("%s", command);
        if (!strcmp(command, "balance")) {
            printf("Your balance is: %d\n", balance);
        } else if (!strcmp(command, "deposit")) {
            printf("Enter your deposit amount: ");
            int dep;
            scanf("%d", &dep);
            balance += dep;
        } else if (!strcmp(command, "q")) {
            printf("Bye!\n"); break;
        } else {
            printf("Invalid command. Please try again.\n");
        }
    }
}
In this banking example, entering a long command causes C to write characters beyond the length of the command array. Eventually, it overwrites the memory where balance is stored.

This is known as a buffer overrun (or buffer overflow). The C language is especially susceptible to buffer overruns, which can cause serious stability and security problems.

In this introductory course, having an array with an appropriate length and using scanf is “good enough”.

In practice you would never use this insecure method for reading in a string.
If you need to read in a string that includes whitespace until a newline (\n) is encountered, the `gets` function can be used (CP:AMA 13.3).

It is also very susceptible to overruns, but is convenient to use in this course.

```c
char name[81];
printf("What is your full name?\n");
gets(name);
```
There are C library functions that are more secure than `scanf` and `gets`.

One popular strategy to avoid overruns is to only read in one character at a time (e.g., with `scanf("%c")` or `getchar`). For an example of using `getchar` to avoid overruns, see CP:AMA 13.3.
Two additional `<string.h>` library functions that are useful, but susceptible to buffer overruns are:

`strcpy(char *dest, const char *src)` overwrites the contents of `dest` with the contents of `src`.

`strcat(char *dest, const char *src)` copies (appends or concatenates) `src` to the end of `dest`.

You should always ensure that the `dest` array is large enough (and don’t forget the null terminator).
Consider this simple implementation of `my_strcpy`:

```c
char *my_strcpy(char *dst, const char *src) {
    char *d = dst;
    while (*src) {
        *d = *src;
        ++d; ++src;
    }
    *d = '\0';
    return dst;
}
```

with the following function call:

```c
char s[9] = "spam";
my_strcpy(s + 4, s);
```

The null terminator of `src` is overwritten, so it will continue to fill up memory with `spamspamspam...` until a crash occurs.
While \textit{writing} to a buffer can cause dangerous buffer overruns, \textit{reading} an improperly terminated string can also cause problems.

```c
char c[3] = "cat"; // NOT properly terminated!
printf("%s\n", c);
printf("The length of c is: %d\n", strlen(c));
```

The string library has “safer” versions of many of the functions that stop when a maximum number of characters is reached.

For example, \texttt{strnlen}, \texttt{strncmp}, \texttt{strncpy} and \texttt{strncat}. 
String literals

C strings in quotations (e.g., "string") that are in an expression (i.e., not part of an array initialization) are known as string literals.

printf("literal\n");

printf("literal %s\n", "another literal");

if (!strcmp(s, "literal")) ...

strcpy(dst, "literal");

int i = strlen("literal");

scanf("%d", &i);
String literal storage

Where are string literals stored?

For each string literal, a null-terminated const char array is created in the read-only data section.

In the code, the occurrence of the string literal is replaced with address of the corresponding array.

The “read-only” section is also known as the “literal pool”.
void foo(int i, int j) {
    printf("i = %d\n", i);
    printf("the value of j is %d\n", j);
}

Although no name is actually given to each literal, it is helpful to imagine that one is:

const char string_literal_1[] = "i = %d\n";
const char string_literal_2[] = "the value of j is %d\n";

void foo(int i, int j) {
    printf(string_literal_1, i);
    printf(string_literal_2, j);
}

You should not try to modify a string literal. The behaviour is undefined, and it causes an error in Seashell.
Note the subtle difference between the following two definitions:

```c
int main(void) {
    char a[] = "mutable char array";
    char *p = "constant string literal";
    //...
}
```

Once again, it is helpful to think of the string literal as a separately defined `const char array`.

```c
const char string_literal_1[] = "constant string literal";
```

```c
int main(void) {
    char a[] = "mutable char array";
    char *p = string_literal_1;
    //...
}
```
Arrays vs. pointers

Earlier, we said arrays and pointers are *similar* but different.

