CSE 143

Dynamic Memory In Classes

[Chapter 4, p 156-157, 172-177]

Remember Class Vector?

```
class Vector {
public:
  Vector ( );
  bool   isEmpty( );
  int    length ( );
  void   vectorInsert (int newPosition, Item newItem);
  Item   vectorDelete (int position);
  Item   vectorRetrieve (int position);
  ...
  };
```


Many Ways to Implement

- Version 1: With fixed length arrays
  - Very efficient to access individual elements
  - Limited in size, flexibility
- Version 2: With a linked list
  - Very flexible in size
  - Inefficient to access individual elements

    `Have to chase pointers down the list`
- Here's a third way:
  - Use an array (for efficient access)
  - Make the array itself "dynamic"

    `Reallocate a larger array as needed to grow`

Vector Implementation

```
class Vector {
  public:
    // constructors and other methods, as before
  private:
    Item *Items;  // items[0..capacity] is space allocated for
                  // this vector
    int  size;    // items are stored in Items[0..size-1]
    int capacity;  //current maximum array size

    // might need additional private helper functions

};
```


Draw the picture!

Vector Constructor

```
Vector::Vector() {
     // set up private variables
    capacity = DEFAULT_CAPACITY;
    size = 0;
     // allocate memory
    items = new Item[capacity];
     // what goes here?
}
```

Except for this, the public methods can be the same as for the fixed array implementation.

Exception: insert needs to insure there is room to add a new item.

Useful Private Functions

```
class Vector {
  public:
    // constructors and other methods
  private:
    // data members here...
    // ensure the Vector can hold at least n elements
    void ensureCapacity(int n);
    // set size of the Vector to n elements
    void growArray(int n);
};
```


ensureCapacity( )

```
// ensure that Vector can hold at least n
// elements
void Vector::ensureCapacity(int n) {
     // return if existing capacity is ok
     if (capacity >= n)
        return;
     // out of space: double capacity
     int newCapacity = capacity * 2;
     if (newCapacity < n)
        newCapacity = n;
     // grow the array
     growArray(newCapacity);
}
```


growArray( )

```
// Set size of vector to newCapacity
void Vector::growArray(int newCapacity) {
   Item *newItems = new Item[newCapacity];
   if(newItems == NULL){ … } //handle error
   for (int i = 0; i < size; ++i)
      newItems[i] = items[i]; //copy items
...
   items = newItems;
   capacity = newCapacity;
}
```

Have we forgotten anything?

Now insert is easy!

```
// insert newItem at newPosition in Vector
void Vector::vectorInsert(int newPosition,
                          Item newItem) {
     // make room
     ensureCapacity(size+1);
     // shift data over
     for (int i=size; i > newPosition; --i)
       items[i] = items[i-1];
     // store the item
     size++;
     items[newPosition] = newItem;
}
```


Issues with Dynamic Memory

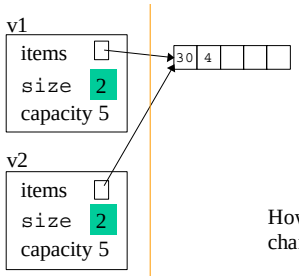
- Using dynamic memory in classes raises issues
- Familiar dangers:
  - Dangling pointers, Uninitialized pointers, Memory leaks, etc.
- **Some new complications:**
  - Many of them arise when objects are copied
    `Copied automatically when passed as params, etc.`
    `Copied explicitly by programmer`
  - Other dangers when objects are deleted
    `Explictly deleted, or just go out of scope`
  - C++ has some special features to help the situation

Innocence Destroyed (I)

```
    // assume Item == int
Vector v1, v2;
v1.insert(0, 30);
v1.insert(1, 4);
v2 = v1;
v2.delete(0);
```

- //Draw the picture and weep!

After v2=v1, Before v2.delete



How does v2.delete change the picture?

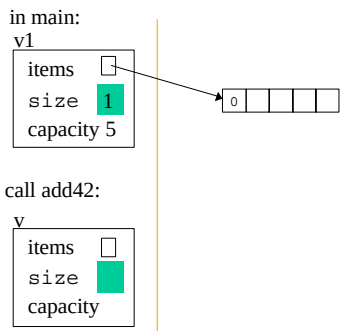
Innocence Destroyed (II)

void add42 (Vector v) { //add 42 to front of vector
v.insert(0, 42); }

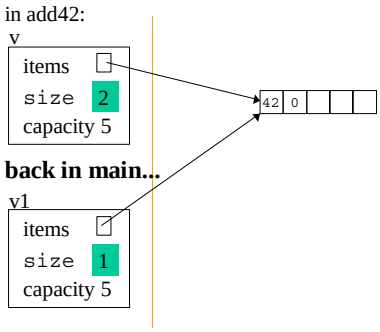
//code in main
Vector v1;
v1.insert(0, 0);
add42(v1);

- v1 passed by value, so no harm done -- right??
- Draw the picture and weep!

