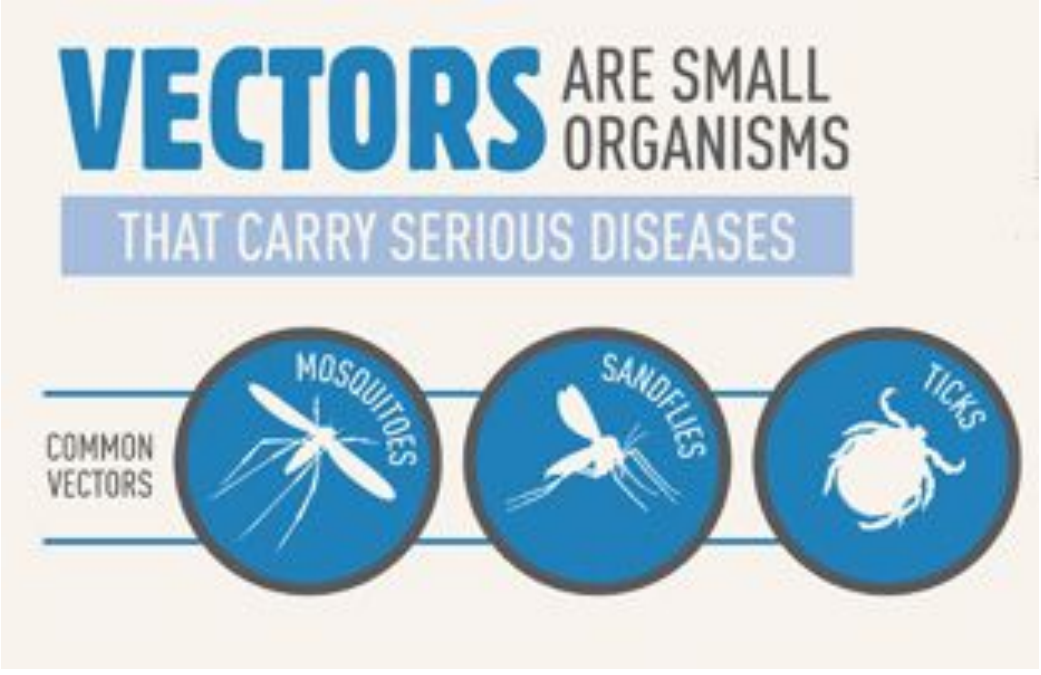
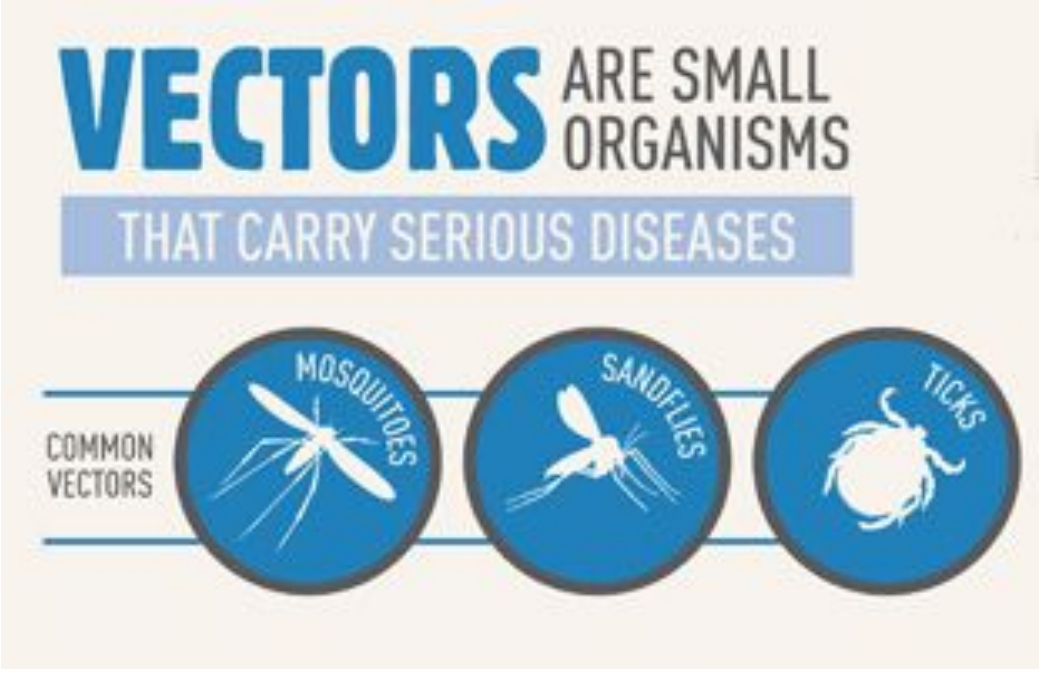
Lecture 27: Vectors



