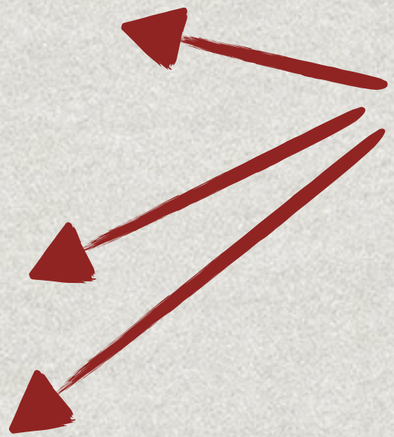
COMP 15 (Data Structures)
Lecture 2: Dynamic Arrays

Announcements

- Join Piazza! CODE: AVLROCK
- Lab 1 today (same as last week)
- Tomorrow is Monday
- Homework 1 out **tomorrow**!
- Office hours announced
- Unix tutorial this Thursday



Check Piazza!

Academic integrity

- Do not cheat!
- Do not cheat!
- Seriously….
  - Do not cheat!

Academic integrity

- Students started working on hw1
  - Is this fair?
  - Would I be happy with that?
- I am trusting you not to cheat
  - I can make assignments harder/less interesting
  - No one wants that

Let’s grade students

- 7 Assignments per students, drop the lowest
  - What data structure to use?
- 10 labs, drop the lowest
  - Do we need to rewrite code?

LET’S COMPARTMENTALIZE!

Abstract Data Types

- We want a structure to hold numbers
  - Don’t care what numbers represent
  - Don’t care about exact amount
  - Operation needed: average after dropping the lowest
  - Other useful features?

NOT SPECIFIC TO COMP15 AND GRADES!

Side bonus

We problem split into two

- **Implementer/Interface**
  - Make a structure to store grades
  - Add simple operations to manipulate
- **Client**
  - Given DS, insert student grades
  - Use operations to determine final score

Example interface

```
class Card {
public:
        Card();                         // defaults to ace of clubs
        Card(char r, char s);
        ~Card();

        void print_card();              // print card, e.g., "3D"
        void print_card_int();          // print integer value of card

        Suit get_suit();                // return suit of card
        void set_suit(Suit s);          // set the suit of card

        Rank get_rank();                // return rank of card (e.g., THREE)
        void set_rank(Rank r);          // set rank of card

        bool same_card(Card c);         // return true of represent same card
        int card_int();                 // return integer value of card
private:
        Suit suit;
        Rank rank;
};
```

bool same_card(Card c);

CLIENT/CUSTOMER SPECIFIES

```
bool same_card(Card c); // return true if the cards are the same
```

IMPLEMENTER WRITES

```
bool Card::same_card(Card c)
{
    return (rank == c.rank and suit == c.suit);
}
```

Exercise in pairs

- Exercise in pairs:
  - **Implementers**: you have array what operations can you give
  - **Client**: what operations do you need from implementer?

KEY POINT: CLIENT DOES NOT KNOW DETAILS!

Summary

- Abstract data types allow
  - Compartmentalize work
    - Split a difficult problem into many smaller ones
    - Great for outsourcing/working in teams
  - Reuse Code
  - …

Bottom line

Do not touch what is not yours!

```
bool has_card(Card c); // returns true if the card is in the list
```

```
bool List_dynamic_array::has_card(Card c)
{
    for (int i = 0; i < cards_held; i++) {
        if (cards[i].same_card(c))
            return true;
    }
    return false;
}
```

We give this in homework

YOU do this

Array

- One of the **simplest** data structures
  - Sequence of information arranged linearly
    - Scores of different students
    - Weight along time
    - Easily read/write with A[i]

Dynamic Array

- Like an array but size is **unknown**
  - How many students will I have?
  - How longer will I live? Need to track weight until I am 200
- Determined during execution
  - Simply start with a reasonable estimate
  - Keep track of how many are used (and free)

Val 1	Val 2	Val 3	Val 4	Val 5	**Rubbish**	**Rubbish**	**Rubbish**

How to manage?

- Structure often has three elements
  - Pointer to information
  - Integer maxElem (denoting maximum size)
  - Integer currElem (number of elements inserted)

Initially currElem=0

When inserting, place info in Array[currElem]

Increase currElem by one

What is array is **small**?

- Need to **expand** the array
- Five big steps
    1. Create a bigger array
    2. Transfer the info to new array
    3. Delete the old array
    4. Update the pointer to new array
    5. Make note of new size

What is array is **too big**?

