# CS 106X
# Arrays; Implementing a Collection (ArrayList)

guest presenter:
Marty Stepp (stepp AT cs DOT stanford DOT edu)

reading:

*Programming Abstractions in C++*,
11.2 - 11.3; 12.1, 12.3, 12.7; 14.1, 14.4



# Motivation

- So far in this course, we have **used** many collection classes:
  - Vector, Grid, Stack, Queue, Map, Set, HashMap, HashSet, Lexicon, ...

- Now let's explore how they are **implemented**.
  - We will start by implementing our own version of a **Vector class**.
    - To do so, we must learn about **arrays** and **memory** allocation.
  - After that, we will implement several other collections:
    - linked list; binary tree set, map; hash table set, map; prefix tree (Lexicon); priority queue; graph; ...

# Arrays (11.3)

```
type* name = new type[length];
```

- A **dynamically allocated** array.
- The variable that refers to the array is a **pointer**.
- The memory allocated for the array must be manually released, or else the program will have a **memory leak**. (>_<)

– Another array creation syntax that we will not use:

```
type name[length];
```

- A fixed array; initialized at declaration; can never be resized.
- Stored in a different place in memory; the first syntax uses the *stack* and the second uses the *heap*. *(discussed later)*

# Initialized?

```
type* name = new type[length];   // uninitialized
type* name = new type[length](); // initialize to 0
```

- If ( ) are written after the array [ ], it will set all array elements to their default zero-equivalent value for the data type. *(slower)*
- If no ( ) are written, the elements are uninitialized, so whatever garbage values were stored in that memory beforehand will be your elements.

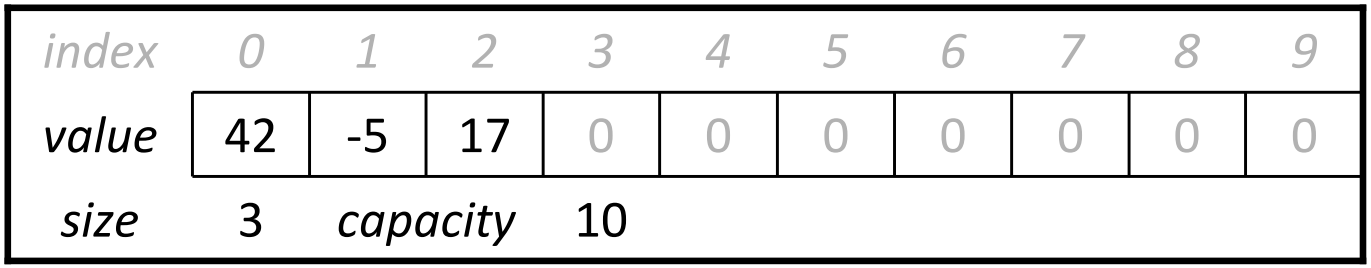
```
int* a = new int[3];
cout << a[0];                // 2395876
cout << a[1];                // -197630894

int* a2 = new int[3]();
cout << a[0];                // 0
cout << a[1];                // 0
```

# How a Vector works

- Inside a Vector is an **array** storing the elements you have added.
  - Typically the array is larger than the data added so far, so that it has some extra slots in which to put new elements later.
    - We call this an *unfilled array*.

```
Vector<int> v;
v.add(42);
v.add(-5);
v.add(17);
```



# Exercise

- Let's write a class that implements a growable array of integers.
  - We'll call it `ArrayList`. It will be very similar to the C++ `Vector`.
  - its behavior:
    - `add(`***`value`***`)`
    - `get(`***`index`***`),`
    - `size(),`
    - `remove(`***`index`***`)`
    - `indexOf(`***`value`***`),`
    - `toString()`

    `...`

    `insert(`***`index, value`***`)`

    `set(`***`index, value`***`)`

    `isEmpty()`

    `contains(`***`value`***`)`
  - The list's *size* will be the number of elements added to it so far.
    - The actual array length ("capacity") in the object may be larger. We'll start with an array of **length 10** by default.

