OOP 3200 – Object Oriented Programming II

Week 5 – Collections

Week 3 Overview

- Collections
- In-class Exercise 4
- Lab 4

Course Outline

Week	Date	Topic	Evaluation	Weight
1	Sep 09, 2020	- Course Orientation - Object-Oriented Programming overview - Partnering for labs		
2	Sep 16, 2020	REVIEW OF CLASSES & OBJECTS in C++ - Encapsulation - Object Attributes and Behaviours - Classes: The Blueprint for Objects - Relationship Between Class and Objects - Static Class Members - Friend Functions	In-Class Exercises 1: (2%) C++ Assignment 1: (6%)	8
3	Sep 23, 2020	CLASS OPERATORS AND DATA TYPE CONVERSIONS in C++: - Creating Class Operators - How Methods Are Shared - Data Type Conversions	In-Class Exercises 2: (2%) C++ Assignment 2: (6%) C++ Quiz 1: (5%)	13
4	Sep 30, 2020	INHERITANCE AND POLYMORPHISM in C++: - Class Inheritance - Polymorphism - Virtual functions - Interfaces / Abstract Classes	In-Class Exercises 3: (2%) C++ Assignment 3: (6%)	8
5	Oct 07, 2020	COLLECTIONS in C++: - Dynamic Object Creation and Deletion - Pointers As Class Members / Destructors - Copy Constructors / Copy Assignment Operators	In-Class Exercises 4: (2%) C++ Assignment 4: (6%)	8
6	Oct 14, 2020	GENERICS in C++ - Method templates - Class Templates	In-Class Exercises 5: (2%)	2
7	Oct 21, 2020	THE C++ STANDARD TEMPLATE LIBRARY (STL): - Vectors and Linked Lists - Stacks and Queues - Maps and Sets	In-Class Exercises 6: (2%) C++ Assignment 5: (6%) C++ Quiz 2: (5%)	13
READING WEEK				

Objectives

- This week, you will learn about:
  - Introduction the STL `vector`
  - Arrays of Objects
  - Array of Structures

Introduction to the STL vector

Concept

- The Standard Template Library includes a data type called a `vector`. It is similar to a **one-dimensional array** but has a number of advantages compared to a standard array.

- **The Standard Template Library (STL)** is a collection of programmer-defined data types and algorithms that are available for you to use in your C++ programs.

- These data types and algorithms are not part of the C++ language but were created in addition to the built-in data types.

- If you plan to continue your studies in the field of computer science, you should become familiar with the STL.

- This section introduces one of the STL data types, the `vector`.

STL Containers

- The data types that are defined in the STL are commonly called **containers**.

- They are called containers because they **store and organize data**.

- There are two types of containers in the STL: **sequence containers** and **associative containers**.

  - A **sequence container** organizes data in a sequential fashion, similar to an array.

  - **Associative containers** organize data with keys, which allow rapid, random access to elements stored in the container.

The `vector` data type

- The vector data type is a sequence container that is like a one-dimensional array in the following ways:
  - A `vector` holds a **sequence of values**, or **elements**.
  - A `vector` stores its elements in **contiguous memory locations**.
  - You can use the array subscript operator [ ] to access individual elements in the vector.
- However, a vector offers several **advantages** over arrays. Here are just a few:
  - You do not have to declare the **number of elements** that the vector will have.
  - If you add a value to a vector that is already full, the vector will **automatically increase its size** to accommodate the new value.
  - Vectors can report the number of elements they contain.

Defining and Initializing a `vector`

- To use **vectors** in your program, you must include the `vector` header file with the following statement:

```
#include <vector>
```

- To **create** a vector object you use a statement whose syntax is somewhat different from the syntax used in defining a regular variable or array. Here is an example:

```
vector<int> numbers;
```

- This statement defines numbers as a vector of `int`s. Notice that the data type is enclosed in **angled brackets**, immediately after the word `vector`.

Defining and Initializing a `vector` (continued)

- Because a `vector` expands in size as you add values to it, there is no need to declare a size. However, you can declare a **starting size**, if you prefer. Here is an example:

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

- This statement defines numbers as a vector of `10 ints`, but this is only a **starting size**. Its size will expand if you add more than 10 values to it.

