arrays

a data structure that can hold a bunch of items of the same type (e.g. int, double, char, string)

```c
int numbers[8]; // declares an array of 8 integers
```

<table>
<thead>
<tr>
<th>value</th>
<th>index</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>

- **size**: number of elements of the array. This must be a constant positive integer: a literal value (e.g. 5, 10) or a variable that is explicitly declared constant variable (e.g. `const int N = 10`)

- **index**: location of in the array. Note the index range is from 0 to 7 (0 to number of elements - 1)

- **why?**: sometimes you have a lot of data, and ain’t nobody’s got time to write `int x1, x2, x3, x4 ... x100`
int x[5];  
\textit{declares an array of 5 integers}

int x[5] = \{1, 2, 3, 4, 5\};  
\textit{declares an array of 5 integers and initializes the elements to 1, 2, 3, 4, 5}

double scores[6] = \{100, 93, 64, 64, 80, 0\};


\textbf{tricky}

int x[] = \{1, 2, 2, 4\};  
\textit{declares an array of 4 integers and initializes the elements to 1, 2, 3, 4. You don’t need to specify the size here.}

int x[7] = \{1, 2, 2, 4\};  
\textit{declares an array of 7 integers and initializes the fills the first 4 slots with 1, 2, 2, 4. The rest will be 0.}

double x[3] = \{100, 93, 64, 64, 80, 0\};  
\textit{ERROR! The size is too small}

int N = 4; double x[N] = \{1, 2.0, 3.14\};  
\textit{ERROR! The size is not const}

const int N = 4; double x[N] = \{1, 2.0, 3.14\};  
\textit{declares an array of 3 doubles and initializes the elements to 1.0, 2.0, 3.14, 0}

const double N = 4; int x[N] = \{1, 2, 3\};  
\textit{ERROR! The size must be an integer type (int, etc.)}
filling arrays

more often than not you won’t know all of the data at the start

\[
\begin{array}{ccccccccccc}
0 & 1 & 4 & 9 & 16 & 25 & 36 & 49 & 64 & 81 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9
\end{array}
\]

```cpp
#include <iostream>
using namespace std;

int main() {
    int x[10];
    x[0] = 0;
    x[1] = 1;
    x[2] = 4;
    x[3] = 9;
    ...  
    x[9] = 100;  // ERROR!
}
```

```cpp
#include <iostream>
using namespace std;

int main() {
    int x[10];
    int i = 0;
    while (i < 10) {
        cout << "type #: ";
        cin >> x[i];
        cout << endl;
        i++;
    }
}
```

```cpp
#include <iostream>
using namespace std;

int main() {
    int x[10];
    int i = 0;
    while (i < 10) {
        x[i] = i * i;
        i++;
    }
}
```
const int N = 7;
double x[N] = {5, 10, 20, -50, 20, 100, -20};
double sum = 0.0;
for (int i = 0; i < N; i++) {
    sum += x[i];
}  
cout << "sum: " << sum << endl;

const int N = 7;
double x[N] = {5, 10, 20, -50, 20, 100, -20};
double sum = 0.0;
for (int i = 0; i < N; i++) {
    sum += x[i];
    x[i] = sum;
}  
cout << "sum: " << sum << endl;

const int N = 7;
double x[N] = {5, 10, 20, -50, 20, 100, -20};
for (int i = 1; i < N; i++) {
    x[i] = x[i] + x[i-1];
}  
cout << "sum: " << x[N-1] << endl;

what's this code snippet do?

reports the sum of the array: 85

sets x to the following:

also reports the sum of the array: 85

also sets x to the following:
```cpp
#include <iostream>

using namespace std;

const int N = 10;

int main() {
    double expenses[N] = {
        10.00, 1.99, 2.00, 1.50,
        0.50, 2.50, 9.00, 20.00};

    double sum_val = 0.0;
    double mean_val = 0.0;
    double max_val = 0.0;
    double min_val = 0.0;

    for (int i = 0; i < size; i++) {
        sum_val += expenses[i];
        max_val = max(max_val, expenses[i]);
        min_val = min(min_val, expenses[i]);
    }
    mean_val = sum_val / N;
}
```

Where are the bugs?
array access practice

x

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>10</td>
<td>3</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

tax is an integer array of size 7

y

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>-4</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

ty is an integer array of size 10

some statements

\[
x[0] + y[0] \\
y[x[0]] \\
y[x[1] + x[6]] \\
x[y[9]] \\
x[y[x[y[9]]]] \\
x[y[x[0]]] \\
x[x[x[x[1]]]] \\
\]

what's the output?

\[
6 \\
Error \\
7 \\
-4 \\
4 \\
3 \\
2 \\
Error \\
1 \\
0
\]
arrays + functions

example function header for sum function with an array input

```
int sum(int a[], int n)
```

**why the n?** arrays don’t know their own size. Unlike strings, you cannot do a.size() to get the length, so to work with arrays of variable lengths, you’ll need to pass in an additional argument with the array’s size.

