Introduction to ADTs and C++ STL

Abstract Data Types
Standard Template Library

CS 3358
Spring 2015

Jill Seaman

Roughly corresponds to chapter 7 of Weiss

Data Structure

- A particular way of storing and organizing data in a computer so that it can be used efficiently

*from wikipedia

- A data type having
  - a specific, physical representation of the data
  - operations over its data
- A concrete description
- defined in terms of how it is implemented
  - implementation-dependent

Abstract Data Type

- A set of data values and associated operations that are precisely specified independent of any particular implementation.

*from http://xlinux.nist.gov/dads/

- A data type having
  - a logical representation of the data
  - operations over its data
- A logical description
- may be implemented in various ways
  - implementation-independent

Data Structures again

- The term "data structures" is often extended to include both concrete AND logical descriptions of complicated data types.
- A list of data structures could include ADTs
  - arrays
  - linked lists
  - stacks
  - queues
  - vectors or lists in C++

Which are concrete?
Which are abstract?

Commonly used ADTs

- The purpose of many commonly used ADTs is to:
  - store a collection of objects
  - potentially organize the objects in a specific way
  - provide potentially limited access to the objects
- These ADTs are often called
  - containers
  - collections
  - container classes

Commonly used ADTs

- Examples:
  - List (or sequence or vector)
  - Set
  - Multi-set (or bag)
  - Stack and Queue
  - Tree
  - Map (or dictionary)
- Each of the above may have several variations

Stacks, Queues, and Trees will be covered later in the semester

A List ADT
(with direct access)

- **Values**: ordered (1st, 2nd, etc) set of objects
- **Operations** often include:
  - constructor: creates an empty list
  - isEmpty: is the list empty
  - size: returns the number of elements
  - add(i,e): inserts an element e at position i
  - remove(i): removes the element at position i
  - get(i): returns the element at position i
  - set(i,e) changes the element at position i to value e

A Set ADT

- **Values**: unordered collection of unique objects
- **Operations** often include:
  - constructor: creates an empty set
  - isEmpty: is the set empty
  - size: returns the number of elements
  - add(e): adds an element to the set (if not there)
  - remove(e): removes an element from the set (if it is there)
  - contains(x): true if x is in the set
  - addAll(s): adds all elements from set s to this one (union)

A Bag (multi-set) ADT

- **Values**: unordered collection of objects (may include duplicates)
- **Operations** may include:
  - constructor: creates an empty bag
  - isEmpty: is the bag empty
  - size: returns the number of elements
  - add(e): adds an element e to the bag
  - remove(e): removes one copy of an element from the bag (if it has any)
  - removeAll(e): removes all copies of e from the bag
  - occurrences(x): how many times x is in the bag

A Map ADT

- **Values**: a collection of unique keys and a collection of values where each key is associated with a single value. Keys have one type, values another.
- **Operations** may include:
  - constructor: creates an empty map
  - isEmpty: returns true if map has no key-value pairs
  - size: returns the number of key-value pairs in the map
  - get(k): returns value associated with key k (if any)
  - put(k,v): associates value v with key k (adds a pair)
  - keySet: returns a set of all the keys in the map

Implementing an ADT

- Interface (*.h):
  - class declaration
  - prototypes for the operations (interface)
  - data members for the actual (concrete) representation
- Implementation (*.cpp)
  - function definitions for the operations
  - depends on representation of data members (their concrete implementation)

Example ADT: bag version 1

bag.h

```
class Bag
{
  public:
    Bag ();

    void add(int element);
    void remove(int element);

    int occurrences(int element) const;
    bool isEmpty() const;
    int size() const;

    static const int CAPACITY = 20;

  private:
    int data[CAPACITY];
    int count;
};
```

true interface: prototypes are independent of the implementation

concrete representation, implementation dependent

what is the difference between count and CAPACITY?

