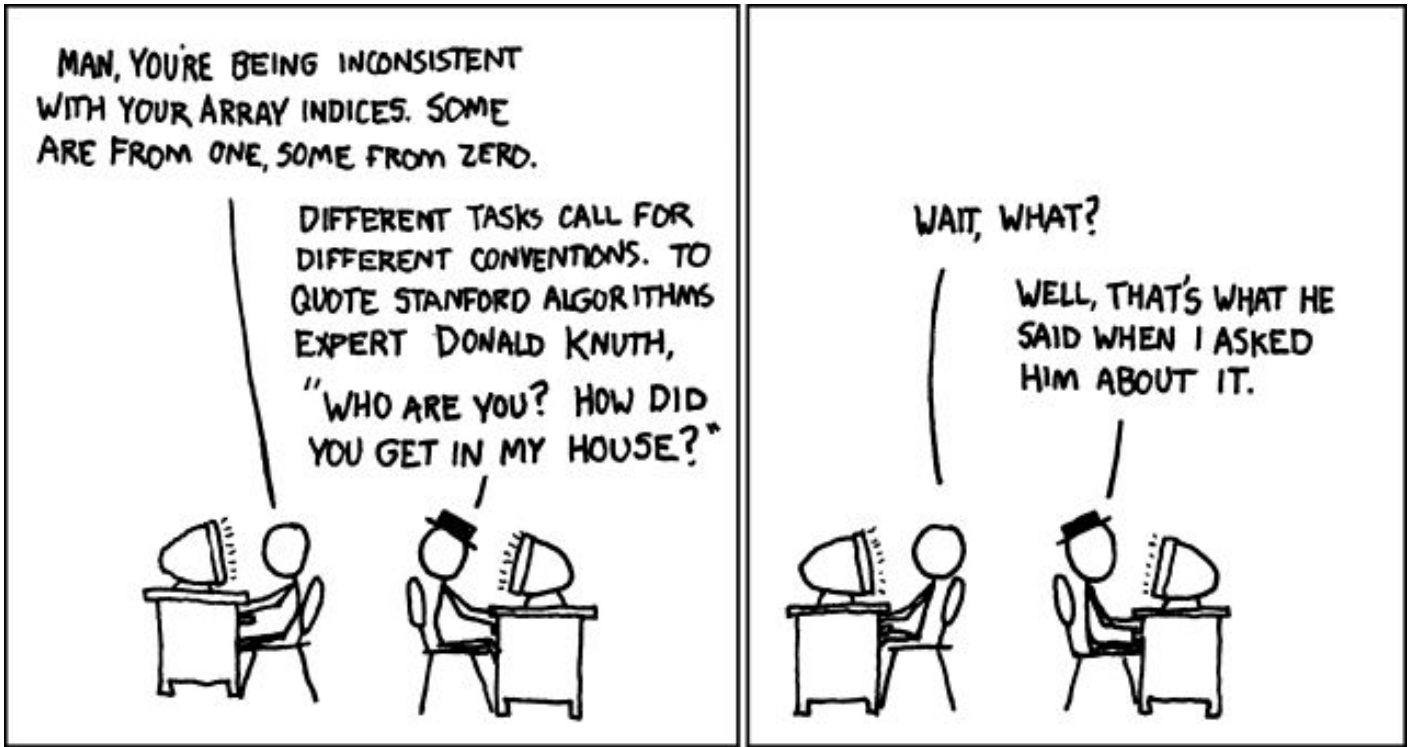
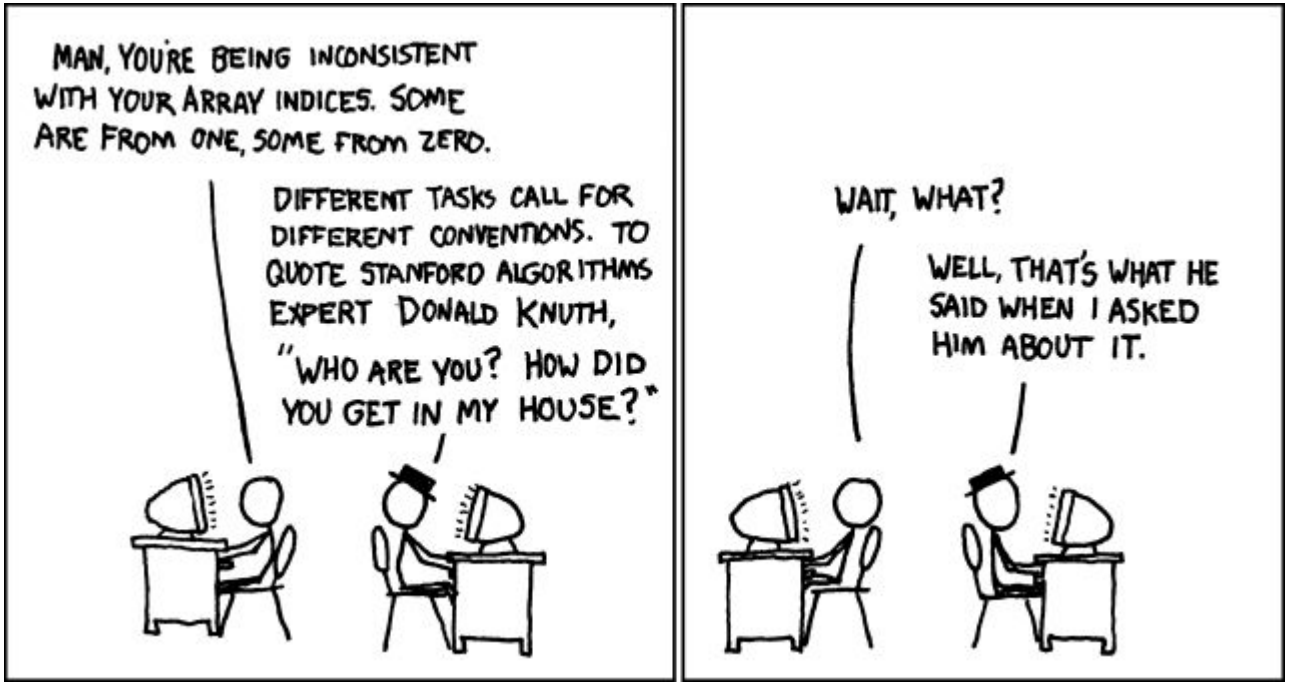
Lecture 18: More Arrays



