ICOM 4035 – Data Structures

Dr. Manuel Rodríguez Martínez
Electrical and Computer Engineering Department

Lecture 5 – September 4th, 2001

Readings

- Read Handout about Container Classes
  - Available from Engineering Reproduction Center as ICOM-4035-Manual # 3

Container Classes

- A container class is a class used to create objects that store collections of other objects.
  - Examples:
    - List of names in a video store.
    - Set of integers that appear in a login name.
- A container class is a data structure.
- There are many types of containers, depending on the way that the objects are organized within the container.
  - Vector – elements are not sorted
  - Set – elements are unique (no duplicates)
  - Sequence – elements are unique and are sorted

Container Classes (cont.)

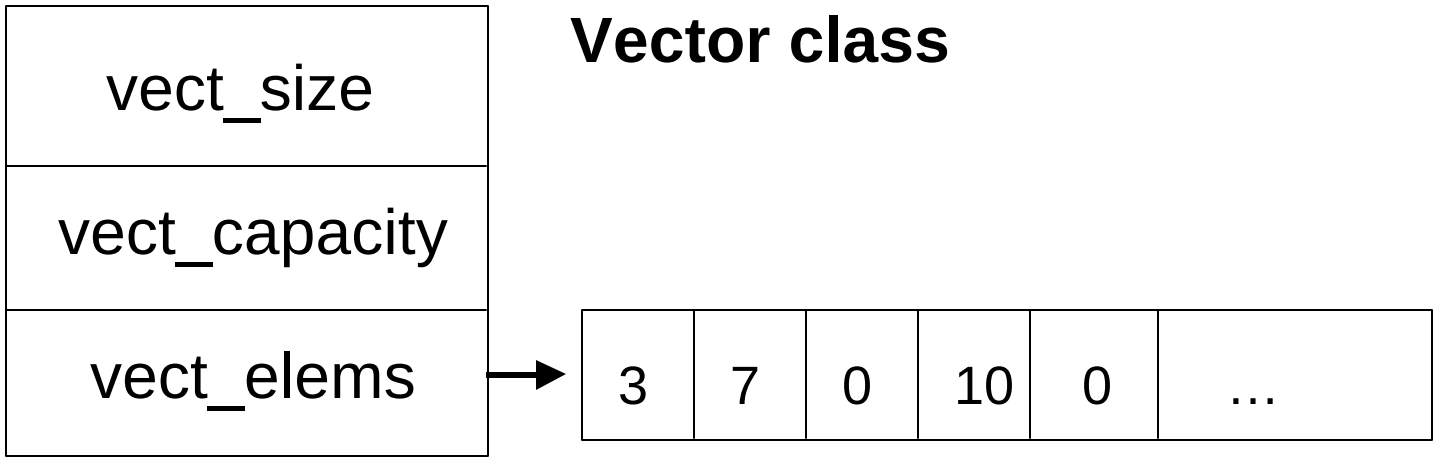
- C++ provides the Standard Template Library which includes several container classes.
  - Vector
  - Deque
  - List
  - Set
  - Multi-Set (Bag)
  - Map (Hash Table)
  - Multi-Map
- Good containers should be re-usable
  - By using typedef
  - Templates – preferred way.

Vector Container

- Enhancement over the built-in array
  - C++ books recommend vector over built-in array

- The vector has the following features
  1. It knows its size – keeps track of number of elements
  2. Can grow or shrink on demand – the number of elements can be reallocated to make it larger or smaller dynamically
  3. Can perform index bounds checking – can catch error when you try to access illegal elements (e.g. v[-1])
     - Not all implementations do this, for example STL performs no bounds checking because of the overhead incurred.

Vector Organization



vect_size – number of element (probably) in use
vect_capacity – number of elements allocated (max size)
vect_elems – pointer to the array of elements

Vector Methods

- Constructors
  - Based on size – vector(int init_size)
  - Copy constructor – vector(const vector& vect)
- Destructor
  - Need to free pointer to elements
- Accessors
  - Get the current size
  - Get the current capacity
  - Access element *i* in the vector
- Mutators
  - Copy assignment
  - Resize the vector
  - Reallocate the vector
  - Add a new element at the end and make vector grow if need be
  - Assign element i in the vector

Vector class interface (vector.h)

