Background

1.1 Arrays

An array is a *data structure* which can store primitive data types like floats, ints, strings, chars, and booleans.

Arrays have both a **type** and a **size**.

- **How to Declare Arrays**

```
data_type array_name[declared_size];
bool myBooleans[10];
string myStrings[15];
int myInts[7];
```

- **How to Initialize Arrays (Method 1)**

```
bool myBooleans[4] = {true, false, true, true};
```

If you do not declare the size inside the square brackets, the array size will be set to however many entries you provide on the right.

```
bool myBooleans[] = {true, false, true}; // size = 3
```

Note: the size specified in the brackets needs to match the number of elements you wrote in the curly brackets.

Example 1: When the specified size is larger than the actual number of elements, the elements provided in the curly brackets will be the first several elements in the array, while the additional elements will be filled with default values. If it's an integer/double array, the default values are zero, while if it's a string array, the default values are empty strings.

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

int main() {
    int intArray[5] = {1,2,3};
    for (int i = 0; i < 5; i ++) {
        cout << intArray[i] << " ";
    }
}
```

Output:

```
1 2 3 0 0
```

Example 2: When the specified size is smaller than the actual number of elements, there will be a compilation error.

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

int main() {
    int intArray[3] =
             {1,2,3,4,5};
}
```

Output (Error message):

```
error: excess elements in array
initializer
int intArray[3] = {1,2,3,4,5};
                          ^

1 error generated.
```

- **How to Initialize Arrays (Method 2)**

  You can also initialize elements one by one using a for or a while loop:

```
int myInts[10];
int i = 0;
while (i < 10) {
    myInts[i] = i;
    i++;
}
//{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

- **How to Access Elements in an Array**

  We have essentially already had practice with accessing elements in an array, as in C++, strings are array of characters.

You can access elements in arrays using the same syntax you used for strings:

```
string greetings[] = {"hello", "hi", "hey", "what's up?"};
cout << greetings[3] << endl;
```

"hello"	"hi"	"hey"	**"What's up?"**
0	1	2	**3**

Arrays can be iterated in the same way we iterated over strings last week. Below we are iterating through an array of strings:

```
string greetings[] = {"hello", "hi",
"hey", "what's up?"};
int size = 4;
int i = 0;
while (i < size){
      cout << greetings[i] << endl;
      i++;
}
```

Output:

```
hello
hi
hey
what's up?
```

1.2 Passing arrays to functions

- **Passing By Value**

  Up until now, when calling functions, we have always *passed by value.* When a parameter is passed in a function call, a new variable is declared and initialized to the value *passed* in the function call.

  Observe that the variable x in **main** and variable x in **addOne** are *separate* variables in memory. When **addOne** is called with x on line 9, it is the *value* of x (i.e. 5) that is *passed* to the function. This *value* is used to initialize a new variable x that exists only in **addOne**'s scope. Thus the value of the variable x in main's scope remains unchanged even after the function **addOne** has been called.

```
1      void addOne(int x){
2          x = x + 1;
3          cout << x << endl;
```

Output:

```
4       }
5
6       int main(){
7          int x = 5;
8          cout << x << endl;
9          addOne(x);
10         cout << x << endl;
11      }
```

```
5
6
5
```

- **Passing By Reference**

  Arrays, on the other hand, are *passed by reference* (to the original array's location in the computer's memory). So, when an array is passed as a parameter, the original array is used by the function.

  Observe that there is only *one* array X in memory for the following example. When the function **addOne** is called on line 9, a *reference* to the original array X is *passed* to **addOne**. Because the array X is *passed by reference*, any modifications done to X in **addOne** are done to the original array. These modifications persist and are visible even after the flow of control has exited the function and we return to main.

```
1       void addOne(int X[]){
2           X[0] = X[0] + 1;
3           cout << X[0] << endl;
4       }
5       int main(){
6          int X[4] = {1, 5, 3, 2};
7          cout << X[0] << endl;
8          addOne(X);
9          cout << X[0] << endl;
10      }
```

Output:

```
1
2
2
```

When we pass a one-dimensional array as an argument to a function we also provide its length. For two-dimensional arrays, in addition to providing the length (or number of rows), we will also assume that we know the length of each of the subarrays (or the number of columns). A function taking a two-dimensional array with 10 columns as an argument then might look something like this:

```
void twoDimensionalFunction(int matrix[][10], int rows){ … }
```

1.3 Multidimensional arrays

In C++ we can declare an array of arrays known as a multidimensional array. Multidimensional arrays store data in tabular form.

- **How to Declare Multidimensional arrays**

```
data_type array_name[row_size][column_size];
int myInts[7][5];
bool myBooleans[10][15][12];
string myStrings[15][10];
```

- **How to Initialize Multidimensional arrays (Method 1)**

```
int myInts[2][2] = {1, 2, 3, 4};
```

The 2D array in this case will be filled from left to right from top to bottom.

