CSCI 2270
Data Structures and Algorithms
Lecture 6

Elizabeth White
elizabeth.white@colorado.edu
Office hours: ECCS 128
Wed 1-2pm
Thurs 2-3pm

Dynamic memory arrays

Read http://www.cplusplus.com/doc/tutorial/dynamic/ for this week.

You should be reading Chapter 1 of CLRS (Intro to Algorithms) for this week. We'll begin talking about the speed of simple algorithms on arrays this week.

Dynamic memory arrays

```
int* int_array_maker(unsigned int size)
 {
      int* heap_array;
      heap_array = new int[size];
      return heap_array;
 }
```

Using the new command causes memory to be allocated from a different memory pool, called the heap. Heap variables don't get destroyed when they hit a closing bracket }.

Lots more space in the heap than in local memory (the stack).

Dynamic memory arrays

```
void int_array_maker_and_destroyer(unsigned int
size)
 {
      int* heap_array;
      heap_array = new int[size];
      delete [] heap_array;
 }
```

Heap variables have to be destroyed using the delete command. When the heap variable is an array, we add empty square brackets [] to the delete command.

Your first struct

```
struct int_array {
	static const unsigned int
DEFAULT_CAPACITY = 20;
	int* data;
	unsigned int count;
	unsigned int capacity;
};
```

This struct is called int_array.

It has 4 little variables, all stuck together.

(If you have seen classes before, this struct is an extremely primitive version of a class.)

Dynamic memory arrays

```
struct int_array {
	static const unsigned int
DEFAULT_CAPACITY = 20;
	int* data;
	unsigned int count;
	unsigned int capacity;
};
```

Array starts out as 20-slot array.

When you run out of room, you

- make a new array with double the capacity,
- copy your integers from the old array to the new array,
  delete the old array,
- and update the capacity variable.

Dynamic memory array variables

```
struct int_array {
      static const unsigned int
DEFAULT_CAPACITY = 20;
      int* data;
      unsigned int count;
      unsigned int capacity;
};
```

DEFAULT_CAPACITY: all arrays begin with capacity 20

data: array of integers we're storing

count: how many integers we've added to the array

capacity: current maximum number of integers we could store

Initializing the int_array

```
void init(int_array& arr)
```

1. Set capacity to the starting value DEFAULT_CAPACITY

```
arr.capacity = arr.DEFAULT_CAPACITY;
```

2. Create an array of integers for this array

```
arr.data = new int[arr.capacity];
```

3. Set count to 0

```
arr.count = 0;
```

```
struct int_array {
        int* data;
        unsigned int count;
        unsigned int capacity;
        static const unsigned int
DEFAULT_CAPACITY = 20;
        };
```

Destroying the int_array

```
void destr(int_array& arr)
```

1. Delete the array of integers

```
delete [] arr.data;
```

2. Set count to 0

```
arr.count = 0;
```

```
struct int_array {
        int* data;
        unsigned int count;
        unsigned int capacity;
        static const unsigned int
DEFAULT_CAPACITY = 20;
        };
```

Resizing the int_array

```
void resize(int_array& arr)
```

1. Update capacity

```
arr.capacity *= 2;
```

2. Create a bigger data array

```
int* bigger_array = new int[arr.capacity];
```

3. Copy data over

Use a loop; it's your call which one

4. Delete original data array

```
delete [] arr.data;
```

5. Set data array to bigger array

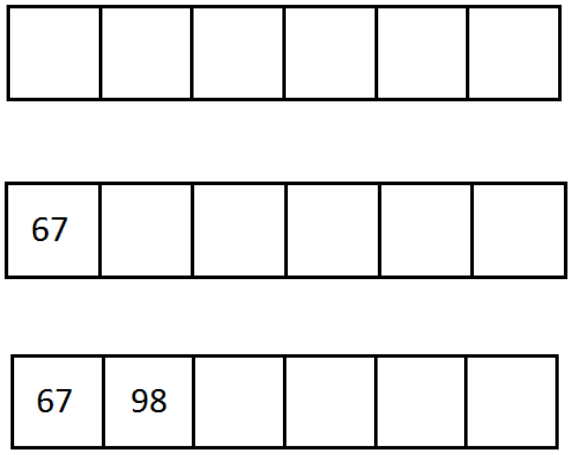
```
arr.data = bigger_array;
```

```
struct int_array {
        int* data;
        unsigned int count;
        unsigned int capacity;
        static const unsigned
int DEFAULT_CAPACITY = 20;
        };
```

Adding to the int_array

```
void add(int_array& arr, const int& payload)
```

1. Check if array is full and call resize if so
2. Add payload to next open slot in array
3. Update count

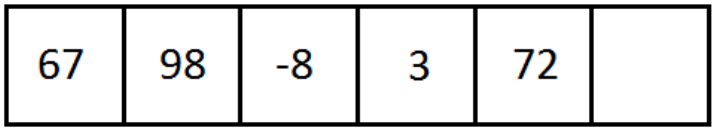


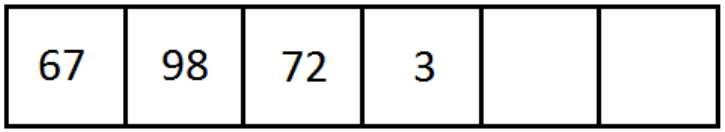

```
struct int_array {
        int* data;
        unsigned int count;
        unsigned int capacity;
        static const unsigned int
DEFAULT_CAPACITY = 20;
        };
```

Removing from the int_array

```
void remove(int_array& arr, const int& target)
```

1. Find index of target integer to remove, if it's there (return false if it's not).
2. If target was found in data array, move last integer stored in data array over the integer in target's slot.
3. Update count.






```
struct int_array {
        int* data;
        unsigned int count;
        unsigned int capacity;
        static const unsigned int
DEFAULT_CAPACITY = 20;
        };
```

Searching the int_array

```
void contains(int_array& arr, const int& target)
```

1. Find index of target integer, if it's in the array
2. If the target is present at least once, return true.
3. Otherwise, return false.

```
struct int_array {
        int* data;
        unsigned int count;
        unsigned int capacity;
        static const unsigned int
DEFAULT_CAPACITY = 20;
        };
```

Next week: make the array sorted

Need to change add and remove; nothing else

```
struct int_array {
        int* data;
        unsigned int count;
        unsigned int capacity;
        static const unsigned int
DEFAULT_CAPACITY = 20;
        };
```

Algorithms

From chapter 1 of CLRS

- Algorithms convert data inputs to answer outputs
- Algorithms are recipes for solving problems

Algorithms come with speed limits on performance

Example: lab this week, unsorted array of integers.

- adding an integer to the data is fast (unless we resize)
  - find data[count],
  - increase count (2 operations)
- removing an integer is slower, because we have to find it
  - this takes time proportional to the count

Algorithms

Example: lab next week, sorted array.

Speed of algorithm changes:

adding can be slower

removing is still ‘about the same’

contains can get much, much faster

This will give you your first example of an algorithm’s performance depending on the data structure it’s using.

Some problems, like the traveling salesman, have no good algorithmic solutions as they get larger.

(If you invent one, you’ll be famous or rich.)