- When you specify a **starting size** for a `vector`, you may also specify an **initialization value**. The initialization value is copied to each element. Here is an example:

```
vector<int> numbers(10, 2);
```

- This defines numbers as a vector of `10 ints` with each element initialized to 2.

Defining and Initializing a `vector` (continued)

- You may also initialize a vector with the values in **another `vector`**. For example, if `set1` is a `vector` of `ints` that already has values in it, the following statement will create a new `vector`, named `set2`, which is an exact copy of `set1`.

```
vector<int> set2(set1);
```

- After this statement executes, the vector `set2` will have the **same number of elements** and hold the **same set of values** as `set1`.

- If you are using C++ 11, you can also initialize a vector with a list of values, as shown in this example:

```
vector<int> numbers { 10, 20, 30, 40 };
```

- This statement defines a `vector` of ints named numbers. The `vector` will have four elements, initialized with the values `10`, `20`, `30`, and `40`. Notice that the initialization list is enclosed in a set of **braces**, but you do not use an = operator before the list.

DURHAM COLLEGE
SUCCESS MATTERS

Example `vector` Definitions

Definition Format	Description
`vector<string> names;`	This defines `names` as an empty vector of `string` objects.
`vector<int> scores(15);`	This defines `scores` as a vector of 15 `ints`.
`vector<char> letters(25, 'A');`	This defines `letters` as a vector of 25 characters. Each element is initialized with 'A'.
`vector<double> values2(values1);`	This defines `values2` as a vector of `doubles`. All the elements of `values1`, which is also a vector of `doubles`, are copied to `values2`.
`vector<int> length{12, 10, 6};`	In C++ 11 this defines `length` as a vector of 3 `ints`, holding the values 12, 10, and 6.

Storing and Retrieving Values in a `vector`

- To store a value in a `vector` element that already exists or to access the data stored in a vector element, you may use the **array subscript operator** [ ].

```
// This program stores employee hours worked
// and hourly pay rates in two parallel vectors.
#include <iostream>
#include <iomanip>
#include <vector>                       // Needed to use vectors
using namespace std;

int main()
{  const int NUM_EMPS = 5;              // Number of employees
   vector <int> hours(NUM_EMPS);        // Define a vector of integers
   vector <double> payRate(NUM_EMPS);   // Define a vector of doubles
   double grossPay;                     // An employee's gross pay

  // Get employee work data
  cout << "Enter the hours worked and hourly pay rates of "
       << NUM_EMPS << " employees. \n";
```

Storing and Retrieving Values in a **`vector`** (continued)

```
for (int index = 0; index < NUM_EMPS; index++)
  {
     cout << "\nHours worked by employee #" << (index + 1) << ": ";
     cin  >> hours[index];
     cout << "Hourly pay rate for this employee: $";
     cin  >> payRate[index];
  }
  // Display each employee's gross pay
  cout << "\nHere is the gross pay for each employee:\n";
  cout << fixed << showpoint << setprecision(2);

  for (int index = 0; index < NUM_EMPS; index++)
  {
     grossPay = hours[index] * payRate[index];
     cout << "Employee #" << (index + 1);
     cout << ": $" << setw(7) << grossPay << endl;
  }
  return 0;
}
```

Using the Range-Based **for** Loop with a **vector**

❖ Using the Range-Based for Loop with a vector

```
// This program uses two range-based for loops with a vector.
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    // Define a vector with a starting size of 5 elements
    vector<int> numbers(5);

    // Get values for the vector elements
    // Make the range variable a reference variable so it can be
    // used to change the contents of the element it references.
    for (int &val : numbers)
    {
       cout << "Enter an integer value: ";
       cin  >> val;
    }

    // Display the vector elements
    cout << "\nHere are the values you entered: \n";

    for (int val : numbers)
       cout << val << "  ";

    cout << endl;
    return 0;
}
```

Using the `push_back` Member Function

- You cannot, however, use the [ ] operator to access a `vector` element that does not yet exist.
- To store a value in a vector that does not have a starting size, or that is already full, you should use the `push_back` member function.
- This function accepts a value as an argument and stores it in a new element placed at the end of the vector. (It "pushes" the value at the "back" of the vector.)
- Here is an example that adds an element to a vector of ints named numbers.

```
numbers.push_back(25);
```

- This statement creates a **new element** holding 25 and places it at the end of numbers. If numbers previously had no elements, the new element becomes its **single element**.

