cstring = array of characters terminated by a zero byte (old school strings from C programming language)

\[
\text{char } \text{example}[8];
\]

- **type**: char
- **name**: example
- **size**: 8

```
value: 'c' 's' 't' 'r' 'i' 'n' 'g' '\0'
index: 0 1 2 3 4 5 6 7
```

- example[0] = 'c'
- example[7] = '\0'

**size**: number of elements of the cstring. This must be a constant positive integer: a literal value (e.g. 5, 10) or a const variable (e.g. const int N = 10). Just like arrays, because it is one!

**index**: location of in the array. Note the index range is from 0 to 7 (0 to number of elements - 1). For a valid cstring, the last index should always contain a zero-byte (i.e. '\0')

**why?**: many libraries are written in C, and can’t use std::string, so you’ll need to use cstrings instead
declaring and initializing cstrings

char x[5];

char x[5] = {'p','e','a','r','\0'};

char x[5] = "pear";

tricky

char x[4] = {'p', 'e', 'a', 'r'};

char x[6] = {'p', 'e', 'a', 'r'};

char x[] = "pear";

You don’t need to specify the size

char x[7] = "pear";

char x[4] = "pear";

int N = 4; char x[N] = "pea";

const int N = 4; char x[N] = "pea";

const double N = 4; char x[N] = "p";

ERROR! The size is too small (we need space for the zero-byte). You’ll see an error like this “initializer-string for array of chars is too long”

ERROR! The size is not const (just like arrays)

does the same as the prior line. Notice that it automatically adds the zero byte and you don’t have to explicitly write it (don’t do “pear\0”)

note: generally i won’t use quote marks in pictured arrays for ease of readability

DECLares an array of 5 chars (or a cstring of length 5)

DECLares a cstring of length 5 and initializes the elements to ‘p’, ‘e’, ‘a’, ‘r’ and the zero byte ‘\0’

DECLares a cstring of length 5 and initializes the elements to ‘p’, ‘e’, ‘a’, ‘r’, and ‘\0’

DECLares a cstring of size 4 and initializes it to “pea” (same as saying the elements were initialized to ‘p’, ’e’, ’a’, and ’\0’.)

DECLares a string of size 7 and fills the first 5 slots with ‘p’,’e’,’a’,’r’, and ‘\0’

DECLares a cstring of size 4 and initializes it to “pea” (same as saying the elements were initialized to ‘p’, ’e’, ’a’, and ’\0’.)

DECLares a cstring of size 4 and initializes it to “pea” (same as saying the elements were initialized to ‘p’, ’e’, ’a’, and ’\0’.)

DECLares an array of 4 chars and initializes the elements to ‘p’, ‘e’, ‘a’, ‘r’, but it is NOT A VALID CSTRING since it doesn’t terminate with a zero-byte

DECLares an array of 6 chars and initializes the first four elements to ‘p’, ‘e’, ‘a’, ‘r’, but it is NOT A VALID CSTRING since we have no idea what comes after the r

ERROR! The size must be an integer type (int, etc.)
cstrings + functions (brief)

example function header for sum function with an array input

```
int capitalize(char a[])
```

what? no n? cstrings don’t know their own size...but they are all terminated by the zero-byte (‘\0’), so with this knowledge we can determine where the end of the cstring is (or it’s length)

**cstrings as input arguments:** always passed by reference, (just like arrays) which means:

1. you can change a in the function and when it finishes a will keep those changes
2. to protect a from being changed, define the input argument as const (i.e. “const int a[]”)

```
if a is declared as const,
    cout << a[5];
    okay
    a[5] = 10;
    compile error
```

to call the function: use the cstring name (no extra brackets): capitalize(a);

capitalize(a[3]); compile error, since a[3] is a char and not a cstring

**tricky:** just like with reference parameters, you must pass in a variable the cstring is stored in

```
char s[] = “hello”;
capitalize(“hello”)
capitalize(s);
```

**can my function return a cstring?** nope! but you can modify an input cstring in place
### cstring library: strlen, strcmp, strcpy, strcat

