Lecture 08: Working with Vectors

In lectures 5 and 6, it was established that C++ allows you to create and manage a dynamic list by declaring the list as a pointer and manipulating it as an array. Another effective way to create and manage dynamic lists is to do so with vectors.

This brief lecture introduces you to vector manipulation. The lecture proceeds under the following captions:

- Introduction
- The vector Class and its Member Functions
- Accessing and Manipulating Vector Elements
- Vectors of Objects
- Summary and Concluding Remarks

8.1 Introduction

A vector is a dynamic list that may grow or shrink in length (also called size) during the execution of a program. When working with vectors, be sure to include the **vector** header file at the beginning of your program file. The syntax for defining a C++ vector is shown in figure 8-1 below.

Figure 8-1: Syntax for Defining a Vector

VectorDefinition ::=
vector "<"<BaseType>">" <VectorName>;

Note:
1. The angular bracket that encloses the base type is required as prat of the syntax, hence the use of quotation marks.
2. The base type specified may be any valid primitive or advanced data type.

Example 8-1: The following code snippet shows the declaration of a vector to store integers.

```
vector <int> v; // vector for integers
vector <double> v2;  // vector for real numbers
// . . .
```

Once created, the vector can be accessed and manipulated, as clarified in the upcoming sections. Each item in the vector is called an element; vector elements start at subscript zero. Moreover, the act of creating a vector enables the C++ compiler to create a class with the same name as the vector. You can access and make use of the various member functions (i.e. methods) of the vector class.

8.2 The Vector Class and its Member Functions

Figure 8-2 shows a summary of selected member functions of the **vector** class; the list is not exhaustive but rather a feature of some of the more useful member functions.

Figure 8-2: Selected Member Functions of the Vector Class

Member Function	Description
begin()	Returns the iterator to the beginning of the vector.
end()	Returns the iterator to the beginning of the vector.
size()	Returns an unsigned integer with the size (i.e. length) of the vector.
max_size()	Returns an unsigned integer with the maximum possible size (i.e. length) of the vector.
resize(. . .)	Resizes the vector to the number items indicated by the first argument; the second argument if specified, contains the default value for each element after the original size has been reached; if the second argument is not specified, additional elements after the original size is reached, are assigned the default value for the base type; returns void.
capacity()	Returns the storage space (in terms of number of elements) currently allocated for the vector; typically varies between **size**() and **max_size**().
empty()	Returns **true** or **false**, indicating whether the vector is empty or not
reserve(. . .)	Requests to ensure a minimum capacity for the vector; the argument specifies the minimum capacity; returns void.

Figure 8-2: Selected Member Functions of the Vector Class (continued)

Member Function	Description
shrink_to_fit()	Requests that the vector capacity be shrunk to be equal to its current size.
operator [ ]	Overloaded operator **[ ]** to access the particular element at the specified relative location in the vector; the argument is identical to an array subscript.
at(. . .)	Returns the particular element at the specified relative location in the vector; the argument is identical to an array subscript.
front()	Returns the element at the front of the vector, i.e., the first element.
back()	Returns the element at the back of the vector, i.e., the last element.
push_back(. . .)	Inserts an element at the back of the vector; the argument must be of the base type of the vector.
insert(. . .)	Inserts an element at the iterator position and returns the iterator position; the first argument states the iterator position; the second argument is the element to be inserted. Example: If **v** is a vector of integers, then two valid insertions are as follows: iterator it = v.insert(v.begin(), 100); it = v.insert(v.begin() +1, 300); Other overloaded forms of this member function exists.
pop_back()	Removes the last element from the vector; returns void.
erase(. . .)	Removes the element(s) indicated by the argument, which is specified in terms of iterator positions, e.g. v.**begin**(), v.**begin**() + 3, etc., where **v** represents a vector; returns void.
swap(. . .)	Exchanges the contents of the current vector with those of the vector supplied as the argument; returns void. Example: If vi and v2 are two vectors, swap them vie the statement v1.swap(v2);
clear()	Removes all elements from the vector; returns void.

You are encouraged to practice writing code that makes use of these member functions. In the upcoming sections, you will see a few examples doing that.

8.3 Accessing and Manipulating Vector Elements

Vector elements may be accessed via the square bracket in the same manner as accessing of array elements; this is due to the fact that the subscripting operator ([ ]) has been overloaded in the **vector** class.

There is one exception to this principle: If you have a vector **v**, you cannot initialize the i^{th} element by directly making an assignment to **v**[i]. However, if the element **v**[i] already exists, it can be accessed directly and changed via an assignment statement.