Example ADT: bag version 1

bag.cpp

```
#include "bag.h"
#include <cassert>
using namespace std;

Bag::Bag () {
   count = 0;
}
void Bag::add(int element) {
    assert (count < CAPACITY);
    data[count] = element;
    count++;
}
void Bag::remove(int element) {
    int index = -1;  //change to position if found
    for (int i=0; i<count && index==-1; i++) {
        if (data[i]==element) {
            index = i;
        }
    }
    if (index!=-1) {  //found, replace w/ last elem
        data[index] = data[count-1];
        count--;
    }
}                //continued...
```

what does this do? (points to: assert (count < CAPACITY);)

Example ADT: bag version 1

bag.cpp, cont.

```
int  Bag::occurrences(int element) const {
    int occurrences=0;
    for (int i=0; i<count; i++) {
        if (data[i]==element) {
            occurrences++;
        }
    }
    return occurrences;
}

bool Bag::isEmpty() const {
    return (count==0);
}

int  Bag::size() const {
    return count;
}
```


bag “driver”

bagTest.cpp

```
#include<iostream>
#include "Bag.h"
using namespace std;

int main ()
{
    Bag b;

    b.add(4);
    b.add(8);
    b.add(4);

    cout << "size " << b.size() << endl;
    cout << "how many 4's: " << b.occurrences(4) << endl << endl;

    b.remove(4);
    cout << "removed a 4" << endl;
    cout << "size " << b.size() << endl;
    cout << "how many 4's: " << b.occurrences(4) << endl << endl;
```


bag “driver”

bagTest.cpp

```
    Bag c(b);

    cout << "copied to c" << endl;
    cout << "size " << c.size() << endl;
    cout << "how many 4's: " << c.occurrences(4) << endl << endl;

    b.add(10);
    cout << "added 10 to b" << endl;
    cout << "b.size " << b.size() << endl;
    cout << "c.size " << c.size() << endl << endl;

    cout << "starting to add 20 items" << endl;
    for (int i=0; i<20; i++)
        b.add(33);
    cout << "added 20 more items to b" << endl;

    return 0;
};
```

bag “driver”: output

output of running bagTest

```
size 3
how many 4's: 2

removed a 4
size 2
how many 4's: 1

copied to c
size 2
how many 4's: 1

added 10 to b
b.size 3
c.size 2

starting to add 20 items
Assertion failed: (count < CAPACITY), function add, file
bag.cpp, line 12.
Abort trap: 6
```


Bag version 1 summary

- Implemented using a fixed size array
- When adding more elements than fit in the bag, the program exits.
- Solution:
  - use a dynamically allocated array
  - when its capacity is reached, allocate a new, bigger array.

bag version 2

bag.h

“The big three”

concrete representation

```
class Bag
{
  public:
    Bag ();

    Bag(const Bag &);
    ~Bag();
    void operator=(const Bag &);

    void add(int element);
    void remove(int element);

    int occurrences(int element) const;
    bool isEmpty() const;
    int size() const;

    static const int INCREMENT = 20;

  private:
    int *data;     //pointer to bag array
    int capacity;  //size of the array
    int count;     //number of elements currently in array
};
```


bag version 2

bag.cpp

```
Bag::Bag () {
    count = 0;
    capacity = INCREMENT;
    data = new int[capacity];
}

//copy constructor
Bag::Bag(const Bag &rhs) {
    data = new int[rhs.capacity];   //allocate new array

    capacity = rhs.capacity;        //copy values
    count = rhs.count;
    for (int i=0; i<count; i++) {
        data[i] = rhs.data[i];
    }
}

//destructor
Bag::~Bag() {
    delete [] data;
}
```

bag version 2

bag.cpp, cont.

```
void Bag::operator=(const Bag &rhs) {
    if (data) delete [] data;      //delete old array
    data = new int[rhs.capacity];  //allocate new array

    capacity = rhs.capacity;       //copy values
    count = rhs.count;
    for (int i=0; i<count; i++) {
        data[i] = rhs.data[i];
    }
}

void Bag::add(int element) {
    //if count is at the capacity, resize
    if (count==capacity) {
        capacity += INCREMENT;
        int *newData = new int[capacity];  //new array
        for (int i=0; i<count; i++) {      //copy values
            newData[i] = data[i];
        }
        delete [] data;       //delete old array
        data = newData;       //make data point to new
    }

    data[count] = element;    //add new element
    count++;
}
```

no changes to remaining functions!