<table>
<thead>
<tr>
<th>Statements</th>
<th>Output?</th>
</tr>
</thead>
<tbody>
<tr>
<td>strlen(a)</td>
<td>5</td>
</tr>
<tr>
<td>strlen(b)</td>
<td>4</td>
</tr>
<tr>
<td>strcmp(a, b)</td>
<td>-1</td>
</tr>
<tr>
<td>strcmp(a, a)</td>
<td>0</td>
</tr>
<tr>
<td>strcpy(b, a)</td>
<td>none, but</td>
</tr>
<tr>
<td></td>
<td>b = [A,p,p,l,e,\0,?,?,?,?]</td>
</tr>
<tr>
<td>strcpy(a, b)</td>
<td>none, but</td>
</tr>
<tr>
<td></td>
<td>a = [P,e,a,r,\0,?,?,?]</td>
</tr>
<tr>
<td>strcat(b, a)</td>
<td>none, but</td>
</tr>
<tr>
<td></td>
<td>b = [P,e,a,r,A,p,p,l,e,\0]</td>
</tr>
<tr>
<td>strcat(a, b)</td>
<td>Error: undefined behavior, but</td>
</tr>
<tr>
<td></td>
<td>a = [A,p,p,l,e,P,e,a]???</td>
</tr>
</tbody>
</table>

| a is cstring of length 7 | b is an cstring of length 10 |

### what’s happening?

- `int strlen(const char a[])` returns the length of `a`, which is the number of chars until the zero-byte.
- `int strcmp(const char a[], const char b[])` returns 0 if `a == b`, -1 if `a[i] < b[i]`, and 1 if `a[i] > b[i]`, where `i` is the index of the first different character.

- `void strcpy(char dst[], char src[])` writes the chars in cstring `src` to cstring `dst` until past the zero-byte (that zero-byte is copied over as well).

- `void strcat(char dst[], char src[])` writes the chars in cstring `src` to cstring `dst` (starting at the zero-byte in `dst`) until past the zero-byte in `src` (that zero-byte is copied over as well).

- `a` is not large enough to hold all of `b`, so it continues writing in memory, but who knows what other variables this could mess up.

**Warning:** The prototypes shown here for these library functions are not the official ones. Those involve pointers, a topic we’ll covered later in this course. For practical purposes (e.g. your projects), this prototype is mostly equivalent.
```cpp
#include <iostream>
using namespace std;

void strrep(char out[], char in[], int n) {
    int oi = 0;
    int ii = 0;
    for (int i = 0; i < n; i++) {
        while (in[ii] != '\0') {
            if (out[oi] == '\0') {
                return;
            }
            out[oi] = in[ii];
            oi++;
            ii++;
        }
        ii = 0;
    }
    out[oi] = '\0';
}

int main() {
    char s[] = "Hi5";
    char d[] = "xxxxxxxxxxxxxxxxx"; // 16 char cstring
    strrep(d, s, 5);
    cout << d << endl;
    strrep(s, d, 1);
    cout << s << endl;
}
```

**output**

```
> Hi5Hi5Hi5Hi5Hi5
> Hi5 (only copies until either s or d terminates)
```
#include <iostream>
using namespace std;

int strcmp(char a[], char b[]) {
    int i = 0;
    while (a[i] != '\0' && b[i] != '\0') {
        if (a[i] < b[i]) {
            return 1;
        } else if (a[i] > b[i]) {
            return -1;
        }
    }
    if (a[i] < b[i]) {
        return 1;
    } else if (a[i] > b[i]) {
        return -1;
    } else {
        return 0;
    }
}

int main() {
    char a[] = "Pear", b[] = "Pears", c[] = "Apple";
    cout << strcmp(a, b) << endl;
    cout << strcmp(a, c) << endl;
    cout << strcmp(b, a) << endl;
    cout << strcmp(b, b) << endl;
}

> -1
> 1
> 1
> 0
```cpp
#include <iostream>
#include <unistd.h> // for the usleep fxn
using namespace std;

void strfly(char a[]) {
    int i = 0;
    int d = 1;
    while (true) {
        a[i] = toupper(a[i]);
        if (i > 0) { // start check
            a[i-1] = tolower(a[i-1]);
        }
        if (a[i] != '\0' && a[i+1] == '\0') { // end check
            a[i+1] = tolower(a[i+1]);
        }
        if (i == 0) {
            d = 1;
        } else if (a[i] == '\0') {
            d = -1;
        }
        i += d;
        cout << a << endl;
        usleep(100000); // 0.1 second
    }
}
```

**goal:** make a cstring fly!

...at each timestep, move forward through a cstring capitalizing each letter and lowercasing everything else.