Announcements and reminders

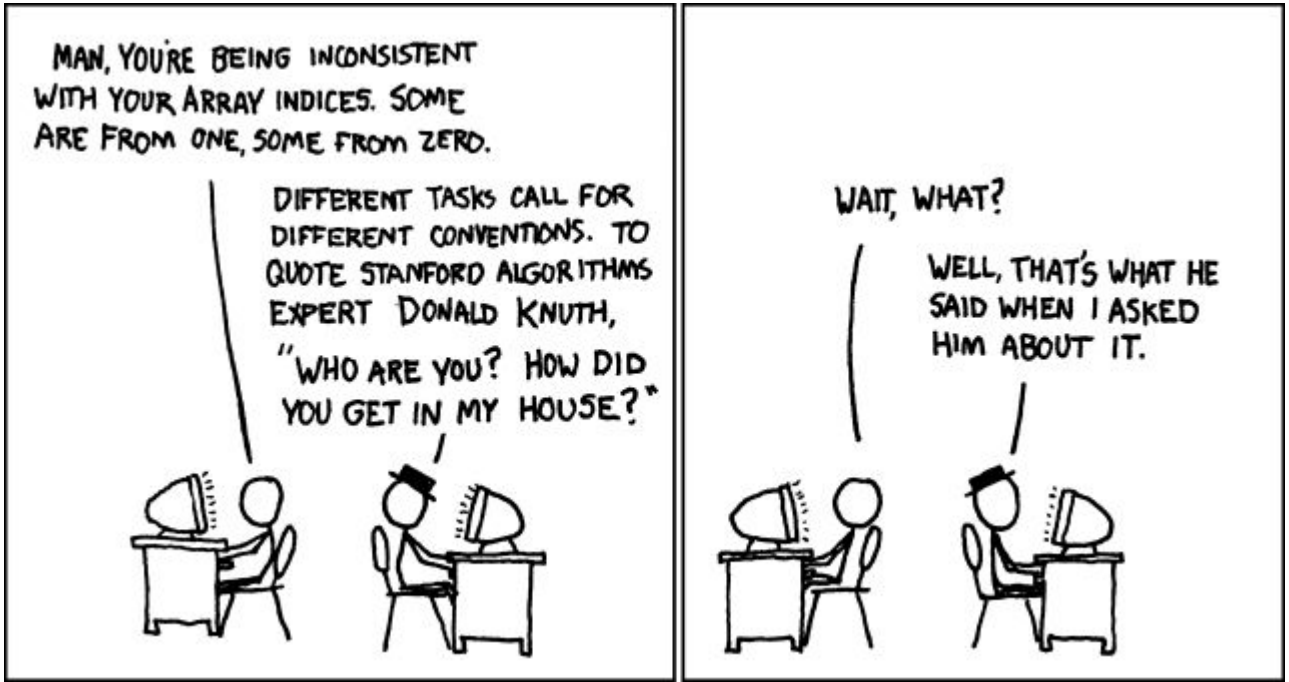
Submissions:

- HW 6 -- due Monday at 6 PM



Last time on *Intro Computing…*

- We saw how to store groups of similar data using **arrays!**
  - … how to declare and define arrays!
  - … how to access data elements within an array!
  - … how to change values in an array!



Arrays as function argument

What does the computer know about an array?

- The data type
- Memory address of the first indexed variable
- The number of indexed variables (size?)

What does ***a function*** know about an array **argument**?

- The data type
- Memory address of the first indexed variable

Formal input parameter argument can be an entire array!

- Argument passed in function call using array name → “array parameter”

Send in size of array as well! You’ll need this

- Typically done using second parameter → `int size` (for example)

Arrays as function argument

What does the computer know about an array?

- The data type
- Memory address of the first indexed variable
- The number of indexed variables (size?)

What does ***a function*** know about an array **argument**?

- The data type
- Memory address of the first indexed variable

Formal input parameter argument can be an entire array!

- Argument passed in function call using array name → “array parameter”

Send in size of array as well! You’ll need this

- Typically done using second parameter → `int size` (for example)

Arrays as function argument

In some main() function definition, consider this call:

```
int score[5], numberOfScores = 5;
fillup(score, numberOfScores);
```

- 1st argument is entire array
- 2nd argument is integer value
- **No brackets on the array argument**

Passing in `score` → provides `fillup()` with **the data type** (int) and **address of `score[0]`**

→ knowing the type helps us retrieve the 2nd-last elements

Passing in `numberOfScores` → provides **size of array**

Arrays as function argument

In some main() function definition, consider this call:

```
int score[5], numberOfScores = 5;

fillup(score, numberOfScores);
```

- Must send in size of array separately
- **Fun fact:** This means we can use the **same function** to fill any sized array! Yehoot!!
  - Exemplifies nice “re-use” properties of functions

```
int score[5], time[10];

fillup(score, 5);

fillup(time, 10);
```

const parameter modifier

Array parameters send in the **memory address** of their 1st element

→ function has access to the location in memory

→ … can modify the array, and mess its values up!

Sometimes you want this, sometimes you don't

If you don't, you can protect your array contents by using the **const** modifier before an array parameter ("constant array parameter")

Example:

```
void addArray(int size, const double A[], const double B[], double C[])
```

→ A and B are arrays that cannot be modified within the addArray function

Example: function definition

```
void addArray(int size,                 // IN size of the arrays
              const double A[],         // IN input array
              const double B[],         // IN input array
              double C[])               // OUT result array

// Takes as input parameters two arrays of same size, and outputs an
// array of same size, whose elements are the sum of the corresponding
// elements in the two input arrays

{
  int i;
  for (i = 0; i < size; i++) {
    C[i] = A[i] + B[i];
  }
}
```

Example: function call

The function addArray could be used as follows:

In main()...

```
double one[6], two[6], three[6];

    // ... intermediate code here should set the values in one and two...

addArray(6, one, two, three);
```

Question: What will this do?

```
addArray(4, one, two, three);
```

Chapter 6: Arrays and Vectors

1. Arrays
2. Common array algorithms
3. Arrays / functions
4. Problem-solving: adapting algorithms
5. Problem-solving: discovering algorithms
6. **2D arrays**
7. Vectors

Two-Dimensional Arrays

If often happens that you want to store collections of values that have a two-dimensional layout

This happens **a lot** in financial or scientific applications

Example: Consider the medal-count from the 2014 Winter Olympic skating competitions:

Country	Gold	Silver	Bronze
Canada	0	3	0
Italy	0	0	1
Germany	0	0	1
Japan	1	0	0
Kazakhstan	0	0	1
Russia	3	1	1
South Koria	0	1	0
United States	1	0	1

Definition: An arrangement of *tabular data* (rows and columns of values) is called a **two-dimensional array**, or a **matrix**.