After v1.insert(0,0)



After v.insert(0, 42);...



The Culprit: "Shallow Copy"

- When structs and classes are copied, all and only the member variables are copied
- When there's dynamic memory, that's not enough
  - Example: the `items` pointer value is copied, so the copy points to the same place
  - Can lead to surprises and bugs
- Solution: need a concept of "deep copy"

More copy problems

- The problem with deep vs. shallow copying can appear in these contexts:
  - Assignment of one object to another
  - Initialization in a variable declaration:

```
SomeClass f1;
SomeClass f2 = f1;
```

  - Passing a copy of an actual to a formal parameter (*pass-by-value*)
  - Returning an instance as the value of a function:

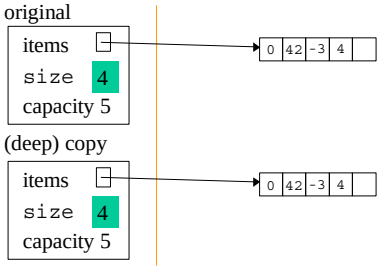
```
return someIntVector;
Why? because a function returns a new, temporary object
```

Needed: Deep Copy

- A "deep copy" should make a complete new copy, including new dynamic memory
- Need a way to make the deep copy happen *automatically* when appropriate
  - Vector v1 = v2;
  - v1 = v2;
  - func1(v1);
  - return v1;
- PS: this won't solve the problem of cleaning up dynamic memory used by local variables
  - We'll get back to that

"Deep copy"

- A deep copy makes a completely independent copy, by allocating more dynamic memory



Deep copy for Vector

- Deep copy logic:
  - Initialize the new vector to empty.
  - For each element in the vector:
    `add it to the new vector`
- Could be a client function
  - *void copyVector (Vector &orig, Vector &newVec);*
  - use member functions like length, retrieve, insert, etc.
- Could be a public or private member function
  - void Vector::copy (Vector &orig);
  - copies from orig to current vector
  - use private data directly

Making It Automatic

- Problem with copyVector: *must be called explicitly*
- We need it to happen *automatically* in certain cases
- Solution: C++ allows a "Copy Constructor"
  - Will be called automatically in certain cases where an object must be initialized from an existing object
- Compiler recognizes it as a constructor with a special parameter list: *(classname &)*
  - or *(const classname &)*

Copy Constructor for Vector Class

```
class Vector {
public:
   Vector ();
   Vector(const Vector &);
...
}
```

- Compiler recognizes this as a copy constructor
- Will call automatically when
  - passing arguments by value
  - initializing variable with = in a variable declaration
  - copying a return value

Inside the Constructor

- It's just a function, it can do anything!
- But... what you normally write is a deep copy
- For our Vector copy constructor:
  - could call a previously defined copyVector function
  - could build the new copy directly
- If you don't define your own copy constructor, the compiler generates a default copy constructor
  - Does a ***shallow*** copy

Look at the code:

```
Vector::Vector(Vector &other) { copy(other); }

// private member function: replace this Vector
// with a deep copy of other
void Vector::copy(Vector &other) {
      // set up private variables
      capacity = other.capacity;
      size = other.size;
      // allocate memory
      items = new Item[capacity];
      if(items == NULL){ … } // handle error
      // copy data
      for (int i = 0; i < size; ++i)
         items[i] = other.items[i];
}
```


Technicalities of '='

- These two look almost identical, but they are NOT the same:
  a) Vector MyVector = YourVector;
  b) Vector MyVector;
     MyVector = YourVector;
- The difference in technical terms:
  - in a), MyVector is being created
    The copy constructor **is** called (if there is one)
    This is a variable declaration with an initializer - it is ***not*** an assignment statement.
  - in b), MyVector already exists
    This ***is*** an assignment statement.
    A copy constructor **is not** called, because we are not constructing a new object!
    For example, the object may already contain heap pointers

Solution: Overloaded =

- To handle the assignment statement, the copy constructor won't do.
- Instead, we can define an "overloaded assignment operator"
  - Slightly weird syntax:
    Vector & Vector::operator = (Vector &other) {
    //The code for this function could perform a deep copy
    }
  - This would normally be a public method
  - Class interface declaration would contain
    Vector & operator = (Vector &other);
    Or (explained later): Vector & operator = (const Vector &other);

Implementing operator =

Four important parts:

1. Test for same object:
   - `if (&other != this) { /* copy code */ }`
2. Delete old dynamically allocated data
   - call `cleanup()` function, or
   - directly (in our case): `delete [] items;`
3. Copy new data
   - call `copy()` if you have one
4. Return a copy (really a ref.) of the current object:
   - `return *this;`