Announcements and reminders

- Project 2 posted, due **Saturday March 23 by 6 PM**
  (no early bonus aside from being a badass)
- Practicum 2 practice problems up ← **do them**
  - Both **MCQ** + **Coderunner**
  - Same rules as Practicum 1 (cheat sheet, Cloud9, etc…)
  - Let your TA know about any conflicts, and **include documentation**
- **Practicum 2: tonight, 5:30p**



Last time on *Intro Computing…*

We saw…

- What is this **object-oriented programming** you speak of?
- What an ***object*** and a ***class*** is!
- What makes up a class and what is **encapsulation**!
  - Member functions (**public interface**)
  - Data members (private data - encapsulated)
- Different types of member functions
  - Getters (accessors) and setters (mutators)

Arrays… some drawbacks

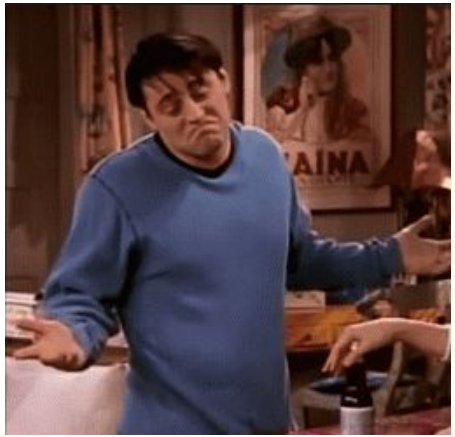
The size of an array **cannot be changed** after it is created

→ you need to know the size **before** you define an array

→ any function that takes the array as an input needs the **capacity/size** too

→ wouldn’t it be nice if there were something we could ***dynamically*** reshape?!

Too bad there isn’t. **Lecture over!**



But for real though: vectors.

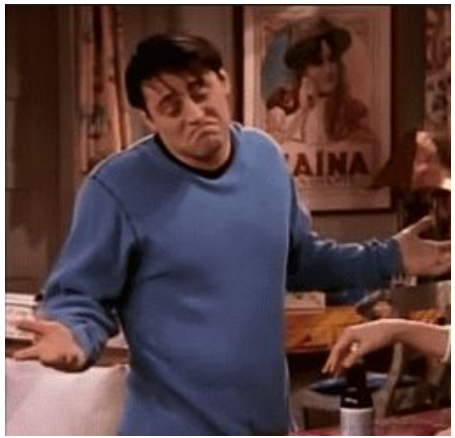
Just kidding! **Vectors** are the answer!