You can insert elements into a vector via either the **insert**(. . .) or **push_back**(. . .) member function. Alternately, if the vector is trivial (for instance vector of a primitive data type with less than 6 items), then you may do a vector initialization similar to initialization of a trivial array; however, this approach is not recommended for more sophisticated vectors.

Once a vector element has been initialized, it can be updated via direct assignment using the appropriate vector subscript. For instance, if **v**[i] has a value, it can be modified as follows: v[i] = newValue;
Of course, **newValue** must be of the same base type as vector **v**.

Example 8-2: The following code snippet (figure 8-3) illustrates how you may create a vector of integers and insert values into it.

Figure 8-3: Creating and Loading a Vector of Integers

```
vector <int> myIntegerList1;
// . . .
// Prompt for vector elements
int vLength, x, intVal;

// myIntegerList1 = {10, 15, 5, 4, 26}; // optional initialization
cout << "How many items do you want to insert? ";
cin >> vLength; getchar();
for (x = 1; x <= vLength; x++)
{
  cout << "Enter value for integer " << x – 1 << "  ";
  cin >> intVal; getchar();
  myIntegerList1.push_back(intVal);
}
```

As the vector grows or shrinks, you can determine its size by calling the **size**() member function. You may then use that information to control a loop or convey information to the user.

Example 8-3: In the following code snippet (figure 8-4), the vector of figure 8-3 is assumed and used to construct a second vector, where each element is a multiple of the corresponding element in **myIntegerList1**.

Figure 8-4: Making use of the Vector Size

```
// . . .
// Create a second vector with items being a specified factor bigger than those in myIntegerList1
vector <int> myIntegerList2; int scaleFactor;
cout << "Enter the desired scale factor for the second vector: ";
cin >> scaleFactor; getchar();

for (x = 1; x <= myIntegerList1.size(); x++)
{ myIntegerList2.push_back(myIntegerList1[x – 1] * scaleFactor); }
```

To access an element in the vector, simply use the square brackets as you would for an array. Alternately, you may use the **at**(. . .) member function.

Example 8-4: The following code snippet (figure 8-5) displays the contents of the vectors in the two previous examples. In each case, the commented statement is simply an alternate way of displaying the information.

Figure 8-5: Displaying Vector Elements

```
// . . .
// Display elements of myIntegerList1 and myIntegerList2
int y;
for (y = 1; y <= myIntegerList1.size(); y++)
{ cout << myIntegerList1[y – 1] << endl } // cout << myIntegerList1.at(y – 1);
// . . .
for (y = 1; y <= myIntegerList2.size(); y++)
{ cout << myIntegerList2[y – 1] << endl }  // cout << myIntegerList1.at(y – 1);
```

8.4 Vectors of Objects

By this point, you may be wondering, so can I create and manipulate vectors of objects? You certainly can! The base type for a vector may be any valid primitive or advanced data type; so you can have a vector of anything, including a vector of vectors.

Recall that in section 6.9 of lecture 6, we discussed the concept of managing a dynamic list of objects, by declaring the list as a pointer and manipulating it as an array. A more superior way of handling this scenario is simply to create and manage a vector of objects.

Example 8-5: In example 6-12, the code for constructing and managing a dynamic list student objects was provided. Following in figure 8-6 is the modified code for addressing the problem via a vector. In the figure, important lines of code are highlighted by an arrow pointing left and preceding a comment.

Figure 8-6: Program Listing for Manipulating a Dynamic List of StudentC Objects via Vector

```
// ********************************************************************************************************************************************
// Program: StudentList03: Builds a list of StudentC objects
// Class: StudentC
// Author:  E. Foster
// ********************************************************************************************************************************************
#include <cstdlib>      #include <iostream>    #include<ctype.h>      #include<string.h>
using namespace std;

typedef char* cString;
class StudentC
{
 float studGPA;  // known only to class members
 protected:
 int studID_Number;
 char studLastName[16];
 char studFirstName[16];
 int studDateOfBirth;
 char studMajor[31];

// member function prototypes
 public:
 StudentC(); // Constructor
 void modify (StudentC aStudent);
 void inputData(int x);
 void printMe();
 void determineGPA (float inScore [ ]);
 int getNumber();
 String getName();
}; // End of Student class declaration

// The Member Functions of StudentC are as specified in figure 6-13a. They will not be duplicated here; see figure 6-13a for such details
```

```
// ********************************************************************************************************************************************
// Program: StudentList03
// Class/File: StudentMain
// Author:  E. Foster
// Purpose: Builds a dynamic list of StudentC objects -- vector implementation
// ********************************************************************************************************************************************
// Note: This is an improvement of the program EFStudList02B of EFConProject7B of the resource library
// ********************************************************************************************************************************************
#include <cstdlib>
#include <iostream>
#include<ctype.h>
#include<string.h>
#include "StudentC.h"
#include <vector>
using namespace std;

// Continued on the next page
```

