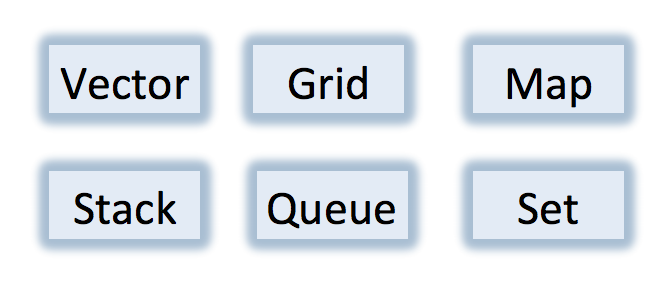
CS106B Handout   
Winter 2016 Feb 6th, 2016

Collections and Usage

We have learned about a variety of data structures so far in the course. In this handout, we will review the major data structures and how to use each one. Recall that collections have 3 major properties:

1. Defined as Classes -- they have constructors and member functions
2. Templatized -- they have a mechanism for collecting different variable types
3. Deep copy assignment -- they are taxing to pass by value, should pass by reference!

For each of the following collections, we will cover how to declare the data structure and how to manipulate the contents.



Collections get their name because they are data structures that is a grouping of some variable number of data items (possibly zero) that have some shared significance to a problem being solved and need to be operated upon together in some controlled fashion.

In CS106B (in lecture and in the book) collections are often referred to as Abstract data Types.

### Vectors

The Vector class provides the same abilities as the ArrayList class you encountered in Java. In C++, collection classes specify the type of object they contain by including the type name in angle brackets following the class name. Therefore, a Vector of ints could be declared in the two following ways:

Vector<int> vec;

Vector<int> vec();

The above constructors create an empty list and the Vector is indexed starting at 0. The following methods can be invoked on a Vector to manipulate and read the data it contains.

|  |
| --- |
|  |

#### Example code: prints out contents of a vector in a comma-separated list

void printContents(Vector<int> vec) {

cout << “[“;

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

if (i > 0) cout << “, “;

cout << vec[i];

}

cout << “]” << endl;

}

### Grids

The Grid class allows us to represent two-dimensional structures. They are helpful in representing spreadsheets, gameboards, matrices and other 2D array structures. Note that the Grid class is a Stanford collection but there is no counterpart for the class in the Standard Template Library.

Grid<int> grid; // default constructor

Grid<int> grid(2, 2); // initializes a 2x2 Grid

The default constructor will create a grid of size zero whereas the second constructor will create an empty 2x2 grid. The following methods can be invoked on a Grid to manipulate and read its data:

|  |
| --- |
|  |

#### Example code: computes the sum of the contents of a Grid of ints

int computeSum(Grid<int>& grid) {

int sum = 0;

for (int i = 0; i < grid.numRows(); i++) {

for (int j = 0; j < grid.numCols(); j++) {

sum += grid[i][j];

}

}

return sum;

}

### Stacks

Conceptually, a stack provides storage for a collection of data values with the restriction that values must be removed from a stack in the opposite order from with they were added. This implies that the last item added to a stack is *always* the first item that gets removed. The common visual representation of a stack is a literal stack of plates. To **push** a plate onto the stack you place it on top. To **pop** a plate from the stack you take the plate that was on top.

Stack<int> plates;

The above constructor would create an empty stack of integers. The following methods can be invoked on a Stack object:

|  |
| --- |
|  |

#### 

#### Example code: print lines of a file in reverse

void printLinesInReverse(ifstream& infile) {

Stack<string> lines;

while (true) {

string line; getline(infile, line);

if (infile.fail()) break;

lines.push(line);

}

while (!lines.isEmpty()) {

cout << lines.pop() << endl;

}

}

### Queue

The Queue Class uses the “first in, first out” strategy, also known as FIFO. The operations on a queue - which are analogous to the push and pop operations for stacks - are called enqueue and dequeue. The enqueue operation adds a new element to the end of the queue. The dequeue operation removes the element at the beginning of the queue.

Queue<string> line;

The above constructor would create an empty Queue of strings. The following methods can be invoked on a Queue object:

|  |
| --- |
|  |

#### Example code: print names of people in a line in order

void printLine(Queue<string> line) {

while (!line.isEmpty()) {

cout << line.dequeue() << endl;

}

}

### Maps

The Map collection is a data structure conceptually similar to a dictionary. A map holds an association between an identifying tag called a ***key*** and an associated ***value***. We also learned about the HashMap. The key difference between a HashMap and a Map is the ordering. When iterating over a Map, the for loop will return the elements in the natural order of the type. For a HashMap, the for loop will return the elements in random order.

Map<string, string> dictionary;

Map<string, double> symbolTable;

Map<string, Set<string> > pals;

The above constructors create empty maps that contain noy keys and values. Note that the constructor requires two parameters, one for the key and one for the value. You would subsequently need to add key/value pairs to the map. The following methods can be invoked on a Map object:

|  |
| --- |
|  |

#### Example code: print all the key/value pairs in a map

Map<string, double> gpa = load();

for (string name : gpa) {

cout << name << "'s GPA is ";

cout << gpa[name] << endl;

}

### Sets

The Set class is one of the most useful collection classes. This class models the mathematical abstraction of a set, which is a collection in which the elements are unordered and in which each value appears only once. We do not think of sets as having indices so you cannot use a normal for loop to iterate over the values. We also learned about the HashSet. Just like Maps, the only major difference between a HashSet and a Set is the ordering. When iterating over a Set, the for loop will return the elements in the natural order of the value type. For a HashSet, the for loop will return the elements in random order.

Set<string> friends;

The above constructor will create an empty Set of strings. The following methods can be invoked on a Set object:

|  |
| --- |
|  |

#### Example code: print all the strings in a Set of strings:

Set<string> friends = loadFriends();

for (string name : friends) {

cout << name << endl;

}