You can think of vectors like a **dynamic array**:

- Not fixed in size when created → **member function: `[vector].size()`**
- Doesn't require an auxiliary variable to track the size
- Can keep adding things to it, taking things out

Super surprising header file:

```
#include <vector>
```



Defining vectors

When you define a vector, you must specify the **type** of the elements in angle brackets:

```
vector<double> data;
```

Default: vector is created **empty**

- Like a string is always initialized to be empty: `string yeet;` // yeet = “”

Similarities to arrays:

- Here, the **data** vector can *only contain doubles*, same way an array (`double array[10]`) could only contain doubles
- Can specify **initial size** in parentheses: `vector<double> data(10);`
- Access elements using **brackets**: `data[i] = 7.0;`

Defining vectors

When you define a vector, you must specify the **type** of the elements in angle brackets:

```
vector<double> data;
```

Default: vector is created **empty**

- Like a string is always initialized to be empty: `string yeet;` // yeet = “”

Similarities to arrays:

- Here, the **data** vector can *only contain doubles*, same way an array (`double array[10]`) could only contain doubles
- Can specify **initial size** in parentheses: `vector<double> data(10);`
- Access elements using **brackets**: `data[i] = 7.0;`

Defining vectors -- some examples

`vector<int> numbers(10);`	
`vector<string> names(3);`	
`vector<double> values;`	
`vector<double> values();`	

`vector<int> numbers(10);`
`for (int i=0; i < numbers.size(); i++)`
`{`
`  numbers[i] = i+1;`
`}`	

`vector<int> numbers;`
`for (int i=1; i <= 10; i++) {`
`  numbers.push_back(i);`
`}`	

Defining vectors -- some examples

`vector<int> numbers(10);`	A vector of 10 integers
`vector<string> names(3);`	A vector of 3 strings
`vector<double> values;`	A vector of size 0 (empty)
`vector<double> values();`	**ERROR:** do not use empty () to create a vector

`vector<int> numbers(10);`
`for (int i=0; i < numbers.size(); i++)`
`{`
`  numbers[i] = i+1;`
`}`
	A vector of 10 integers, filled with 1, 2, 3, … 10
	
	Demonstrating the **.size() member function**

`vector<int> numbers;`
`for (int i=1; i <= 10; i++) {`
`  numbers.push_back(i);`
`}`
	Also a vector of 10 integers, filled with 1, 2, 3, … 10
	
	Demonstrating the **.push_back() member function**

Accessing elements in vectors

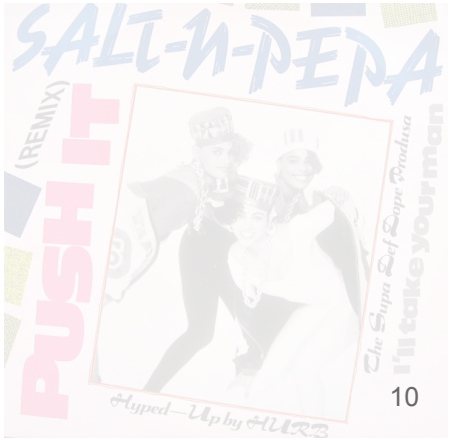
You access elements in a vector the same way as in an array, using an index and brackets:

```
vector<double> values(10);
// display the fourth element
cout << values[3] << endl;
```

But a common error is to attempt to access an element that is not there:

```
vector<double> values;
// display the fourth element
cout << values[3] << endl;
```

So… how do we put values into a vector?



Accessing elements in vectors

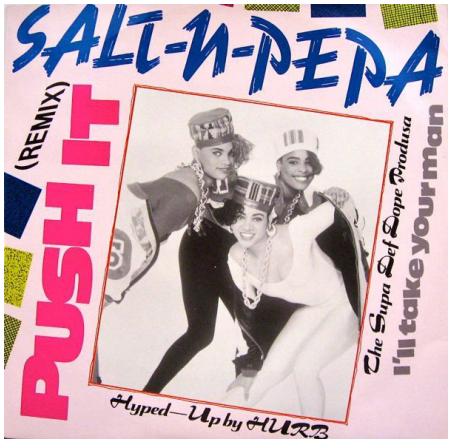
You access elements in a vector the same way as in an array, using an index and brackets:

```
vector<double> values(10);
// display the fourth element
cout << values[3] << endl;
```

But a common error is to attempt to access an element that is not there:

```
vector<double> values;
// display the fourth element
cout << values[3] << endl;
```

So… how do we put values into a vector?