Two-Dimensional Arrays -- defining them

C++ uses an array with ***two*** subscripts to store a 2D array:

```
const int N_COUNTRIES = 8;
const int N_MEDALS = 3;
int counts[N_COUNTRIES][N_MEDALS];
```

→ `counts` is an array with 8 **rows** (`N_COUNTRIES`)
and 3 **columns** (`N_MEDALS`)

→ Often, it is useful to **sketch** our arrays, and what goes where:

Two-Dimensional Arrays -- initializing them

Just as with 1D arrays, you ***cannot*** change the size of a 2D array once it has been defined

You ***can*** initialize them:

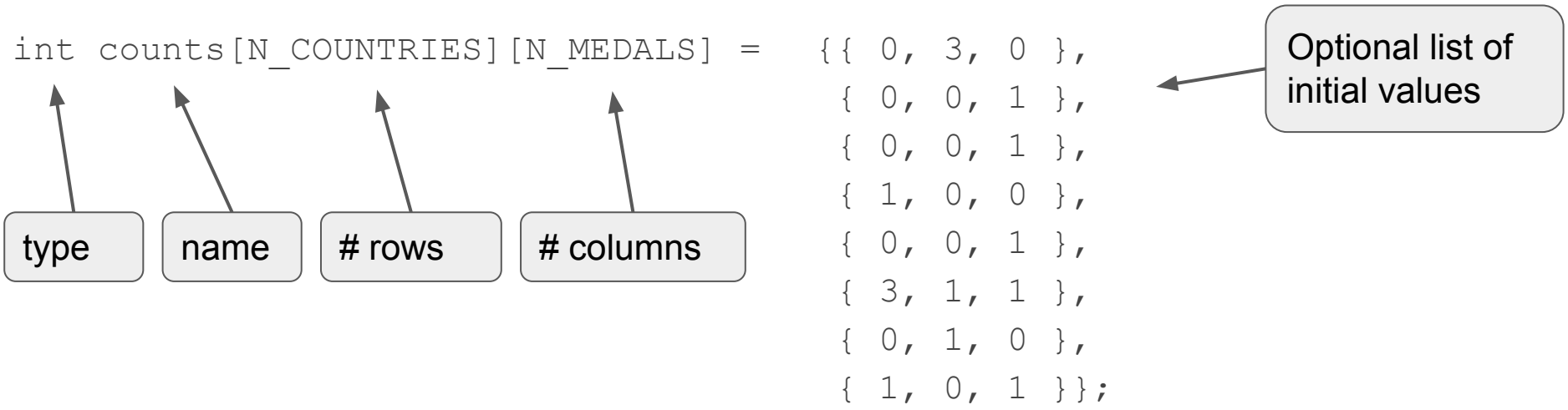
```
int counts[N_COUNTRIES][N_MEDALS] =   {{ 0, 3, 0 },
                                       { 0, 0, 1 },
                                       { 0, 0, 1 },
                                       { 1, 0, 0 },
                                       { 0, 0, 1 },
                                       { 3, 1, 1 },
                                       { 0, 1, 0 },
                                       { 1, 0, 1 }};
```

2D arrays are built up as an array of 1D arrays!
Each **row** is a 1D array.

Two-Dimensional Arrays -- initializing them

Just as with 1D arrays, you ***cannot*** change the size of a 2D array once it has been defined

You ***can*** initialize them:



2D arrays are built up as an array of 1D arrays!
Each **row** is a 1D array.