Figure 8-6: Program Listing for Manipulating a Dynamic List of StudentC Objects via Vector (continued)

```
// Global Variables
typedef char* cString;
typedef vector <StudentC> StudentList;  // ← Note the change here from a pointer to a vector
const cString HEADING = "Process of Student Objects";

//Other Function Prototypes
StudentList inputStudents(); // ← Note: Since we are passing a vector to the function, size determination will be easy
void printList(StudentList sList); // ← Note again: Since we are passing a vector to the function, size determination will be easy

// Main function
int main(int argc, char *argv[])
{
 // Declarations
bool exitTime = false;   char exitKey;
 StudentList studList; // int studLim;

 while (!exitTime) // While user wishes to continue
 {
  // Obtain list of students
  studList = inputStudents(); // ← Simpler function call

  // Print the list, then check whether user wishes to continue
  printList(studList); // ← Simpler function call
  cout << "\n Press any key to continue or X to exit ";    exitKey = getchar();
  if (toupper(exitKey) == 'X') exitTime = true;

 } // End of While-user -wishes-to-continue

 system("PAUSE");
 return EXIT_SUCCESS;
} // End of main function

// InputStudents Function
StudentList inputStudents()
{
  int lSize;  StudentList rList; StudentC thisStud;
  cout << HEADING;   cout << "\n\nPlease enter the number of students required: ";
  cin >> lSize; getchar();

  // Prompt for information on each student
  for (int x = 1; x <=lSize; x++)
  {
          thisStud = StudentC(); thisStud.inputData(x); // Instantiate the item and then obtain values for it
          rList.push_back(thisStud); // ← Insert into the vector
  }
  return rList;
} // End of inputStudents Function

// Function printList
void printList (StudentList sList)
{
          int sLimit = sList.size(); // ← Determine and store the vector size
          int x;  cout << "\nYou have entered information for " << sLimit << " " << "Student(s)";
          cout << " as follows: \n";
          for (x = 1; x <= sLimit; x++) sList [x-1].printMe( );
} // End of printList function
```

8.5 Summary and Concluding Remarks

Let us summarize what has been covered in this lecture:

- The act of creating a vector enables the C++ compiler to create a class with the same name as the vector. You can access and make use of the various member functions (i.e. methods) of the vector class.
- Vector elements may be accessed via the square bracket in the same manner as accessing of array elements. There is one exception to this principle: You are not allowed to initialize vector elements via direct assignment to the elements; they must be initialized via the **push_back**(. . .) or the **insert**(. . .) member function.
- To access an element in the vector, simply use the square brackets as you would for an array. Alternately, you may use the **at**(. . .) member function.
- As the vector grows or shrinks, you can determine its size by calling the **size**() member function. You may then use that information to control a loop or convey information to the user.
- The base type for a vector may be any valid primitive or advanced data type; so you can have a vector of anything, including a vector of objects or a vector of vectors.

Working with vectors is really fun! You are encouraged to practice writing code that create and manipulate dynamic lists via this methodology.

The next lecture discusses inheritance.

8.6 Recommended Readings

[Friedman & Koffman 2011] Friedman, Frank L. & Elliot B. Koffman. 2011. *Problem Solving, Abstraction, and Design using C++*, 6th Edition. Boston: Addison-Wesley. See chapter 11.

[Gaddis, Walters & Muganda 2014] Gaddis, Tony, Judy Walters, & Godfrey Muganda. 2014. *Starting out with C++ Early Objects*, 8th Edition. Boston: Pearson. See chapter 8.

[Savitch 2013] Savitch, Walter. 2013. *Absolute C++*, 5th Edition. Boston: Pearson. See chapter 7.

[Savitch 2015] Savitch, Walter. 2015. *Problem Solving with C++*, 9th Edition. Boston: Pearson. See chapter 8.

[Yang 2001] Yang, Daoqi. 2001. *C++ and Object-Oriented Numeric Computing for Scientists and Engineers*. New York, NY: Springer. See chapter 6.