# ArrayList.h

```
#ifndef _arraylist_h
#define _arraylist_h
#include <string>
using namespace std;

class ArrayList {
public:
    ArrayList();
    void add(int value);
    void clear();
    int get(int index) const;
    void insert(int index, int value);
    bool isEmpty() const;
    void remove(int index);
    void set(int index, int value);
    int size() const;
    string toString() const;

private:
    int* myElements;   // array of elements
    int myCapacity;    // length of array
    int mySize;        // number of elements added
};

#endif
```

# Implementing add

- How do you append to the end of a list? `list.add(42);`
  - place the new value in slot number *size*
  - increment *size*

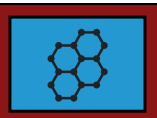
| *index* | *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* |
|---|---|---|---|---|---|---|---|---|---|---|
| *value* | 3 | 8 | 9 | 7 | 5 | 12 | 0 | 0 | 0 | 0 |

*size* 6 *capacity* 10

| *index* | *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* |
|---|---|---|---|---|---|---|---|---|---|---|
| *value* | 3 | 8 | 9 | 7 | 5 | 12 | **42** | 0 | 0 | 0 |

*size* **7** *capacity* 10

# add solution



```
// in ArrayList.cpp
void ArrayList::add(int value) {
    myElements[mySize] = value;
    mySize++;
}
```

# Implementing insert

- How do you insert in the middle of a list? `list.insert(3, 42);`
  - shift elements right to make room for the new element
  - increment *size*

| *index* | *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* |
|---|---|---|---|---|---|---|---|---|---|---|
| *value* | 3 | 8 | 9 | 7 | 5 | 12 | 0 | 0 | 0 | 0 |

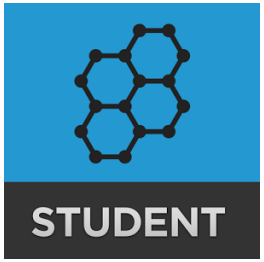
*size* 6 *capacity* 10

| *index* | *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* |
|---|---|---|---|---|---|---|---|---|---|---|
| *value* | 3 | 8 | 9 | **42** | 7 | 5 | 12 | 0 | 0 | 0 |

*size* **7** *capacity* 10

**Q:** In which direction should our array-shifting loop traverse?

**A.** left-to-right **B.** right-to-left **C.** either is fine



# insert solution

```
// in ArrayList.cpp
void ArrayList::insert(int index, int value) {
    // shift right to make room
    for (int i = mySize; i > index; i--) {
        myElements[i] = myElements[i - 1];
    }
    myElements[index] = value;
    mySize++;
}
```

# Other members

- Let's implement the following member functions in our list:
  - `size()` - Returns the number of elements in the list.
  - `get(`***`index`***`)` - Returns the value at a given index.
  - `set(`***`index, value`***`)` - Changes the value at the given index.
  - `isEmpty()` - Returns `true` if list contains no elements.
    - (Why bother to write this if we already have a `size` function?)
  - `toString()` - String of the list such as `"{4, 1, 5}"`.
  - `operator <<` - Make the list printable to `cout`

# Other members code

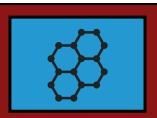
```
// in ArrayList.cpp
int ArrayList::get(int index) {
    return myElements[index];
}

void ArrayList::set(int index, int value) {
    myElements[index] = value;
}

int ArrayList::size() {
    return mySize;
}

bool ArrayList::isEmpty() {
    return mySize == 0;
}
```

# Other members code



```
// in ArrayList.cpp
ostream& operator <<(ostream& out, const ArrayList& list) {
    out << "{";
    if (!list.isEmpty()) {
        out << list.get(0);
        for (int i = 1; i < list.size(); i++) {
            out << ", " << list.get(i);
        }
    }
    out << "}";
    return out;
}

string ArrayList::toString() const {
    ostringstream out;
    out << *this;
    return out.str();
}
```

# Implementing remove

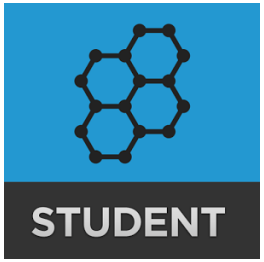
- How do you remove an element from a list? `list.remove(2);`
  - shift elements left to cover the deleted element
  - decrement *size*

| *index* | *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* |
|---|---|---|---|---|---|---|---|---|---|---|
| *value* | 3 | 8 | **9** | 7 | 5 | 12 | 0 | 0 | 0 | 0 |
| *size* | 6 | *capacity* | 10 | | | | | | | |

| *index* | *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* |
|---|---|---|---|---|---|---|---|---|---|---|
| *value* | 3 | 8 | 7 | 5 | 12 | 0 | 0 | 0 | 0 | 0 |
| *size* | **5** | *capacity* | 10 | | | | | | | |

