**Vectors**

**Vector Practice:** Create a c++ project using the code below. Then answer the questions.

#include <iostream>

#include <vector>

using namespace std;

void displayVector(vector<double> values){

for(int i = 0; i < values.size(); i++){

cout << values.at(i) << ", ";

}

cout << endl;

}

int main() {

vector<double>values;

for(double i = .5; i < 5; i+= .5)

values.push\_back(i);

displayVector(values);

}

1. Are there any additional include statements needed to use a vector? **For basic use, there are no other include statements needed.**
2. Notice the declaration statement for the vector, do you have to specify a size? **No, this is a large benefit of a vector. It can act as an array similarly but without a set/defined size.**
3. Add the following code to the end of the main(), how many elements are in the list? **9 elements are in the list.**

cout << "Num Items: " << values.size() << endl;

1. There are two ways to access each element in the list. The braces[] and the .at() method. Add the following statement to display the value at position 2

cout << "Position 2: " << values.at(2) << endl;

1. What happens if you use the .at() method to display a position 20? **An error is thrown as it is out of range.**
2. Modify the code to use the square brackets [ ] to display position 2 in the list. Is the value the same as when you used the .at() method. **Yes, they are both the same value.**
3. Modify the code to use the brackets to display position 20? Was there a difference in the error message you received? **I didn’t get an error when using position 20, instead it gave me the value was 1.39069e-309.**
4. From your observations, which access method brackets[] or .at() is better to use and why? **I feel that .at() would be a better choice for vectors as it will throw an error when you select out-of-range and not a false value, this will allow for more accurate results.**
5. Look at the following line of code, what value do you think is being inserted and where will it be in the list? **9.9 is being inserted in the 3rd position, or as the 4th item.**

values.insert(values.begin() + 3, 9.9);

1. Add the statement to your code and display the list again. Change the number to make it insert a value right after the 3.
2. Look at the following statement. What position in the list is being erased? **The 2nd position, 1 after the first.**

values.erase(values.begin() + 1);

1. What do you think is the purpose of the begin() function? **Get the start of the vector, the first item.**
2. From what you have observed, what are some advantages and/or disadvantages of using a vector over a primitive array? **You are able to slide values into the vector without adjusting all the other values manually, you are able to remove the same way, and you have undefined limits of the list.**

**Iterators**

We are going to demonstrate the efficiency of iterators by timing how long it takes to iterate the list using two different methods. Copy the following code into your editor.

#include <iostream>

#include <vector>

#include <chrono>

using namespace std;

using namespace std::chrono;

int main() {

const unsigned int LIST\_SIZE = 1000;

vector<int>myList;

for(int i = 0; i < LIST\_SIZE; i++){

myList.push\_back(rand() % 100);

}

auto start = high\_resolution\_clock::now();

*//First Check*

for(int i = 0; i <myList.size(); i++){

cout << myList[i] << " ";

}

auto end = high\_resolution\_clock::now();

auto duration = duration\_cast<microseconds>(end-start);

cout << "\nDuration: " << duration.count() << endl;

start = high\_resolution\_clock::now();

*//Second Check*

for(vector<int>:: iterator i = myList.begin(); i != myList.end();++i){

cout << \*i << " ";

}

end = high\_resolution\_clock::now();

duration = duration\_cast<microseconds>(end-start);

cout << "\nDuration: " << duration.count() << endl;

}

1. What is the size of the list we are testing? **1000 items.**
2. What values are being added to the list? **Random values between 0 and 99.**
3. Look under the comment “First Check”. You will see a for loop iterating the list. Is this using direct access to the list, or an iterator to access each element of the array? **It is using direct access to the vector with square brackets.**
4. Look at the code under “Second Check”. You will see a for loop iterating the list. Is this direct access or an iterator? **Second check is using the iterator, from begin() to end().**
5. Run the program. The list will be displayed, then a statement with the duration it took to display that list. This will be done twice for the two types of iteration. Which was faster? **The Iterator is faster by 4000 microseconds, or 4 milliseconds.**
6. Run the program several times and observe the results. Try making the list size larger and/or smaller to see if it affects the outcome.
7. Which way to iterate the array is more efficient? Why do you think it is more efficient? **In general, the iterator is faster, although sometimes, especially at exceptionally large numbers, the direct access was faster. I believe the iterator is faster because it stores the value in memory as it encounters it, whereas the direct access has 2 separate values, one to know which value and the list itself which it has to access separately and find the given value.**

**Stack and Queue**

We are going to practice using stack and queues. Later in the class, we will actually create our own stack and queue class. For now, we will use that in the standard template directory. Use the following code to practice stacks and queues.

#include <iostream>

#include <stack>

#include <queue>

using namespace std;

int main() {

//Stack Code

stack<int>myStack;

queue<int>myQueue;

for(int i = 10; i < 100; i+= 10 ){

myStack.push(i);

myQueue.push(i);

}

}

1. What do you think will be the value on the top of the stack? **I think that 90 will be at the top of the stack.**
2. What do you think will be the value on the front of the queue? **I think that 10 will be in the front of the queue.**
3. Key in the following to check your answers.

cout << "Stack top: " << myStack.top() << endl;

cout << "Queue Front: " << myQueue.front() << endl;

1. Since both data structures contain the same values, why did that value differ? **The top of a stack is essentially the bottom, as new items are placed on the top, whereas the front of the queue is the beginning where new items are placed at the back, or end.**
2. How many values are currently in the stack? Use the following code to check your answer. **There are 9 items in the stack/queue**

cout << "Stack size: " << myStack.size() << endl;

cout << "Queue size: " << myQueue.size() << endl;

1. To remove an element from the data structure, you use the pop command. Add the command to remove an element from the stack and the queue.
2. Now what do you think the top value will be? Display the top of the stack to check your answer. **The top of the stack should be 80 now, as the next in line would become the top.**
3. The .empty() command will return true/false if the stack or queue is empty. Write a while loop that will remove all the items from the queue. Then display the size to make sure it worked.
   1. While (!myQueue.empty()) {
      1. myQueue.pop();
   2. }
   4. cout << “Queue size: “ << myQueue.size() << endl;
4. Now let’s try it with the Stack, but as you remove items from the stack, add them to the queue.
5. What do you think the front value of the Queue will be? Write a cout statement to check your answer. **I think that the front of the Queue will be 10, as that would be the last item in the stack that gets pushed to the Queue.**
6. The .back() command will allow you to view the last item in the queue. Write what you think the value will be, then write a cout statement to check your answer. **The last value in the Queue should also be 10 as it was the first item you pushed to the queue.**

**#include <iostream>**

**#include <stack>**

**#include <queue>**

**using namespace std;**

**int main() {**

**//Stack Code**

**stack<int>myStack;**

**queue<int>myQueue;**

**for(int i = 10; i < 100; i+= 10 ){**

**myStack.push(i);**

**myQueue.push(i);**

**}**

**cout << "Stack top: " << myStack.top() << endl;**

**cout << "Queue Front: " << myQueue.front() << endl;**

**cout << "Stack size: " << myStack.size() << endl;**

**cout << "Queue size: " << myQueue.size() << endl;**

**myStack.pop();**

**cout << "Stack top: " << myStack.top() << endl;**

**while (!myStack.empty()) {**

**myQueue.push(myStack.top());**

**myStack.pop();**

**}**

**cout << "Stack size: " << myStack.size() << endl;**

**cout << "Queue Front: " << myQueue.front() << endl;**

**cout << "Queue Back: " << myQueue.back() << endl;**

**}**