bag "driver": output version 2

output of running bagTest

```
size 3
how many 4's: 2

removed a 4
size 2
how many 4's: 1

copied to c
size 2
how many 4's: 1

added 10 to b
b.size 3
c.size 2

starting to add 20 items
added 20 more items into b
```

resizing succeeded!

C++ STL: Standard Template Library

- A library of ADTs implemented in C++
- Two categories of STL ADTs:
  - containers: classes that store a collection of data and impose some organization on it
  - iterators: behave like pointers; a mechanism for accessing elements in a container the iterator is associated with.

STL Containers: sequence

- Two categories of STL Containers:
- sequence containers: organize and access data sequentially, as in an array:
  - **`vector:`** expandable array, values are quickly added to or removed from the end of the list.
  - **`deque:`** like a vector, but can add values quickly to front and end of the list.
  - **`list:`** can add values quickly anywhere in its sequence, but does not provide random access.

Note the emphasis on performance. Not so abstract ADTs.

STL Containers: associative

- <u>associative containers</u>: use keys to allow data elements to be quickly accessed. These include:
  - **`set`:** stores a set of keys, no duplicates allowed.
  - **`multiset`:** stores a set of keys, duplicates are allowed.
  - **`map`:** maps a set of keys to values, the keys must be unique (but the values may appear multiple times).
  - **`multimap`:** maps a set of keys to values, keys are not unique (one key can have many values).

STL Iterators:

- <u>iterators</u>: Generalizations of pointers, used to access data stored in containers.
- They point to a certain value (or the *past-the-end* element).
- They may be dereferenced with *.
- Some types of iterators:
  - **`forward`:** uses ++ to advance to next element.
  - **`bidirectional`:** uses ++ and --.
  - **`random access`:** uses ++ and -- and uses [i] to jump to a specific element.

Some vector member functions

- **`size()`:** returns number of elements in the vector.
- **`push_back(x)`:** inserts x at end of vector (increases size by 1)
- **`pop_back()`:** removes the last element from the vector (decreases size by 1)
- **`operator[i]`:** allows random access to specific element (i must be less than the size of the vector).
- **`begin()`:** returns an iterator pointing to the vector's first element.
- **`end()`:** returns an iterator pointing to the vector's *past-the-end* element.

Sample code using vectors+iterators

```
#include <iostream>
#include <vector>    // Include the vector header
using namespace std;

int main()  {
   int count;         // Loop counter
   vector<int> vect;  // Define a vector of int object
   vector<int>::iterator iter; // Defines an iterator object

   // Use push_back to push values into the vector.
   for (count = 0; count < 10; count++)
      vect.push_back(count);

   // Step the iterator through the vector to display:
   cout << "Here are the values in vect: " << endl;
   for (iter = vect.begin(); iter < vect.end(); iter++) {
      cout << *iter << " ";
   }

   // Step the iterator through the vector backwards.
   cout << "and here they are backwards: " << endl;;
   for (iter = vect.end() - 1; iter >= vect.begin(); iter--) {
      cout << *iter << " ";
   }
}
```

Vector member function using iterator

- **`erase(iter)`:** Removes from the vector either the single element the iterator argument is referring to.
- erase reduces the vector size by 1.

```
int main ()
{
  vector<int> myvector;

  // set some values (from 1 to 10)
  for (int i=1; i<=10; i++) myvector.push_back(i);

  // erase the 6th element
  myvector.erase (myvector.begin()+5);  //advances 5 times

  cout << "myvector contains:";
  for (int i=0; i<myvector.size(); i++)
    cout << ' ' << myvector[i];
  cout << endl;
}
```