Vectors

Due this week

- **Project 1**
  - Write solutions in VSCode and paste in **Project 1 CodeRunner**.
  - Reach out for any clarifications
- Grading Interviews: October 16th – October 20th

Vectors

Array: Drawbacks

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

- You need to know the size **before** you define an array
  - int array[**5**];
- Any function that takes the array as an input needs the **capacity/size**
  - void print(int array[], **int size**);
- Wouldn't it be nice if there were something we could ***dynamically reshape***?

Vectors

- Vectors are arrays that don’t have a fixed size. It adapts itself to how much you wish to add into it
- Doesn’t require an extra variable to track the size
  - Has vector.size() that we can use
- Useful data structure to use when we have items to add frequently and have items to remove frequently.
- Header file
  - #include<vector>

Declaring vectors

- When you declare 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 empty; // empty = ""
```

Similarities to arrays

- Here, the data vector `(vector<double> data)` can only contain doubles, same way an array `(double array[10])` could only contain doubles.

- double counts[10];
- vector<double> counts;

Vectors declaration

`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);`	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**

Vectors initialization

- We can also initialize vectors like we have initialized arrays:

```
vector<int> your_money = { 0, 18, 7, 43, 4 };
```

- … is equivalent to…

```
vector<int> your_money;
your_money.push_back(0);
your_money.push_back(18);
your_money.push_back(7);
your_money.push_back(43);
your_money.push_back(4);
```

Declaration vs Initialization

- Well almost the same terms, but the difference lies in whether we put in initial values in a vector or not!
- `vector<int> v;` // creates an empty vector
- `vector<int> v(5);` // creates a vector of 5 elements
- `vector<int> v(5, 1);` // creates a vector of 5 elements, each element being a 1.
- `vector<int> v = {1,1,1,1,1};`

Let’s test ourselves

- vector<int> v(7) = {1,2,3,4,5,6,7};

  A. This is a vector declaration.
  B. The size of the vector is 7
  C. This is a vector initialization
  D. It fails to compile

Let’s test ourselves

- vector<int> v(7);
  v = {1,2,3,4,5,6};

  A. This is a vector declaration.
  B. The size of the vector is 7
  C. This is a vector initialization
  D. It fails to compile

Accessing elements in a vector

- 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(2);
// display the fourth element
cout << values[3] << endl;
```

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 vectors:

```
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()

Arrays vs Vectors - Creating a copy

- Remember the problem with arrays, we couldn’t copy arrays by saying `array2 = array1`. Why?
- With vectors, you can copy over easily by using `vec2 = vec1`.

Important Vector functions

- [vector].size() – returns currents size of vector
- [vector].at(i) – returns element at i^{th} position
- [vector].push_back(element) – add element to the back of vector
- [vector].pop_back() – removes the last in vector
- [vector].front() – returns first element in vector
- [vector].back() – returns last element in vector
- [vector].empty() – returns true if no element in vector

Test yourselves

- ```
  vector<int> v(2);
  v.push_back(1);
  print(v); // assume this function exists!
  ```

  A. 1
  B. 1 1 1
  C. 0 0 1
  D. Compile time error
  ```

Test yourselves

- ```
  vector<int> v = {1};
  v.pop_back();
  v.pop_back();
  print(v); // assume this function exists!
  ```

  A. 1
  B. 0
  C. No output
  D. Compile time error
  ```

Vectors as Function Parameters

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:
  - we do not want to change the values in the vector
  - 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.

