CSI 1430
Week 6 Resource

Colin Burdine

2/22/21

Major Topics:

1. Arrays
2. Vectors

Keywords: *arrays, vectors, STL, standard template vectors*

Reminder: Weather permitting, tutoring sessions for this course will be held online every Monday from 7:00pm to 8:00pm CT. For more information on how to sign up for these sessions, go to: https://www.baylor.edu/tutoring.

1 Notes on Major Topics:

1.1 Arrays

Last week, we introduced the concept of loops, which are useful in executing some sequence of code repeatedly until some condition is met. However, we may also encounter problems where we not only want to perform the same task repeatedly, but we want to repeat the same take on an unknown number of values of the same type. For example, suppose we want to sum a list of integers given by the user. At the time the program is compiled, we may not know *how many* numbers the user will enter; however, we do know, the iterative process we would use to sum these numbers if they are entered all at once.

This creates a problem- how do we store an unknown number of instances of the same type?. The solution to this problem is to use *arrays* and *vectors*. Arrays in C++ use the syntax below:

```
// this creates an array to store 10 integers:
int myArray[10];

// we can access the (n+1)th integer in the array using:
int n = 4;
int fifthElement = myArray[n];
```

Note that in the example above, we access the 5th item in the array by using `myArray[4]` instead of `myArray[5]`. This is because the first element of any array is found at index 0, and can be accessed using `myArray[0]`.

It is important to keep in mind that arrays in C++ are all fixed in size. That is, in our program we must specify the size of an array explicitly and cannot change it at any time. This has some benefits and drawbacks. One of the benefits of having a statically sized (i.e. fixed size) array is that it is very memory efficient. Since the elements of arrays are stored sequentially in the computer's logical memory, the array has a minimal memory footprint. However, on today's systems memory is typically not an issue. One of the drawbacks of arrays, as we mentioned above, is that arrays cannot be made larger than their fixed size. This can create several problems if you are not careful in how you code. To illustrate this, consider the following example program:

```

#include <iostream>
using namespace std;

int main(){

    const int ARRAY_SIZE = 5;
    int myArray[ARRAY_SIZE] = { 10, 20, 30, 40, 50 };
    int numToPrint;

    // prompt user for number of elements of myArray to print:
    cout << "Enter how many numbers to print: ";
    cin >> numToPrint;

    // print the indicated number of elements of myArray:
    for(int i = 0; i < numToPrint; ++i){
        cout << myArray[i] << ' ';
    }
    cout << endl;

    return 0;
}
```

Notice that if we run this program and enter the number 4, we obtain the expected

output:

```
10 20 30 40
```

So what's the problem? Well, suppose that the user enters a value greater than 5, say 10. How will the program behave?
One possible output that could be printed is:

```
10 20 30 40 50 32766 -699793152 -2115789050 0 0
```

On some systems, you may may also get the following ominous message:

```
Segmentation fault (core dumped)
```

So what is going on? By default, C++ provides no *address protection*, which means that if we attempt to access memory that is outside the range of an array, C++ will not always try to stop us. In fact, C++ will only try and stop us from accessing memory if we are far outside the boundary of the memory set aside for our program. This is referred to as a segmentation fault, and is what causes the second error message above. To summarize our findings, we observe that we must be careful when using fixed size arrays in C++.

1.2 Arrays

One way of dealing with the problem of fixed size arrays in C++ is to use vectors instead. vectors are much like arrays, except they grow in size automatically as we add more elements. In C++ vectors are included as part of the STL (Standard Template Library) and can be used in your programs by adding the "`#include <vector>`" preprocessor directive. Some of the basic syntax for vectors is given below:

```
1  // be sure to use #include <vector> at the top of your source file!
2
3  // creates an empty vector to store int types:
4  vector<int> myVector();
5
6  // adds an integer to the end of the array:
7  int someInt = 2;
8  myVector.push_back(someInt);
9
10 // gets the size of a vector:
11 int arrSize;
12 arrSize = myVector.size()
```

For more information on how to use vectors (and more STL types) see the online documentation at http://www.cplusplus.com/reference/vector/vector/

2 Programming Example Problem

To illustrate how we can use arrays and vectors to solve problems, consider the following example programming problem:

Suppose we are given two lists of integers by the user, List A and List B. These lists are of unknown length, however neither List A nor List B contain a number more than once. The *intersection* of A and B is the list of numbers contained in both A and B. Write a program that prompts the user for the size of List A followed by the integers contained in List A. Then do the same for list B. Once you program has read all of the numbers in List A and List B, it should print out all numbers in the intersection of List A and List B. The order of the numbers in the intersection may vary. An example is given below:

```
How many numbers are in List A? 4
Enter the numbers in List A:
1 2 3 4
How many numbers are in List B? 5
Enter the numbers in List B:
-1 3 -2 0 4
The numbers in the intersection of List A and List B are:
3 4
```

My solution to this problem is on the next page. You can look at my answer if you are stuck, but I would strongly encourage you to use the empty space below to sketch out a design first, and then proceed to write your solution in C++. This exercise is intended to check your understanding of the concepts from this week and last week.

```cpp
/* Your proper header comments should go here :) */
#include <iostream>
#include <vector>
using namespace std;

int main(){

    // allocate vectors to store our lists:
    int sizeA, sizeB, num;
    vector<int> listA, listB, intersection;

    // prompt for the size of List A:
    cout << "How many numbers are in List A? ";
    cin >> sizeA;

    // read the numbers in List A:
    cout << "Enter the numbers in List A:" << endl;
    for(int i = 0; i < sizeA; ++i){
        cin >> num;
        listA.push_back(num);
    }

    // prompt for the numbers in List B:
    cout << "How many numbers are in List B? ";
    cin >> sizeB;

    // read the numbers in List B:
    cout << "Enter the numbers in List B:" << endl;
    for(int i = 0; i < sizeB; ++i){
        cin >> num;
        listB.push_back(num);
    }

    // for every number in List A, determine if it is in List B also:
    cout << "The numbers in the intersection of List A and List B are:" <<
         endl;
    for(int i = 0; i < sizeA; ++i){

        bool numInB = false;
        for(int j = 0; j < sizeB && !numInB; ++j){
            if(listA[i] == listB[j]){
                cout << listA[i] << ' ';
                numInB = true;
            }
        }
    }
    cout << endl;
    return 0;
}
```