- Need to **shrink** the array
- Five big steps

EXACTLY THE REVERSE OF EXPAND!

What operations do we want?

- Create array (estimated size)
- Insert element
- Search element
- Is empty
- …

- Expand
- Shrink

What operations do we want?

- Create array (estimated size)
- Insert element
- Search element
- Is empty
- …

- Expand
- Shrink

Client wants

Implementer needs

Public vs Private

```
class Card {
public:
        Card();                         // defaults to ace of clubs
        Card(char r, char s);
        ~Card();

        void print_card();              // print card, e.g., "3D"
        void print_card_int();          // print integer value of card

        Suit get_suit();                // return suit of card
        void set_suit(Suit s);          // set the suit of card

        Rank get_rank();                // return rank of card (e.g., THREE)
        void set_rank(Rank r);          // set rank of card

        bool same_card(Card c);         // return true of represent same card
        int card_int();                 // return integer value of card

private:
        Suit suit;
        Rank rank;
};
```

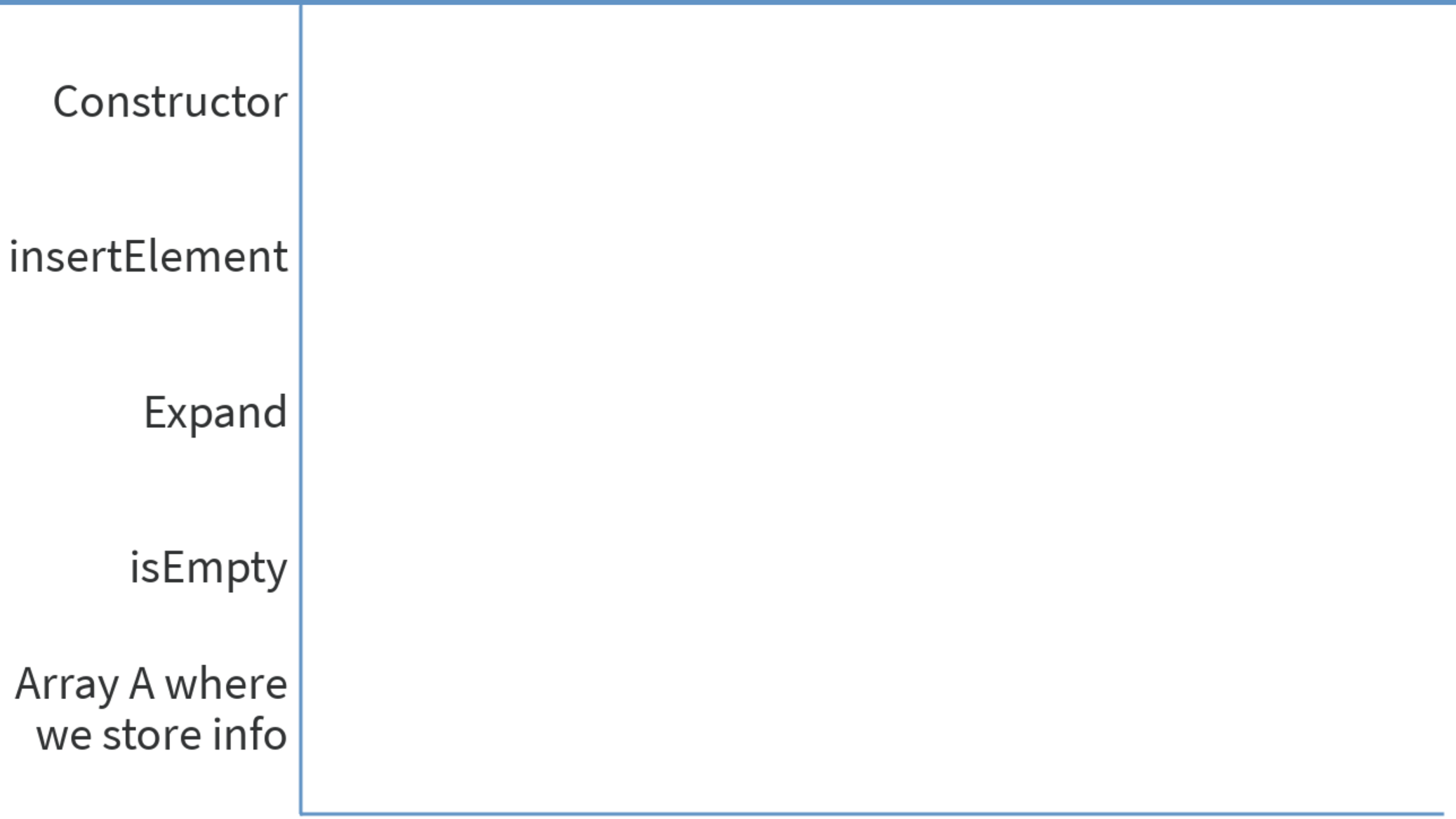
Public vs Private

- Two types of operations
  - Allow access to data
    - **Public**
  - Maintain invariants of data structure
    - **Private**