...and then go backwards!

> Oaxaca
> oAxaca
> oaXaca
> oaxAca
> oaxaCa
> oaxacA
> oaxaCa
> oaxAca
> oaXaca
> ...

**output**

> (give it a shot ;))

---

tocaps & tolower
example: reverse a sentence

Let's start with easier problems:

Let's reverse a cstring

```c
void reverse_chars(char a[]) {
    int i = 0;
    int j = strlen(a) - 1;
    while (i < j) {
        char c = a[i];
        a[i] = a[j];
        a[j] = c;
        i++; j--;
    }
}
```

Cool, but let's modify it by giving positions to start and end at, that way we can use it for parts of a sentence

```c
void reverse_chars(char a[], int i, int j) {
    while (i < j) {
        char c = a[i];
        a[i] = a[j];
        a[j] = c;
        i++; j--;
    }
}
```

Now let's reverse chars in individual words in a sentence by using the fxn we’ve already written!

```c
void reverse_chars_in_words(char a[]) {
    int i1 = 0;
    int i2 = i1 + 1;
    int n = strlen(a);
    while (i2 < n) {
        if (a[i2] == ' ') {
            reverse_chars(a, i1, i2-1);
            i1 = i2 + 1;
        }
        i2++;
    }
    reverse_chars(a, i1, i2-1);
}
```

Lastly, put it to the test!

```c
int main() {
    string s = "Hello everyone happy Friday";
    cout << s << endl;
    reverse_words_in_sentence(s);
    cout << s << endl;
}
```

output

> Hello everyone happy Friday
> Friday happy everyone Hello

before: Hello everyone happy Friday
after:  Friday happy everyone Hello
cstring arrays (or 2-D char arrays)

since a cstring is an array of chars. An array of cstrings is just a 2-D array of chars

```c
char example[3][8];
```

(index): now we have 2 indices to use. The first will choose which cstring to look at, and the second will select the position within a c string. This is similar to choosing `[row][column]` in a matrix of characters

<table>
<thead>
<tr>
<th>index</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>‘p’</td>
</tr>
<tr>
<td>1</td>
<td>‘c’</td>
</tr>
<tr>
<td>2</td>
<td>‘b’</td>
</tr>
<tr>
<td>3</td>
<td>‘e’</td>
</tr>
<tr>
<td>4</td>
<td>‘s’</td>
</tr>
<tr>
<td>5</td>
<td>‘t’</td>
</tr>
<tr>
<td>6</td>
<td>‘a’</td>
</tr>
<tr>
<td>7</td>
<td>‘r’</td>
</tr>
<tr>
<td>8</td>
<td>‘t’</td>
</tr>
<tr>
<td>9</td>
<td>‘r’</td>
</tr>
<tr>
<td>10</td>
<td>‘o’</td>
</tr>
<tr>
<td>11</td>
<td>‘i’</td>
</tr>
<tr>
<td>12</td>
<td>‘n’</td>
</tr>
<tr>
<td>13</td>
<td>‘g’</td>
</tr>
<tr>
<td>14</td>
<td>‘\0’</td>
</tr>
</tbody>
</table>

why?: many libraries are written in C, and can’t use std::string, so you’ll need to use cstrings instead
```cpp
#include <iostream>
using namespace std;

int enumerate(const char a[][10], int n, const char target[]) {
    if (n < 0) {
        return -1;
    }
    int count = 0;
    for (int k = 0; k < n; k++) {
        if (isequal(a[k], target)) {
            count++;
        }
    }
    return count;
}

bool isequal(const char a[], const char b[]) {
    int i = 0;
    while (a[i] != ‘\0’ && b[i] != ‘\0’) {
        if (a[i] != b[i]) { return false; }
    }
    return true;
}

int main() {
    const char fruit[7][10];
    strcpy(fruits[0], “apple”);
    strcpy(fruits[1], “pear”);
    strcpy(fruits[2], “banana”);
    strcpy(fruits[3], “apple”);
    strcpy(fruits[4], “applepie”);
    strcpy(fruits[5], “apple”);
    strcpy(fruits[6], “tomato”);
    char target[] = “apple”;
    int x = enumerate(fruits, 7, target);
    cout << x << endl; // prints 3
}
```