And the code...

```
Vector & Vector::operator=(Vector &other) {
   if (&other != this) //step 1
   {
      cleanup();      //step 2
      copy(other);    //step 3
   }
   return *this;      //step 4
}
// private member function
void Vector::cleanup() {
   delete [] items;
}
```


Detour: *this*

- A reserved word in C++
- Means "a pointer to the current object"
- Like a hidden 0th parameter to *all* member functions
  - int Vector::length(Vector *this) { ... }
  - only exists when you are in member functions!
- Can use like any other pointer
  - Vector *vp = this;
  - if (vp == this) ...
  - return this->size;
  - this->capacity = this->capacity * 2;
  - return *this;

Another Use For *This*

```
class Point {
  public:
    Point();
    Point(int x);
    Point(double x);

  private:
    int x,y; …
};
```

```
Point::Point() {
  x = 5; }  //nothing new

Point::Point(double x) {
  x = (int) x; }//useless

Point::Point(int x) {
  this->x = x; }//useful
```

```
Point p1;
Point p2(3.14};
Point p3(42);
```

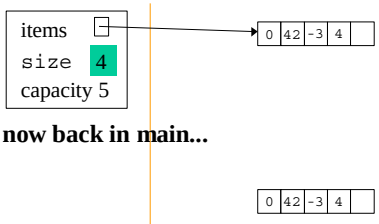

Innocence Destroyed (III)

void MyFunction () {
Vector tempVector; //local variable
// build a temporary vector for whatever reason
...
}

- When a function exits
  - local variables are automatically destroyed
  - so having a local Vector is no problem -- right?
- Draw the picture and weep!

Local variable goes away...

tempVector (in MyFunction)



now back in main...

Next Problem: Cleanup

- When a function exits, only the local memory is released
  - Dynamic memory pointed to inside the variable stays allocated
  - results in a memory leak
    `unless there is another pointer to the data`
- One solution: write a function to delete the allocated dynamic memory
  - E.g., the cleanup() function we used in operator =
  - For Vector, this would be simply `delete [] items;`
  - Drawback: you (or client) must remember to call the function

C++ Solution: A "Destructor"

- Called automatically to de-construct the object
  - When it goes out of scope (e.g. end of function)
  - When *delete* operator used
- Can contain most any code
  - Normally it would contain code to release all dynamically allocated memory
- Special syntax identifies it:
  ~classname ()
  - no return value
  - no arguments allowed
- The compiler-generated default destructor does nothing.

Vector Destructor I

```
//Dynamic array version
//use same "cleanup" function
  needed by deep copy
Vector::~Vector()
{
  cleanup();
}
```

Vector Destructor II

```
//destructor for linked list implemention
Vector::~Vector()
{
   Node *p = head;
   while (p != NULL) {
      Node* prev = p;
      p = p->next;
      delete prev;
   }
}
```


Wise Advice

- When defining a class which uses dynamic memory, ALWAYS provide
  - a default constructor
  - a deep copy method
  - a copy constructor (calls the deep copy method)
  - an overloaded assignment operator (calls the deep copy)
  - a destructor
- It may seem like unnecessary work, but will save you (and your readers) from nasty surprises.

Constructor Puzzle

- Assume the class Vector has all of the following defined: **DC**: default constructor; **CC**: copy constructor; **op =**: overloaded assignment operator; **D**: destructor
- On each line, say if DC, CC, op =, or D is called.

```
Vector  puzzlfunction (Vector & v1) {  //line 0
   Vector v2;                          //line 1
   Vector Varray[40];                  //line 2
   Vector v3 = v1;          //line 3
   v2 = v1;                            //line 4
   v2.VectorInsert(1, 0);              //line 5
   Vector * v4;                        //line 6
   v4 = new Vector;                    //line 7
   delete v4;                          //line 8
   printVector(v2);                    //line 9
   return (v2);                        //line 10 (tricky)
}                                      // line 11(trickier)
```


More Wrinkles

- Classes within classes, i.e., member variables which are themselves classes
  - Have to know what order the constructors are called in
    Answer: bottom up
  - Have to know what order destructors are called in
    Answer: top down
  - Special syntax for calling non-default constructors of member variables within outer-level constructors
    "member initializer list" in implementation
    trivial examples p.172, 173
- Nothing is ever as simple as it seems in C++!

Where We're Headed

- We know the C++ features for dynamic memory
- We know how to package ADTs that use dynamic memory
- Armed with this... we can begin to investigate a series of interesting and useful data structures and ADTs. For each one:
  - What the ADT is (abstractly)
  - How to implement (often more than one way)
  - Applications