**Q:** In which direction should our array-shifting loop traverse?

**A.** left-to-right **B.** right-to-left **C.** either is fine



# remove solution

```
// in ArrayList.cpp
void ArrayList::remove(int index) {
    // shift left to cover up the slot
    for (int i = index; i < mySize; i++) {
        myElements[i] = myElements[i + 1];
    }
    myElements[mySize - 1] = 0;
    mySize--;
}
```

# Freeing array memory

`delete[] name;`

- Releases the memory associated with the given array.
- Must be done for all arrays created with new
  - Or else the program has a *memory leak*. (No garbage collector like Java)
  - Leaked memory will be released when the program exits, but for long-running programs, memory leaks are bad and will eventually exhaust your RAM.

```
int* a = new int[3];
a[0] = 42;
a[1] = -5;
a[2] = 17;
for (int i = 0; i < 3; i++) {
    cout << i << ": " << a[i] << endl;
}
...
delete[] a;
```

# Destructor (12.3)

```
// ClassName.h
~ClassName();
```

```
// ClassName.cpp
ClassName::~ClassName() { ...
```

- **destructor**: Called when the object is deleted by the program.
  (when the object goes out of { } scope; opposite of a constructor)

  - Useful if your object needs to do anything important as it dies:
    - saving any temporary resources inside the object
    - freeing any dynamically allocated memory used by the object's members
    - ...

  - Does our `ArrayList` need a destructor? If so, what should it do?

# Destructor solution

```
// in ArrayList.cpp
void ArrayList::~ArrayList() {
    delete[] myElements;
}
```

# Running out of space

- What if the client wants to add more than 10 elements?

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| value | 3 | 8 | 9 | 7 | 5 | 12 | 4 | 8 | 1 | 6 |

size 10 capacity 10

  - `list.add(75);  // add an 11th element`

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| value | 3 | 8 | 9 | 7 | 5 | 12 | 4 | 8 | 1 | 6 | **75** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

size **11** capacity **20**

  - Answer: **Resize the array** to one twice as large.
    - Make sure to *free the memory* used by the old array!

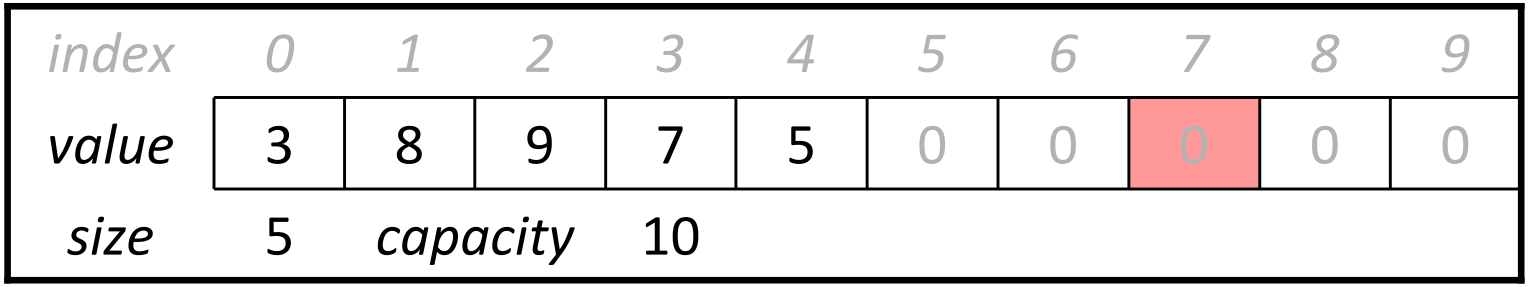
# Resize solution

```
// in ArrayList.cpp
void ArrayList::checkResize() {
    if (mySize == myCapacity) {
        // create bigger array and copy data over
        int* bigger = new int[2 * capacity]();
        for (int i = 0; i < myCapacity; i++) {
            bigger[i] = myElements[i];
        }
        delete[] myElements;
        myElements = bigger;
        myCapacity *= 2;
    }
}
```

# Problem: size vs. capacity

- What if the client accesses an element past the size?

```
list.get(7)
```




  - Currently the list allows this and returns 0.
    - Is this good or bad? What (if anything) should we do about it?