Consider again two similar string definitions:

```c
void f(void) {
    char a[] = "pointers are not arrays";
    char *p = "pointers are not arrays";
    ...
}
```

- The first reserves space for an initialized 24 character array (`a`) in the stack frame (24 bytes).
- The second reserves space for a `char` pointer (`p`) in the stack frame (8 bytes), *initialized* to point at a string literal (`const char array`) created in the read-only data section.
example: more arrays vs. pointers

```c
char a[] = "pointers are not arrays";
char *p = "pointers are not arrays";
char d[] = "different string";
```

a is a char array. The identifier a has a constant value (the address of the array), but the elements of a can be changed.

```c
a = d; // INVALID
a[0] = 'P'; // VALID
```

p is a char pointer. p is initialized to point at a string literal, but p can be changed to point at any char.

```c
p[0] = 'P'; // INVALID (p points at a const literal)
p = d; // VALID
p[0] = 'D'; // NOW VALID (p points at d)
```
An array is more similar to a `constant` pointer (that cannot change what it “points at”).

```c
int a[6] = {4, 8, 15, 16, 23, 42};
int * const p = a;
```

In most practical expressions `a` and `p` would be equivalent. The only significant differences between them are:

- `a` has the same value as `&a`, while `p` and `&p` have different values
- The size of `a` is 24 bytes, while `sizeof(p)` is 8
C running times (strings & I/O)

<string.h> functions (e.g., strlen, strcpy) are $O(n)$, where $n$ is the length of the string. For strcmp, $n$ is the length of the smallest string.

<stdio.h> functions printf and scanf are $O(1)$, except when working with strings ("%s"), which are $O(n)$, where $n$ is the length of the string.

Note that the string literal used with printf must always be constant length (i.e., printf("literal")).
Do **NOT** put the `strlen` function within a loop.

```c
int char_count(char c, char *s) {
    int count = 0;
    for (int i=0; i < strlen(s); ++i) {  // BAD !!!!!
        if (s[i] == c) ++count;
    }
    return count;
}
```

By using an $O(n)$ function (`strlen`) inside of the loop, the function becomes $O(n^2)$ instead of $O(n)$.

Unfortunately, this mistake is common amongst beginners.

This will be harshly penalized on assignments & exams.
Arrays of Strings

An array of strings can be defined as a 2D array of `char`s, but it requires that each string is allocated the same fixed number of `char`s (regardless of the actual string length).

```c
char aos[][21] = {"my", "two dimensional", "char array"};
```

This is awkward because a function would need to know the fixed length in advance.

```c
void aos_function(char aos[][21], int num_strings) { ... }
```

This method is rarely used.

It is possible to send the fixed length as a separate parameter if the array is “re-interpreted” (cast) to a 1D array.
Instead, an array of pointers is more often used.

```c
char *aos[] = {"my awesome array", "of string", "literals"};
```

In the above example, `aos` is an array of pointers, with each pointer pointing to a string literal.

```c
// equivalent definition

const char str_lit_0[] = "my awesome array";
const char str_lit_1[] = "of string";
const char str_lit_2[] = "literals";

char *aos[] = {str_lit_0, str_lit_1, str_lit_2};
```

This array of pointers can be passed to a function, but as with all arrays, you must still pass the array length:

```c
void aos_function(char *aos[], int num_strings) { ... } // OR
void aos_function(char **aos, int num_strings) { ... }
```
Until we learn how to use dynamic memory, defining an array of *mutable* strings is a little more awkward.

```c
char s0[] = "my mutable array";
char s1[] = "of strings";
char *aos[] = {s0, s1};
```

With an array of pointers to strings, you can access any `char` as if it was a 2D array of `chars`.

For example, `aos[0][1]` is `(aos[0])[1]`, which is `'y'`. 
Goals of this Section

At the end of this section, you should be able to:

- define and initialize strings
- explain and demonstrate the use of the null termination convention for strings
- explain string literals and the difference between defining a string array and a string pointer
- sort a string or sequence lexicographically
• use I/O with strings and explain the consequences of buffer overruns

• use `<string.h>` library functions (when provided with a well documented interface)