Determining the Size of a Vector

- Unlike arrays, `vectors` can **report the number of elements** they contain. This is accomplished with the `size` member function. Here is an example of a statement that uses the size member function:

```
numValues = set.size();
```

- In this statement, assume that `numValues` is an int and set is a vector. After the statement executes, `numValues` will contain the number of elements in set.

- The size member function is especially useful for writing functions that accept vectors as arguments. For example, look at the following code for the `showValues` function:

```
void showValues(vector<int> vect)
{
   for (int count = 0; count < vect.size(); count++)
      cout << vect[count] << endl;
}
```

Removing Elements from a `vector`

- To **remove** the last element from a vector you can use the `pop_back` member function. The following statement **removes the last element** from a `vector` named collection:

```
collection.pop_back();
```

Clearing a Vector

- To completely clear the contents of a vector, use the clear member function, as shown in the following example:

```
numbers.clear();
```

- After this statement executes, numbers will be cleared of all its elements.

Detecting an Empty `vector`

- To determine if a `vector` is empty, use the `empty` member function.

- The function returns `true` if the `vector` is empty, and `false` if the vector has elements stored in it. Assuming `numberVector` is a `vector`, here is an example of its use:

```
if (numberVector.empty())
    cout << "No values in numberVector.\n";
```

Summary of `vector` Member Functions

Member Function	Description
`at(position)`	Returns the value of the element located at *position* in the vector. *Example:* `x = vect.at(5);` // Assigns the value of `vect[5]` to `x`.
`clear()`	Clears a vector of all its elements. *Example*: `vect.clear();` // Removes all the elements from `vect`.

`empty()`	Returns `true` if the vector is empty. Otherwise, it returns `false`.
	
	Example:
	
	`if (vect.empty(); // If the vector is empty`
	`cout << "The vector is empty."; // the message is displayed.`

`pop_back()`	Removes the last element from the vector.
	
	Example:
	
	`vect.pop_back(); // Removes the last element of vect, thus`
	`// reducing its size by 1.`

Member Function	Description
`push_back(value)`	Stores a value in the last element of the vector. If the vector is full or empty, a new element is created. *Example*: `vect.push_back(7); // Stores 7 in the last element of vect.`

`resize(n)`
 	
`resize(n, value)`
	Resizes a vector to have `n` elements, where `n` is greater than the vector's current size. If the optional `value` argument is included, each of the new elements will be initialized with that value.
	
	Example where `vect` currently has four elements:
	
	`vect.resize(6,99); // Adds two elements to the end of the vector,`
	`// each initialized to 99. The vector now has six`
	`// elements.`

`size()`	Returns the number of elements in the vector. *Example:* `numElements = vect.size();`
`swap(vector2)`	Swaps the contents of the vector with the contents of *vector2*. *Example*: `vect1.swap(vect2); // Swaps the contents of vect1 and vect2.`

Arrays of Objects

Concept

- Elements of arrays can be **class objects**.

- You have learned that all the elements in an array must be of the **same data type**, and you have seen arrays of many different simple data types, like `int` arrays and `string` arrays.

- However, arrays can also hold more **complex data types**, such as programmer-defined structures or objects.

- All that is required is that each element hold a structure of the same type or an object of the same class.

Arrays of Objects (continued)

- Let's look at arrays of objects.
- You define an array of objects the same way you define any array.
- If, for example, a class named `Circle` has been defined, here is how you would create an array that can hold four `Circle` objects:

```
Circle circle[4];
```

- The four objects are `circle[0]`, `circle[1]`, `circle[2]`, and `circle[3]`.
- Notice that the name of the class is `Circle`, with a capital C. The name of the array is `circle`, with a lowercase c. The convention is to begin the name of a class with a **capital letter** and the name of a variable or object with a **lowercase letter**.

Arrays of Objects (continued)

- Calling a **class function** for one of these objects is just like calling a class function for any other object, except that a **subscript** must be included to identify which of the objects in the array is being referenced.

- For example, the following statement would call the `findArea` function of `circle[2]`.

```
circle[2].findArea();
```

- Whenever an array of objects is created with **no constructor arguments**, the **default constructor**, if one exists, runs for every object in the array.

