Vectors

Due this week

- **Project 1**
  - Write solutions and tests in VSCode, then paste in Coderunner.
- Interview Grading Sign-up: Read the rules
  - make sure to arrive 5~ 10 min early and prepare! Have your laptop/tablet ready with your code.

```
int main(void)
{
    // testShiftChar();
    // testGetMethod2ShiftAmo
    // testGetMethod3ShiftAmo
    // testEncryptChar();
    // testDecryptChar();
    // testEncryptMessage();
    // testDecryptMessage();

    return 0;
}
```

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
- any function that takes the array as an input needs the **size** too
- wouldn't it be nice if there were something we could ***dynamically reshape***?!

Vectors

Dynamic array

- Not fixed in size when created
  - member function: [vector].size()
- Doesn’t require an auxiliary variable to track the size
- 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 `(vector<double> data)` 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;
```

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

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()

Manipulating elements in vectors

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);`

Manipulating elements in vectors

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)
  - LIFO 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);
}
```

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);
```

Arrays

- If you have two arrays: `int your_money[5]={ 0, 18, 7, 43, 4 };`
  `int my_money[5];`
- And further, we want what is stored in your_money to become my_money

- With arrays, we can not simply do this: `my_money = your_money;`
- Instead, we must loop:

```
for (int i=0; i < 5; i++) {
      my_money[i] = your_money[i];
}
```

Vectors

- With vectors, we can simply do this: `my_money = your_money;`

Other 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

2D Vectors

2D Vectors: a vector of vectors

- There are no 2D vectors, but if you want to store rows and columns, you can use a vector of vectors.

```
vector<vector<int>> counts;
//counts is a vector of rows. Each row is a vector<int>
```

- You need to initialize it, to make sure there are rows and columns for all the elements.

vector of vectors: advantages

The advantage over 2D arrays:

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

```
int COUNTRIES;
int MEDALS;
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();
              values[1].size();
              values[2].size();
```

Vectors and Functions

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.

How do arrays work with functions?

- 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?

```
void fillArray(int score[], int size)
{
    cout << "Enter 5 scores: \n";
    for(int i=0; i<5; i++)
    {
        cin >> score[0];
    }
}
```

```
void fillVector(vector<int> score)
{
    int input;
    cout << "Enter 5 scores: \n";
    for(int i=0; i<5; i++)
    {
        cin >> input;
        score.push_back(input);
    }
}
```

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.

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

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

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)

Common algorithms: removing an element, unordered

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

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)

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);
```

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.