**arrays as input arguments:** always passed by reference, 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];     a[5] = 10;
      okay                                                  compile error
```

**to call the function:** use the array name (no extra brackets): `int sum = sum(x, 10);`

```
int sum = sum(x[3], 10);    compile error, since x[3] is an integer and not an integer array
```

**can my function return an array?** nope! but you can modify an input array in place.
sum example

sums all the elements of x.

```c
int sum(int x[], int n) {
    int total = 0;
    for (int i = 0; i < n; i++) {
        total += x[i];
    }
    return total;
}
```
paranoid sum example

sums all the elements of x. returns an empty x with it’s values deleted, so no one else can read them after

```c
int sum(int x[], int n) {
    int total = 0;
    for (int i = 0; i < n; i++) {
        total += x[i];
        x[i] = 0;
    }
    return total;
}
```
How do I search an array for a target integer?

```c
bool find_number(int x[], int n, int target) {
    for (int i = 0; i < n; i++) {
        if (x[i] == target) {
            return true;
        }
    }
    return false;
}
```

What if my array is sorted from small to large? Can I do it faster?

```c
bool find_number(int x[], int n, int target) {
    for (int i = n - 1; i >= 0; i--) {
        if (x[i] == target) { return true; }
        if (x[i] > target) { return false; }
    }
    return false;
}
```

What if I'm looking for a big number in my sorted array?

```c
bool find_number(int x[], int n, int target) {
    for (int i = n - 1; i >= 0; i--) {
        if (x[i] == target) { return true; }
        if (x[i] < target) { return false; }
    }
    return false;
}
```
bool find_number(int x[], int n, int target) {
    int imin = 0;
    int imax = n - 1;

    while (imax >= imid) {
        int imid = (imin + imax) / 2;
        if (x[imid] == target) {
            return true;
        } else if (x[imid] < target) {
            imin = imid + 1;
        } else {
            imax = imid - 1;
        }
    }
    return false;
}

This is called “binary search” -- fastest comparison-based search algorithm in computer science for an array of N elements, it takes only $\log_2(N)$ comparisons to find the target or say it’s not there
given a name, retrieve a person’s phone number:

```c++
int find_number(const string names[], const int numbers[], int n, string name) {
    for (int i = 0; i < n; i++) {
        if (names[i] == name) {
            return numbers[i];
        }
    }
    return -1; // failed to find the number
}
```

if the number isn’t there, ask the user for the phone number (assume the array is larger than num_entries)

```c++
int find_number_v2(string names[], int numbers[], int num_entries, string name) {
    int result = find_number(names, numbers, num_entries, name);
    if (result == -1) {
        cout << "Number not found, please enter a # for " << name << ": " << endl;
        cin >> numbers[n];
        names[n] = name;
    }
    return result;
}
```

**notes:**
1. This will error if the array isn’t big enough to add a new element to it (if num_entries is the same size of the array), then we cannot add new_entries. Hence our assumption.

2. This will error if the input arguments names and numbers are declared const outside of this function, and the user tries to call this function
debug: how to print out arrays

You can’t just do cout << array_name. You must walk through the array and print its elements one by one. We’ll use **function overloading** so we won’t need different names for different data types. These functions can be dropped into your .cpp files and called from main to inspect your arrays.

```cpp
void array_show(const int x[], int n) {
    cout << "int[" << n << "] = [";
    for (int i = 0; i < n; i++) {
        cout << x[i] << ", ";
    }
    cout << "]" << endl;
}

prints integers
```

```cpp
void array_show(const double x[], int n) {
    cout << "double[" << n << "] = [";
    for (int i = 0; i < n; i++) {
        cout << x[i] << ", ";
    }
    cout << endl;
}

prints doubles
```

```cpp
void array_show(const string s[], int n) {
    cout << "string[" << n << "] = [";
    for (int i = 0; i < n; i++) {
        cout << s[i] << ", ";
    }
    cout << endl;
}

prints strings
```
debug: how to test if array elements are equal

You can’t just do array1 == array2. You must walk through the array and test its elements one by one. We’ll use function overloading so we won’t need different names for different data types. These functions can be put into your .cpp files and called from main to check your arrays. This is useful for test cases.

Given a function which tells us the location of the first mismatch between a1 and a2,

```cpp
int differ(const string a1[], int n1, const string a2[], int n2);
```

let’s wrap it with a function to tell us true or false if the strings in an array are the same or not,

```cpp
bool string_array_eq(const string a1[], int n1, const string a2[], int n2) {
    int diff = differ(in, size, out_correct, size);
    if ((diff < n1) || (diff < n2)) {
        cout << “TEST FAILED at location: “ << diff << endl;
    }
}
```

and then let’s test some string arrays!

```cpp
int main() {
    const int size = 4;
    string in[size] = {“david”, “holly”, “jenny”, “ken”};
    string out_correct[size] = {“David”, “Holly”, “Jenny”, “Ken”};

    capitalize_names(in); // function we are testing

    string_array_eq(in, size, out_correct, size);
}
```