```
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.

```
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.

Arrays vs Vectors - Function Parameters

- Recollect how arrays worked with functions. Arrays when passed to a function, was **passed by reference.**
- On the other hand, vectors are **passed by value.** Which means, any modifications to a vector parameter inside a function, **is only visible inside the function.**
- *So, how do we make modifications to a vector inside a function?*

Vectors as return values from functions

- **Example:** Write a function that will take as input a vector and returns a vector with each element doubled
- Sample input: [1, 2, 3, 4] → Sample output: [2, 4, 6, 8]

```
vector<double> double(vector<double> values) {
    vector<double> new_vec;
    for (int i=0; i < values.size(); i++) {
        new_vec.push_back(values[i]*2);
   }
   return new_vec;
}
```

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

Common algorithms: finding matches

- Suppose we want to keep all values from an array that are greater than a certain value, say, 100.
- How could we do this with arrays?

Common algorithms: finding matches

- Suppose we want to keep all values from an array that are greater than a certain value, say, 100.
- How could we do this with arrays?
- Create a second array
- … same size as the original
- Loop over it, and copy all elements that meet the condition
- **Drawback:** new array is same size as old one (maybe only partially filled)
- Better idea: this is MUCH easier with vectors!
- Reflect: why?

Common algorithms: finding matches

```
// input: double scores[SIZE];
// an array of scores of size SIZE
vector<double> overachievers;
for (int i=0; i < SIZE; i++) {
  if (scores[i] > 100)
    {
    overachievers.push_back(scores[i]);
  }
}
```

Common algorithms: removing an element, unordered

- Suppose we want to remove an element from a vector values and the order of the vector values elements is not important. Then we could…
- Find the position of the element we want to remove (call it index i_rem)
- Overwrite that element with the last one from the vector
- Remove the last element from the vector
  - (makes the vector smaller by 1)
- **Handy member function:** [vec].back() -- returns the last element of a vector (doesn't pop it)

```
68 23 41 92 34 4 15 87 76
```

Common algorithms: removing an element, unordered

```
// first, need to loop over to find i_rem
values[i_rem] = values.back();
values.pop_back();
```

68 23 41 92 34 4 15 87 76

Common algorithms: removing an element, ordered

- Suppose we want to remove an element from a vector values and the order of the vector values elements **is important**. Then we could…
- Find the position of the element we want to remove (call it index i_rem)
- Overwrite that element with the next one from the vector (values[i_rem+1])
- Overwrite the next element with the one after that (values[i_rem+2])… and so on
- Remove the last element from the vector
  - (makes the vector smaller by 1)

68 23 41 92 34 4 15 87 76

Common algorithms: removing an element, ordered

```
// first, need to loop over to find i_rem
for (int i=i_rem; i<(values.size()-1); i++) {
   values[i] = values[i+1];
}
values.pop_back();
```

Common algorithms: inserting an element, unordered

- Suppose we want to insert an element into a vector values and the order of the vector values elements **is not** important. Then we could…
- Slap the new element (`noob`) onto the end of our vector!

```
values.push_back(noob);
```

Common algorithms: inserting an element, ordered

- Suppose we want to insert an element into a vector values **and** the order of the vector values elements **is important**. Then we could…
- … basically do our algorithm for removing an element, but in reverse.
- Suppose we have `i_ins` as the index we want the inserted element to be at

68 23 41 92 34 4 15 87 76

Common algorithms: inserting an element, ordered

- Suppose we want to insert an element into a vector values **and** the order of the vector values elements **is important**. Then we could…
- … basically do our algorithm for removing an element, but in reverse.
- Suppose we have `i_ins` as the index we want the inserted element to be at
- Add the last element to the **new last element slot**

  values.push_back(values.back()); // now vector is one size larger!
- Move the third-to-last element into the second-to-last slot
- Move the fourth-to-last element into the third-to-last slot … and so on.
- Place the new element at i_ins after all those after i_ins are shifted backward to make room

Common algorithms: inserting an element, ordered