Key Points to remember about Arrays of Objects

1. The elements of an array can be objects as long as they are **objects of the same class**.

2. If you do not use an initialization list when an array of objects is created, the **default constructor** will be invoked for each object in the array.

3. It is **not necessary** that all objects in the array use the same constructor.

4. If you do use an **initialization list** when an array of objects is created, the **correct constructor** will be called for each object, depending on the number and type of arguments used.

5. If a constructor requires **more than one argument**, the initializer must take the form of a **constructor function call**.

6. If there are **fewer initializer calls** in the list than there are objects in the array, the **default constructor** will be called for all the remaining objects.

7. It is best to always provide a default constructor; but if there is none you must be sure to furnish an initializer for every object in the array.

❖ These seven statements also apply to **arrays of structures.**

Arrays of Objects (continued)

- It is also possible to create an array of objects and have another constructor called for each object. To do this you must use an **initialization list**.

- The following array definition and initialization list creates four `Circle` objects and initializes them:

```
Circle circle[NUM_CIRCLES] = {0.0, 2.0, 2.5, 10.0};
```

- This invokes the constructor that accepts one **double argument** and sets the **radii** shown here.

Object	radius
circle[0]	0.0
circle[1]	2.0
circle[2]	2.5
circle[3]	10.0

Arrays of Objects (continued)

- If the **initialization list** had been shorter than the number of objects, any remaining objects would have been initialized by the default constructor.

- For example, the following statement invokes the constructor that accepts **one double argument** for the first three objects and causes the default constructor to run for the fourth object. The fourth object is assigned a default radius of 1.0.

```
const int NUM_CIRCLES = 4;
```

```
Circle circle[NUM_CIRCLES] = {0.0, 2.0, 2.5};
```

Arrays of Objects (continued)

- To use a constructor that requires **more than one argument**, the **initializer** must take the form of a function call.

- For example, look at the following definition statement. It invokes the three-argument constructor for each of three **`Circle`** objects.

```
Circle circle[3] = { Circle(4.0, 2,  1),
                     Circle(2.0, 1,  3),
                     Circle(2.5, 5, -1) };
```

- The `circle[0]` object will have its `radius` variable set to `4.0`, its `centerX` variable set to 2, and its `centerY` variable set to `1`.

- The `circle[1]` object will have its `radius` variable set to `2.0`, its `centerX` variable set to `1`, and its `centerY` variable set to 3.

- The `circle[2]` object will have its `radius` variable set to `2.5`, its `centerX` variable set to 5, and its `centerY` variable set to `–1`.

Arrays of Objects (continued)

- It isn't necessary to call the same constructor for each object in an array. For example, look at the following statement:

```
Circle circle[3] = { 4.0,
                     Circle(2.0, 1,  3),
                     2.5 };
```

- This statement invokes the one-argument constructor for `circle[0]` and `circle[2]` and the three-argument constructor for `circle[1]`.

Arrays of Structures

- As mentioned earlier in this section, array elements can also be **structures**. This is useful when you want to store a collection of records that hold multiple data fields, but you aren't using objects.

- Because structures can hold **multiple items of varying data types**, a single array of structures can be used in place of **several arrays of regular variables**.

- An **array of structures** is defined like any other array. Assume the following structure declaration exists in a program:

```
struct BookInfo
{
   string title;
   string author;
   string publisher;
   double price;
};
```

Arrays of Structures (continued)

- The following statement defines an array, bookList, which has 20 elements. Each element is a BookInfo structure.

```
BookInfo bookList[20];
```

- Each element of the array may be accessed through a **subscript**. For example, `bookList[0]` is the first structure in the array, `bookList[1]` is the second, and so forth.

- Because members of structures are `public` by default, you do not need to use a function, as you do with class objects, to access them.

- You can access a member of any element by simply placing the `dot operator` and `member name` after the **subscript**.

```
bookList[5].title
```

Arrays of Structures (continued)

- The following loop steps **through the array**, displaying the data stored in each element:

```
for (int index = 0; index < 20; index++)
{
  cout << bookList[index].title << endl;
  cout << bookList[index].author << endl;
  cout << bookList[index].publisher << endl;
  cout << bookList[index].price << endl << endl;
}
```

- Because the members **`title`**, `author`, and `publisher` are `string` objects the individual characters making up the string can be accessed as well.

```
cout << bookList[10].title[0];
```