Object.h

```
#ifndef _OBJECT_H
#define _OBJECT_H
typedef int Object
#endif
```

vector.h

```
#ifndef _VECTOR_H
#define _VECTOR_H
#include "Object.h"
class vector{
    public:
        explicit vector(int init_size = 0);
        vector(const vector& vect);
        ~vector();
        const vector& operator=(const vector& vect);
        Object& operator[](int index);
        const Object& operator[](int index) const;
        int size() const;
        int capacity() const;
        void resize(int new_size);
        void reserve (int new_capacity);
    private:
        int vect_size;
        Int vect_capacity;
        Object *vect_elems;
};
#endif
```

Vector implementation (vector.cc)

```
#include “vector.h”
#include <cassert>
// constructor based on size
vector::vector(int init_size)
        : vect_size(init_size), vect_capacity(init_size){
          vect_elemns = new Objects[vect_capacity];
}
// copy constructor
vector::vector(const vector& vect)
                :vect_size(vect.size()),
                vect_capacity(vect.capacity()),
                vect_elems (NULL) {

        int i=0;
        vect_elems = new Objects[vect_capacity];
        for (i=0; i < vect_size; ++i){
                vect_elems[i] = vect[i];
        }
}
// destructor
vector::~vector() {
        delete [] vect_elems;
        vect_size = vect_capacity = 0;
}
```

Vector Implementation II (vector.cc)

```
const vector& vector::operator=(const vector& vect){
    int i=0;
    if (this != &vect){
        vect_size = vect.size();
        vect_capacity = vect.capacity();
        // recycle old elements
        delete [] vect_elems;
        // reallocate new elements
        vect_elems = new Objects[vect_capacity];
        for (i=0; i < vect_size; ++i){
            vect_elems[i] = vect[i];
        }
    }
    return *this;
}
int vector::size() const{
    return vect_size;
}
int vector::capacity() const{
    return vect_capacity;
}
```

Vector Implementation III (vector.cc)

```
void vector::reserve(int new_capacity){
    Object *old = vect_elems;
    int num_copy = 0;

    assert(new_capacity > 0);
    num_copy = (new_capacity < vect_size ?
    new_capacity : vect_size);
    vect_capacity = new_capacity;
    vect_size = num_copy;

    vect_elems = new Objects[vect_capacity];
    for(i=0; i < num_copy; ++i){
        vect_elems[i] = old[i];
    }
    delete [] old;
}
void vector::resize(int new_size){
    assert(new_size > 0);
    if (new_size > vect_capacity){
        reserve(new_size * 2);
    }
    vect_size = new_size;
}
```

Vector Implementation IV (vector.cc)

```
void vector::push_back(const Object& new_obj){
        if (vect_size == vect_capacity){
                resize(vect_capacity * 2 + 1);
        }
        vect_elems[vect_size++] = new_obj;
}

Object& vector::operator[](int index){
        assert(((index >=0) && (index < vect_capacity));
        return vect_elems[index];
}

const Object& operator[](int index) const {
        assert(((index >=0) && (index < vect_capacity));
        return vect_elems[index];
}
```

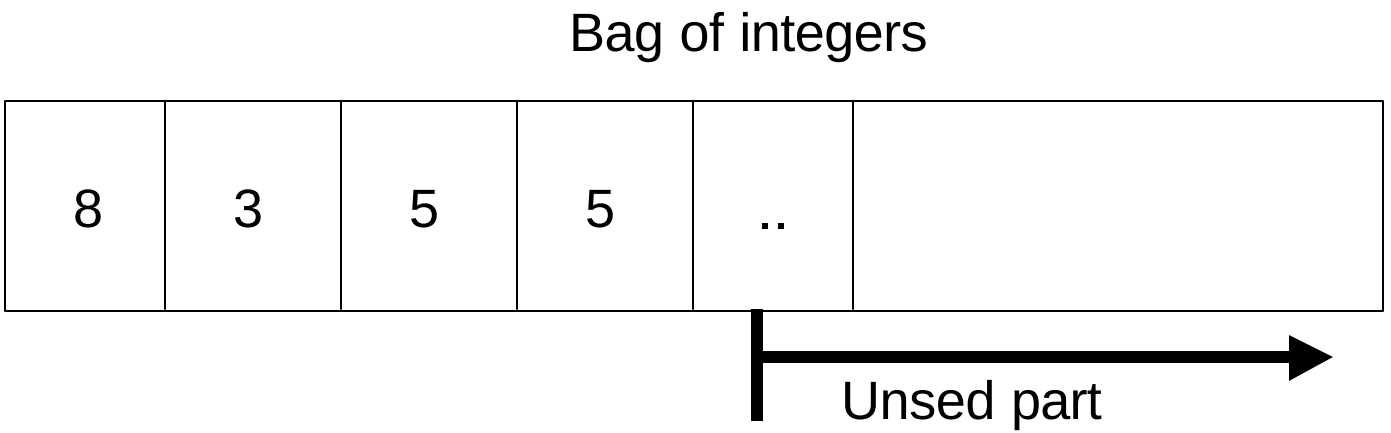
Using the vector class

```
#include<iostream>
#include <cassert>
#include “vector.h”
int main(){
    vector v;
    int n=0, num=0, sum=0;

    cout << "This program adds numbers" << endl;
    cout << "How many number to add: " << endl;
    cin >> num;
    v = vector(num);
    cout << "Start typing the numbers" << endl;

    for (int i=0; i < num; ++i){
        cin >> n;
        vect.push_back(n);
        // can also use: v[i] = n
    }

    // add them;
    for (i=0; i < vect.size(); ++i){
        sum += v[i];
    }
    cout << "Total sum is : " << sum << endl;
    return 0;
}
```