```
int myInts[2][2] = {{1, 2}, {3, 4}}
```

You can also initialize a 2D array by explicitly separating the rows as shown above.

- **How to Initialize Multidimensional arrays (Method 2)**

You can also initialize elements using nested loops:

```
int myInts[2][2];
for(int i=0; i < 2; i++){
      for(int j=0; j < 2; j++){
            myInts[i][j] = i + j;
      }
}
```

The above code will create the following 2D array `{{0, 1}, {1, 2}}`.

- **How to Access Elements in a Multidimensional array**

You can use `myInts[i][j]` to access the i^{th} row and j^{th} column of a 2D array

Multidimensional arrays can be iterated using nested loops as shown below:

```
int myInts[2][2] = {{0, 1}, {1, 2}};
int res = 0;
for(int i=0; i < 2; i++){
      for(int j=0; j < 2; j++){
            res = res + myInts[i][j];
      }
}

cout << "Result is " << res;
```

Output:
Result is 4

1.4 Introduction to Vectors

Let's start with something we already know about- Arrays.

To recap, an array is a contiguous series that holds a *fixed number of values* of the same datatype.

A vector is a template class that uses all of the syntax that we used with vanilla arrays, but adds in functionality that relieves us of the burden of keeping track of memory allocation for the arrays. It also introduces a bunch of other features that makes handling arrays much simpler.
How Vectors Work- Syntax and Usage

First things first. We need to include the appropriate header files to use the vector class.

```
#include <vector>
```

We can now move on to declaring a vector. This is general format of any vector declaration:

```
vector <datatype_here> name(size);
```

The size field is optional. Vectors are *dynamically-sized*, so the size that you give them during initialization isn't permanent- they can be resized as necessary.

Let's look at a few examples

```
//Initializes a vector to store int values of size 10.
```

```
vector <int> vec1(10);

//Initializes an empty vector that can store strings.
vector <string> vec2;
```

You can access elements of a vector in the same way you would access elements in an array, for example **array[4]**. Remember, indices begin from 0.

You can find a quick reference to the functions available in C++ vector class in a nice PDF form here, but following are the ones you will need in this recitation:

All these functions belong to the vector *class*, so you need to call them on an instance of a vector. (vec1.size(), for example).

- **`size()`** returns the size of a vector.
- **`at()`** takes an integer parameter for index and returns the value at that position

Adding elements to the vector is done primarily using two functions

- **`push_back()`** takes in one parameter (the element to be added) and appends it to the end of the vector.

```
vector <int> vector1; // initializes an empty vector
vector1.push_back(1); //Adds 1 to the end of the vector.
vector1.push_back(3); //Adds 3 to the end of the vector.
vector1.push_back(4); //Adds 4 to the end of the vector.
cout<< vector1.size(); //This will print the size of the
vector - in this case, 3.
//vector1 looks like this: [1, 3, 4]
```

- **`insert()`** can add an element at some position in the middle of the vector.

```
//vectorName.insert(vectorName.begin() + position, element)
vector1.insert(vector1.begin() + 1, 2);
cout << vector1.at(1) << endl; // 2 is at index=1
//vector1 looks like this: [1, 2, 3, 4]
```

Here, the **begin** function returns an iterator to the first element of the vector. You can think of it as an arrow pointer that points to the memory location just before the first

element. The **begin()** would thus refer to the first element and **begin()+k** would refer to the kth element in the vector, k starting at 0.

Elements can also be removed.

- **`pop_back()`** deletes the last element in the vector
- **`erase()`** can delete a single element at some position

```
//vector_name.erase(vector_name.begin() + position)
vector1.erase(vector1.begin() + 0);
cout << vector1.at(0) << endl; //2 is at index=0
//vector1 looks like this: [2, 3, 4]
```

The vector class has many more functions so look at: http://www.broculos.net/2007/11/c-stl-vector-class-cheatsheet.html if you are interested in learning more. (You may want to use vectors instead of arrays for project 3).