Two-Dimensional Arrays -- accessing elements

First index gives the row, second index gives the column

Example:

```
int counts[N_COUNTRIES][N_MEDALS] =   {{ 0, 3, 0 },
                                       { 0, 0, 1 },
                                       { 0, 0, 1 },
                                       { 1, 0, 0 },
                                       { 0, 0, 1 },
                                       { 3, 1, 1 },
                                       { 0, 1, 0 },
                                       { 1, 0, 1 }};
```

counts[3][1] = _____

counts[6][0] = _____

Two-Dimensional Arrays -- printing elements

Country	Gold	Silver	Bronze
Canada	0	3	0
Italy	0	0	1
Germany	0	0	1
Japan	1	0	0
Kazakhstan	0	0	1
Russia	3	1	1
South Koria	0	1	0
United States	1	0	1

In order to print each element, we need **two for loops:**

- One to loop over all rows,
- and another to loop over all columns.

Let's **pseudocode** it up!

Two-Dimensional Arrays -- printing elements

Country	Gold	Silver	Bronze
Canada	0	3	0
Italy	0	0	1
Germany	0	0	1
Japan	1	0	0
Kazakhstan	0	0	1
Russia	3	1	1
South Koria	0	1	0
United States	1	0	1

In order to print each element, we need **two for loops:**

- One to loop over all rows,
- and another to loop over all columns.

Let's ~~**pseudo**~~**code** it up!

```
for (int i=0; i < N_COUNTRIES; i++) {
   // process ith row
   for (int j=0; j < N_MEDALS; j++) {
      // process jth column of ith row
      cout << setw(8) << counts[i][j];
   }
   // start new line at end of each row
   cout << endl;
}
```

Multi-dimensional array parameters (2 (or more!) dimensions)

Similar to 1D array:

- 1st dimension size not given (always first)
- … provide this as another input parameter argument to your function
- 2nd dimension size **is given**

Example:

```
void displayPage(const char p[][100], int sizeDim1) {
   for (int i1=0; i1 < sizeDim1; i1++) {
      for (int i2=0; i2 < 100; i2++) {
         cout << p[i1][i2] << endl;
      }
   }
}
```

Multi-dimensional array parameters (2 (or more!) dimensions)

Warning! This function only works for arrays with 100 columns though!

We would need to write a **different function** if we wanted to use it on an array with 101 columns (or any number that isn't 100)

Definition: We would say that the number of columns here is **hardwired** to be equal to 100

Example:

```
void displayPage(const char p[][100], int sizeDim1) {
  for (int i1=0; i1 < sizeDim1; i1++) {
    for (int i2=0; i2 < 100; i2++) {
      cout << p[i1][i2] << endl;
    }
  }
}
```

Omitting column size of 2D array parameter

When passing a 1D array to a function, you specify the size of the array as a separate parameter variable:

```
void print(double values[], int size)
```

This function can print 1D arrays of any size.

However, for 2D arrays, you **can't** simply pass the number of rows ***and*** number of columns as parameter variables:

```
void print(double values[][], int rows, int cols)
```

← **Won't work!**

```
const int COLS = 3;
void print(double values[][COLS], int rows)
```

← **Will work!**

This function can print tables with any number of rows, but the column size (3) is **fixed**.

2D array storage

```
int counts[N_COUNTRIES][N_MEDALS] =    {{ 0, 3, 0 },
                                        { 0, 0, 1 },
                                        { 0, 0, 1 },
                                        { 1, 0, 0 },
                                        { 0, 0, 1 },
                                        { 3, 1, 1 },
                                        { 0, 1, 0 },
                                        { 1, 0, 1 }};
```

What is the reason behind this?

→ Although array *appears* to be 2D, elements are actually stored as a linear sequence

→ `counts` is stored as a sequence of **rows**, each 3 long

2D array storage

```
int counts[N_COUNTRIES][N_MEDALS] =    {{ 0, 3, 0 },
                                        { 0, 0, 1 },
                                        { 0, 0, 1 },
                                        { 1, 0, 0 },
                                        { 0, 0, 1 },
                                        { 3, 1, 1 },
                                        { 0, 1, 0 },
                                        { 1, 0, 1 }};
```

Fun fact!

C++ stores an array as a sequence of its ***rows***, making C++ is a **row-major** language.

Other languages (e.g., Fortran) are **column-major**, in that they store arrays as a sequence of their ***columns***.

What is the reason behind this?