Manipulating elements in vectors

Much like Salt N Pepa, we **push it**

Think of the vector a **stack of papers**

- Starts out empty `vector<int> papers;`
- Then somebody (say, the number 3) arrives
  → they go to the "back" of the line `papers.push_back(3);`
- Then the numbers 5, 1 and 8 arrive, in **that order**
  → they each go to the "back" of the line (or *top* of the stack)

```
papers.push_back(5);
papers.push_back(1);
papers.push_back(8);
```

Check: What now should be the elements of `papers`? `papers.size()`? What order?

Manipulating elements in vectors

We can also **remove elements** from the back: `.pop_back()`

→ removes the **last element** placed into the vector

- Starting with papers = {3, 5, 1, 8} …
- We pick up paper 8 off the stack `papers.pop_back();`

  → `.pop_back()` doesn't need an argument!
  Just removes the **last element**
  (whatever is at the **top** of the stack)

Check: What now should be the elements of `papers`? `papers.size()`? What order?

Manipulating elements in vectors

We can also **remove elements** from the back: `.pop_back()`

→ removes the **last element** placed into the vector

- Starting with papers = {3, 5, 1, 8} …
- We pick up paper 8 off the stack `papers.pop_back();`

  → `.pop_back()` doesn't need an argument!
  Just removes the **last element**
  (whatever is at the **top** of the stack)

→ this is a **last in first out** method

Check: What now should be the elements of `papers`? `papers.size()`? What order?

Manipulating elements in vectors

Example: We can fill vectors from user input.

```
vector<double> values;
double input;
while (cin >> input) {
   values.push_back(input);
}
```

Using vectors

How can we visit every element in a vector?

With arrays, we could do:

```
for (int i=0; i < 10; i++) {
    cout << values[i] << endl;
}
```

Using vectors

How can we visit every element in a vector?

With arrays, we could do:

```
for (int i=0; i < values.size(); i++) {
    cout << values[i] << endl;
}
```

But with vectors, we don't know if 10 is still the current size or not

→ use the **.size()** member function -- returns the current size of the vector

→ all those looping algorithms for arrays work for vectors too! Just use [vector].size()

Vectors as input parameters in functions

How can we pass vectors as parameters to functions?

… in the same way we pass arrays!

But this time there are two cases:

(1) we do ***not*** want to change the values in the vector

(2) we ***do*** want to change the values in the vector

Vectors as input parameters in functions -- without changing the values

Example: Write a function to add up and return the **sum** of all the elements of an input vector of doubles.

Note: this function ***visits*** each vector element but ***does not*** change them.

Vectors as input parameters in functions -- without changing the values

Example: Write a function to add up and return the **sum** of all the elements of an input vector of doubles.

```
double sum(vector<double> values) {
    double total = 0;
    for (int i=0; i < values.size(); i++) {
        total += values[i];
    }
    return total;
}
```

Note: this function ***visits*** each vector element but ***does not*** change them.

Vectors as input parameters in functions -- … and changing the values

Example: Write a function to multiply each element of an input vector of doubles by some factor.

Note: this function ***visits*** each vector element ***and still does not*** change them!

Vectors as input parameters in functions -- … and changing the values

Example: Write a function to multiply each element of an input vector of doubles by some factor.

```
void multiply(vector<double> values, double factor) {
    for (int i=0; i < values.size(); i++) {
        values[i] = values[i] * factor;
    }
}
```

Note: this function ***visits*** each vector element ***and still does not*** change them!

Vectors as input parameters in functions -- … and changing the values

Example: Write a function to multiply each element of an input vector of doubles by some factor.

```
void multiply(vector<double> values, double factor) {
    for (int i=0; i < values.size(); i++) {
        values[i] = values[i] * factor;
    }
}
```

The key with arrays was that we **passed by reference**
→ the function would know where the array is **in memory** and modify it
→ so can we do the same with vectors?