```
// first, add a dummy element at the end
values.push_back(noob); // or any number
for (int i= values.size()-2; i>i_ins; i--) {
   values[i] = values[i-1];
}
values[i_ins] = noob;
```

68 23 41 92 34 4 15 87 76

Decoding what vectors are

Decoding Vectors

- Similar to arrays - except
  - Doesn’t need size to be known during declaration
  - Can add as many elements as we wish (`vector.push_back()`)

- Vectors are a **standard container class** in C++. What this means is that, C++ packages a few data and members inside a vector that you can use - like size, push_back, pop_back etc. It’s grouped together all of these in one umbrella termed as a *vector.*

Decoding Vectors

- How does it allow you to add as many elements as you wish??

2D Vectors

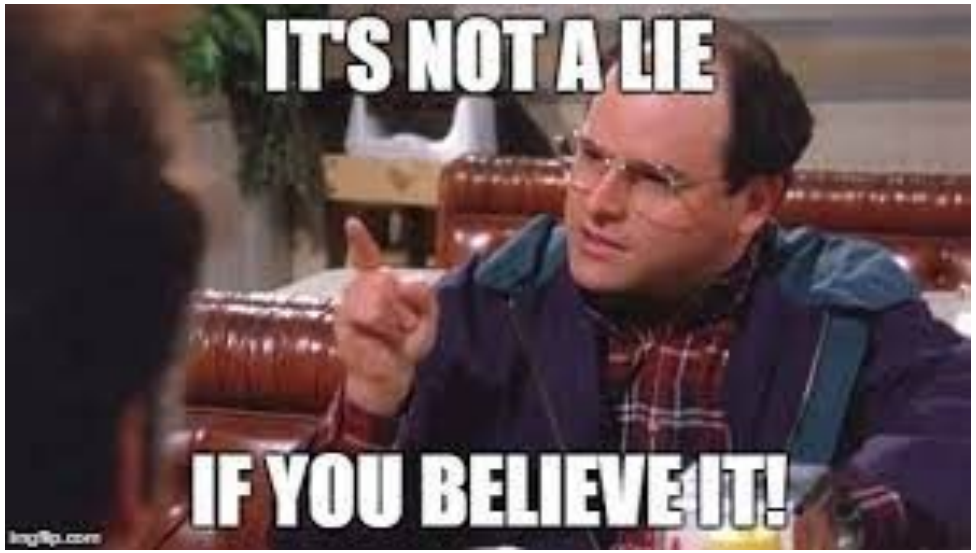
2D Vectors?

- There are no 2D vectors!
- *It's how you envision things!*

2D Vectors -> More like a vector of vectors

- There are no 2D vectors, but you can use a vector of vectors!

```
vector<vector<int>> counts;
```



A Vector of vectors

The advantage over 2D arrays:

- vector row and column sizes don't have to be fixed at compile time.

```
int countries = 7;
int medals = 3;
vector<vector<int>> counts;
for (int i = 0; i < countries; i++)
{
   vector<int> row(medals);
   counts.push_back(row);
}
```

vector of vectors

- You can access the vector `counts[i][j]` in the same way as 2D arrays.
- `counts[i]` denotes the i^{th} row, and
- `counts[i][j]` is the value in the j^{th} column of the i^{th} row.

vector of vectors: Determining row/columns

- To find the number of rows and columns:

```
vector<vector<int>> values = . . .;

int rows = values.size();
int columns = values[0].size();
```

vector of vectors: Different row sizes

- It is also possible to declare vectors of vectors in which the row size varies.

```
t[0][0]
t[1][0] t[1][1]
t[2][0] t[2][1] t[2][2]
t[3][0] t[3][1] t[3][2] t[3][3]
```

vector of vectors: Different row sizes

- It is also possible to declare vectors of vectors in which the row size varies.

```
t[0][0]
t[1][0] t[1][1]
t[2][0] t[2][1] t[2][2]
t[3][0] t[3][1] t[3][2] t[3][3]
```

- Add rows of the appropriate sizes:

```
vector<vector<int>> t;
for (int i = 0; i < 3; i++)
{
   vector<int> row(i + 1);
   t.push_back(row);
}
```

Arrays or vectors?

Short answer: Vectors are usually easier, and more flexible.

- Can grow/shrink as needed
- Don’t have to keep track of their size in a separate variable (vec.size())
- Pass-by-value

- But arrays are often **more efficient**. So beefier programs typically use arrays
- You still need to use arrays if you work with older programs or use C without the "++", such as in microcontroller applications.