→ Although array *appears* to be 2D, elements are actually stored as a linear sequence

→ `counts` is stored as a sequence of **rows**, each 3 long

2D array storage

```
int counts[N_COUNTRIES][N_MEDALS] =    {{ 0, 3, 0 },
                                        { 0, 0, 1 },
                                        { 0, 0, 1 },
                                        { 1, 0, 0 },
                                        { 0, 0, 1 },
                                        { 3, 1, 1 },
                                        { 0, 1, 0 },
                                        { 1, 0, 1 }};
```

Question: So… where is `counts[3][1]`?

Answer: The ***offset*** (calculated by the compiler) from the start of the array is

$$3 \times \textit{number of columns} + 1$$

counts =

…

Row 0 Row 1 Row 2 Row 3

2D array storage

```
int counts[N_COUNTRIES][N_MEDALS] =    {{ 0, 3, 0 },
                                        { 0, 0, 1 },
                                        { 0, 0, 1 },
                                        { 1, 0, 0 },
                                        { 0, 0, 1 },
                                        { 3, 1, 1 },
                                        { 0, 1, 0 },
                                        { 1, 0, 1 }};
```

Question: So… where is `counts[i][j]`? (**in general**)

Answer: The ***offset*** (calculated by the compiler) from the start of the array is

`i` × *`number of columns`* + `j`

`counts` =

…

Row 0 Row 1 Row 2 Row 3

2D array parameters -- rows

The `row_total` function does not need to know the number of rows of the array, in order to add up the contents of a given `row`:

```
const int COLUMNS = 3;

int row_total(int table[][COLUMNS], int row) {
    int total = 0;
    for (int j = 0; j < COLUMNS; j++) {
        total = total + table[row][j];
    }
    return total;
}
```

2D array parameters -- rows

But what about a function that needs to know the number of rows?

→ Just pass it in as another parameter argument!

```
const int COLUMNS = 3;

int column_total(int table[][COLUMNS], int rows, int col) {
    int total = 0;
    for (int i = 0; i < rows; i++) {
        total = total + table[i][col];
    }
    return total;
}
```

Two-Dimensional Arrays -- calculate some stuff

We have two 2D array functions:

- `row_total` -- add up the contents of a given row
- `column_total` -- add up the contents of a given column

Country	Gold	Silver	Bronze
Canada	0	3	0
Italy	0	0	1
Germany	0	0	1
Japan	1	0	0
Kazakhstan	0	0	1
Russia	3	1	1
South Korea	0	1	0
United States	1	0	1

Questions:

What should we do to calculate the total number of events?

What should we do to calculate how many medals were won by Japan?

Practice it! 2D array parameters

Example: Insert the missing statement.

```
/* Function to add up all of the elements in the first row in
 * the input integer array, and return their sum.
 */
int add_first_row(int array[][MAX_COLS], int n_rows, int n_cols)
{
    int sum = 0;
    for (int k = 0; k < cols; k++) {
        sum = sum + ____________________________ ;
    }
    return sum;
}
```

Practice it! 2D array parameters

Example: Insert the missing statement.

```
/* Function to add up all of the elements in the first row in
 * the input integer array, and return their sum.
 */
int add_first_row(int array[][MAX_COLS], int n_rows, int n_cols)
{
    int sum = 0;
    for (int k = 0; k < cols; k++) {
        sum = sum + array[0][k];        // 0 → first row; kth column
    }
    return sum;
}
```

Arrays -- fixed size can be a drawback

The size of an array **cannot** be changed after it is created.

→ You have to get the size right ***before*** you define an array

→ The compiler needs to know the size in order to build the array,

and functions need to be told number of elements in array, and possibly its capacity (and arrays can't hold more than their initial capacity)

Later, we'll talk about **vectors**, which can have variable size and some other nice flexible features that arrays don't have.

What just happened?! … *a summary*

- Array = collection of “same type” data
- Index of an array element is like that array element’s address
- Indexed variables of array are used like any other simple variable
- for-loop is the “natural” way to traverse arrays (since you know the size)
- Programmer is responsible for staying “**in bounds**” of array
  - You’ll get nasty errors if you don’t!
- Array elements are stored sequentially, in *contiguous* portions of computer memory
  - Only the address of the 1st element of an array is passed into functions
- Constant array parameters are useful to prevent accidental modification of array contents
- Multi-dimensional arrays → create an “array of arrays”