Note: this function ***visits*** each vector element ***and still does not*** change them!

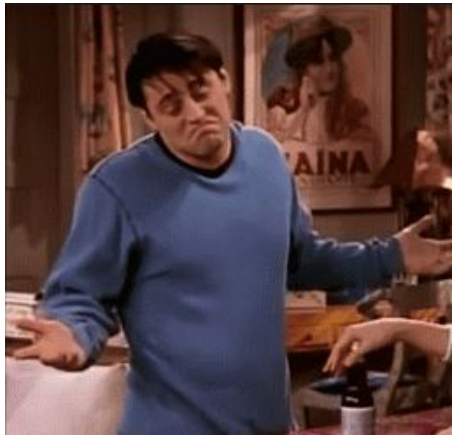
Vectors as input parameters in functions -- … and changing the values

Example: Write a function to multiply each element of an input vector of doubles by some factor.

```
void multiply(vector<double> values, double factor) {
    for (int i=0; i < values.size(); i++) {
        values[i] = values[i] * factor;
    }
}
```

The key with arrays was that we **passed by reference**
→ the function would know where the array is **in memory** and modify it
→ so can we do the same with vectors?

NOPE LECTURE OVER. AGAIN.

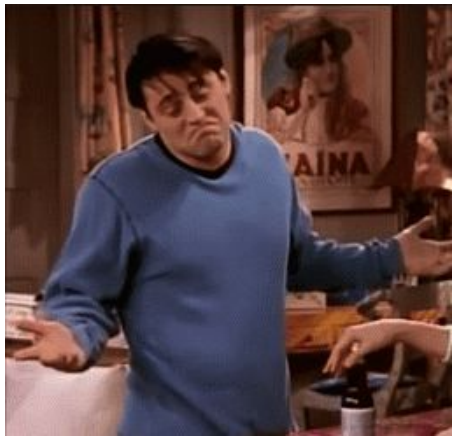


Note: this function ***visits*** each vector element ***and still does not*** change them!

Vectors as input parameters in functions -- … and changing the values

Example: Write a function to multiply each element of an input vector of doubles by some factor.

```
void multiply(vector<double>& values, double factor) {
    for (int i=0; i < values.size(); i++) {
        values[i] = values[i] * factor;
    }
}
```



The key with arrays was that we **passed by reference**
→ the function would know where the array is **in memory** and modify it
→ so can we do the same with vectors?
→ **Yes! Slap a & after the type of variable in the function argument to pass it by reference. Can treat exactly like array input/output!**

Note: this function ***visits*** each vector element ***and totally does*** change them!

Vectors as return values from functions

Example: Write a function that will take **as input** a vector and return a vector that is the values of the input vector, **squared**

Sample input: [0, 1.5, -10, 2.3] → Sample output: [0, 2.25, 100, 5.29]

Note: this function ***returns a vector*** of same size as the input vector (which is unchanged)

Vectors as return values from functions

Example: Write a function that will take **as input** a vector and return a vector that is the values of the input vector, **squared**

Sample input: [0, 1.5, -10, 2.3] → Sample output: [0, 2.25, 100, 5.29]

```
vector<double> square(vector<double> values) {
    vector<double> new_vec;
    for (int i=0; i < values.size(); i++) {
        new_vec.push_back(values[i]*values[i]);
    }
    return new_vec;
}
```

Note: this function ***returns a vector*** of same size as the input vector (which is unchanged)

What just happened?!

We just saw… **vectors!**

- Like a 1d array whose size can change
- Get current size of vector `vec` using: `vec.size()`
- Add elements to the ***back*** using: `vec.push_back()`
- Remove elements from the ***back*** using: `vec.pop_back()`
- Passing vectors into/out of functions
- … and using as return values

Last in first out (LIFO) →