# Private helpers

```
// in ClassName.h file
private:
    returnType name(parameters);
```

- a **private member** function can be called only by its own class
  - your object can call the "helper" function, but clients cannot call it

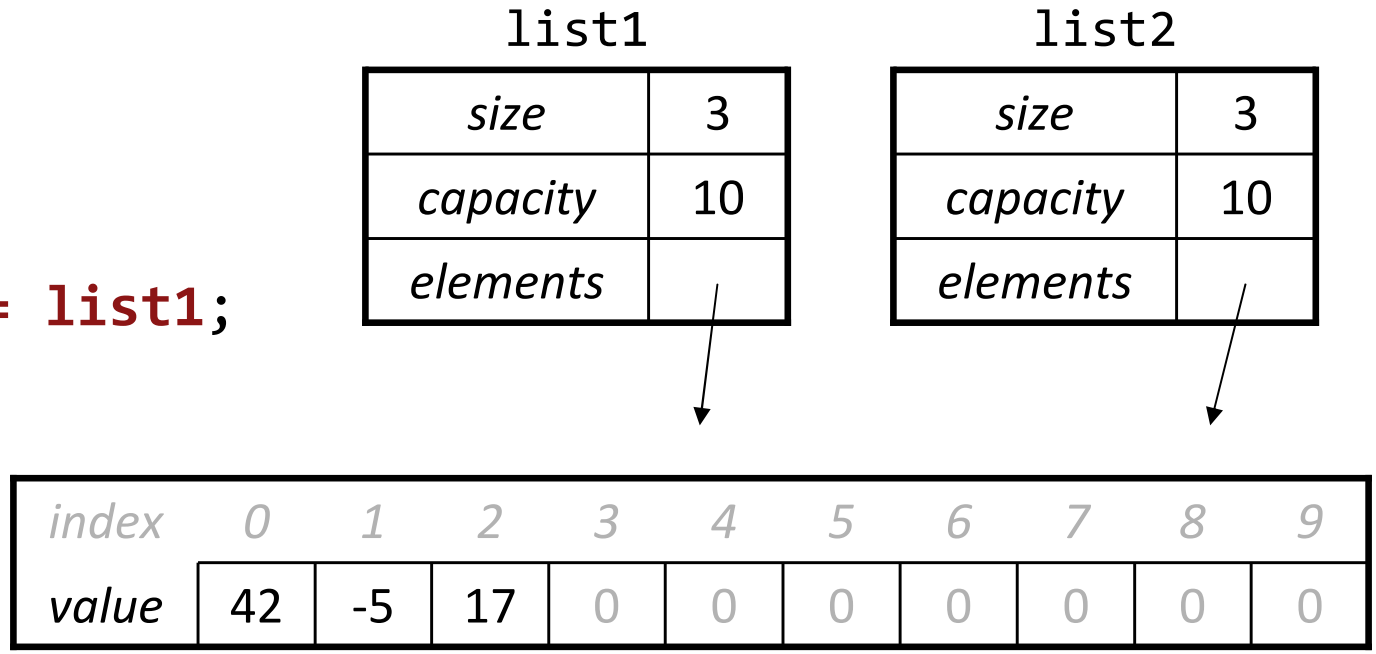
```
void ArrayList::checkIndex(int i, int min, int max) {
    if (i < min || i > max) {
        throw "Index out of range";
    }
}
```

# Overflow (extra) slides

# Shallow copy bug (12.7)

- Assigning one list to another causes them to share an array:

```
ArrayList list1;
list1.add(42);
list1.add(-5);
list1.add(17);
ArrayList list2 = list1;
```




- A change to one will affect the other. *(That's bad!)*

```
list2.add(88);
list.remove(0);
```

  - Also: When the objects are deleted, the memory is freed twice. *(bad)*

# Deep copy

- To correct the shallow copy bug, we must define:
  - a **copy constructor** (constructor that takes a list as a parameter)

```
ArrayList(const ArrayList& list);
```

  - an **assignment operator** (overloaded = op between two lists)

```
ArrayList& operator =(const ArrayList& list);
```

    - in both of these, we will make a *deep copy* of the array of elements.