Bag Class Container

- Bag is data structure used to store elements with the following semantics:
  - Copies of the same element can stored in the bag B.
  - A find operation is supported to determine if an element x is present in the bag B.
  - An erase operation is supported to erase an instance of an element x in the bag B.
  - An erase all operation is supported to erase all instances of an element x in the bag B
  - A union operation is supported to concatenate the contents of two bags.

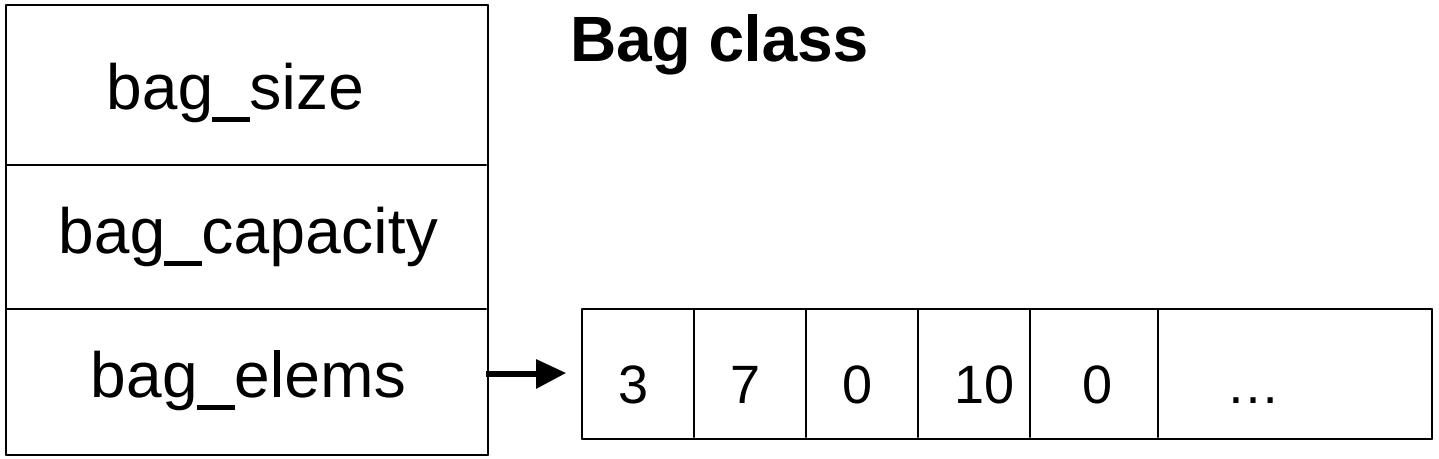
Bag Class Conceptual Example



Operation:

B.count(8) = 1
B.count(5) = 2
B.count(7) = 0

Bag Class Design



Wait a minute!!!
We can use vector for this!
add more semantics to implement:
count()
insert()
erase()

Bag Class Methods

- Constructor
  - Make an empty bag
  - Make a bag from another bag (copy constructor)
- Accessor
  - Get current size
  - Get current capacity
  - Get the count of an element
- Mutators
  - Insert a new element
  - Erase one instance of element x
  - Erase all instances of element x
- Non-member
  - Addend all elements from two bags to create a new one.