Exam question

Q1.5 What is the difference between public and private functions? Say that I decide to implement a new function within an existing class. How should I determine whether to make this function private or public?

Which of these functions should be private?



What about deletions?

- Find element in array
- Delete it

What about deletions?

- Find element in array
- Delete it
  - Need to **shift** everything to the left

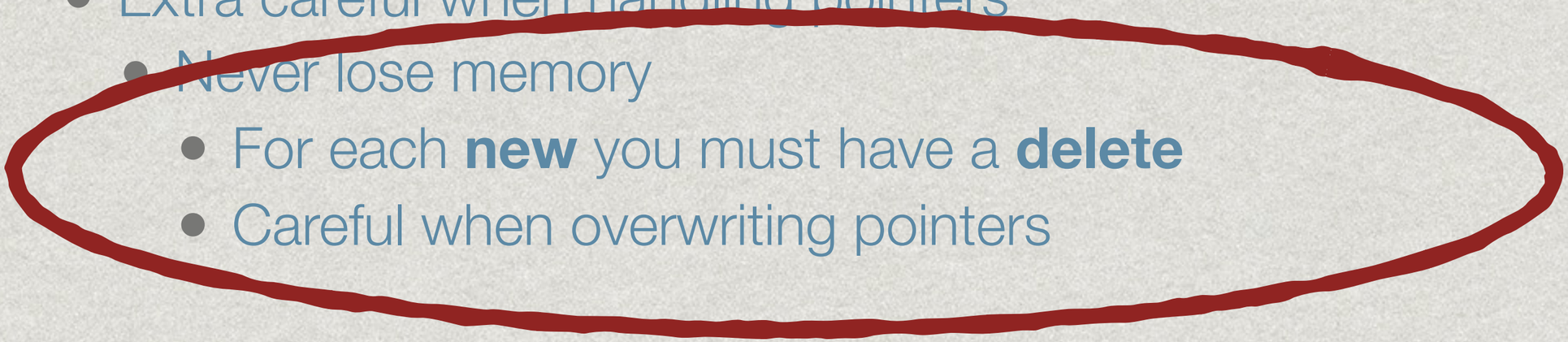
What about deletions?

- Find element in array
- Delete it
  - Need to **shift** everything to the left
  - Decrease numElem

No pointers damaged in the creation of this procedure

Let’s get hands dirty

- Extra careful when handling pointers
  - Never lose memory
    - For each **new** you must have a **delete**
    - Careful when overwriting pointers



Checking for leaks

We check your code with valgrind

- Big eye picture: Memory leak managing tool
- **`valgrind ./hw1`**
- Test ugly cases
  - Create lists of size 0
  - Alternate insertions, deletions, searches
  - Do you **always** shrink/expand?

Valgrind Sample execution

```
==27483==
==27483== HEAP SUMMARY:
==27483==     in use at exit: 88 bytes in 1 blocks
==27483==   total heap usage: 14 allocs, 13 frees, 1,959 bytes allocated
==27483==
==27483== LEAK SUMMARY:
==27483==    definitely lost: 88 bytes in 1 blocks
==27483==    indirectly lost: 0 bytes in 0 blocks
==27483==      possibly lost: 0 bytes in 0 blocks
==27483==    still reachable: 0 bytes in 0 blocks
==27483==         suppressed: 0 bytes in 0 blocks
==27483== Rerun with --leak-check=full to see details of leaked memory
==27483==
==27483== For counts of detected and suppressed errors, rerun with: -v
==27483== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 6 from 6)
```

Desired result

```
==32146==
==32146== HEAP SUMMARY:
==32146==     in use at exit: 0 bytes in 0 blocks
==32146==   total heap usage: 278 allocs, 278 frees, 41,184 bytes allocated
==32146==
==32146== All heap blocks were freed -- no leaks are possible
==32146==
==32146== For counts of detected and suppressed errors, rerun with: -v
==32146== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 6 from 6)
```

Not enough!