- *Rule of Three*: In C++, when you define one of these three items in your class, you probably should define all three:

  1) copy constructor  2) assignment operator  3) destructor

# Forbid copying

- One quick fix is to just forbid your objects from being copied.
  - Declare a *private* copy constructor and = operator in the `.h` file.
  - Don't give them any actual definition/body in the `.cpp` file.

```
// ArrayList.h
private:
    ArrayList(const ArrayList& list);
    ArrayList& operator =(const ArrayList& list);
```

  - Now if the client tries `list2 = list1;` it will not compile.
  - Solves the shallow copy problem, but restrictive and less usable.

# Deep copy code

- To support deep copying, write a public copy constructor and = operator. They share a lot of code, so we suggest a helper:

```
// ArrayList.cpp
ArrayList::ArrayList(const ArrayList& other) {
    copy(other);
}

ArrayList& ArrayList::operator =(const ArrayList& other) {
    delete[] myElements;
    copy(other);
    return *this;          // = returns 'this' so it can be chained
}

void ArrayList::copy(const ArrayList& other) {  // private helper
    myCapacity = other.myCapacity;
    mySize = other.mySize;
    myElements = new int[myCapacity];           // deep copy
    for (int i = 0; i < mySize; i++) {
        myElements[i] = other.myElements[i];
    }
}
```

# Template function (14.1-2)

```
template<typename T>
returntype name(parameters) {
    statements;
}
```

- **Template**: A function or class that accepts a *type parameter(s)*.
  - Allows you to write a function that can accept many types of data.
  - Avoids redundancy when writing the same common operation on different types of data.
  - Templates can appear on a single function, or on an entire class.
  - FYI: Java has a similar mechanism called *generics*.

# Template func example

```
template<typename T>
T max(T a, T b) {
    if (a < b) { return b; }
    else       { return a; }
}
```

– The template is *instantiated* each time you use it with a new type.
  - The compiler actually generates a new version of the code each time.
  - The type you use must have an operator < to work in the above code.

```
int i    = max(17, 4);           // T = int
double d = max(3.1, 4.6);        // T = double
string s = max(string("hi"),     // T = string
               string("bye"));
```

# Template class (14.1-2)

- **Template class**: A class that accepts a type parameter(s).
  - In the header and cpp files, mark each class/function as templated.
  - Replace occurrences of the previous type `int` with T in the code.

```
// ClassName.h
template<typename T>
class ClassName {
    ...
};

// ClassName.cpp
template<typename T>
type ClassName::name(parameters) {
    ...
}
```

# Recall: ArrayList.h

```
class ArrayList {
public:
    ArrayList();
    ~ArrayList();
    void add(int value);
    void clear();
    int get(int index) const;
    void insert(int index, int value);
    bool isEmpty() const;
    void remove(int index);
    void set(int index, int value) const;
    int size() const;
    string toString() const;

private:
    int* elements;
    int mysize;
    int capacity;
    void checkIndex(int index, int min, int max) const;
    void checkResize();
};
```

# Template ArrayList.h

```
template <typename T> class ArrayList {
public:
    ArrayList();
    ~ArrayList();
    void add(T value);
    void clear();
    T get(int index) const;
    void insert(int index, T value);
    bool isEmpty() const;
    void remove(int index);
    void set(int index, T value) const;
    int size() const;
    string toString() const;

private:
    T* elements;
    int mysize;
    int capacity;
    void checkIndex(int index, int min, int max) const;
    void checkResize();
};
```

# Template ArrayList.cpp

```
template <typename T>
ArrayList<T>::ArrayList() {
    myCapacity = 10;
    myElements = new T[myCapacity];
    mySize = 0;
}

template <typename T>
void ArrayList<T>::add(T value) {
    checkResize();
    myElements[mySize] = value;
    mySize++;
}

template <typename T>
T ArrayList<T>::get(int index) const {
    checkIndex(index, 0, mySize - 1);
    return myElements[index];
}

...
```

# Template .h and .cpp

- Because of an odd quirk with C++ templates, the separation between .h header and .cpp implementation must be reduced.
  - Either write all the bodies in the .h file (suggested),
  - Or `#include` the .cpp at the end of .h file to join them together.

```
// ClassName.h
#ifndef _classname_h
#define _classname_h

template<typename T>
class ClassName {
    ...
};

#include "ClassName.cpp"
#endif   // _classname_h
```