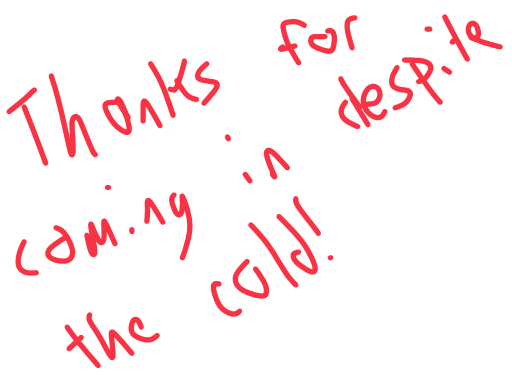
Data Structures

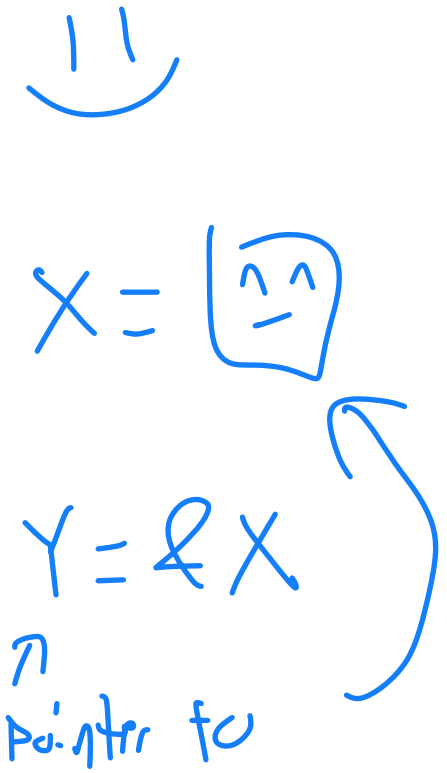
C++ Review

CS 225

Brad Solomon

January 23, 2026





UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Do you want to do research?. . .

. . . Are you a freshman or sophomore?

Come apply to **URSA**!

Undergraduate **R**esearch in **S**cientific **A**dvancement

Benefits:

- ☑ Research Experience
- ☑ Networking
- ☑ Soft and Hard Skill Development
- ☑ 1 credit hour + GPA boost
- ☑ Resume Booster

student run

apply for Spring

Scan for:

- Website
- Application
- Interest form

New Resources on Website

The Resources page on websites updates with lectures!

Inheritance

Pointers and References

Testing a 'Clicker' Set-up!

Have you signed up to take exam 0?

A) Yes! 92%

B) No! 8%

Join Code: 225

You can participate by going to website:

https://clicker.cs.illinois.edu/

Exam 0 (1/26 — 1/29)

An introduction to CBTF exam environment / expectations

Quiz on foundational knowledge from all pre-reqs

Practice questions can be found on PL

Topics covered can be found on website

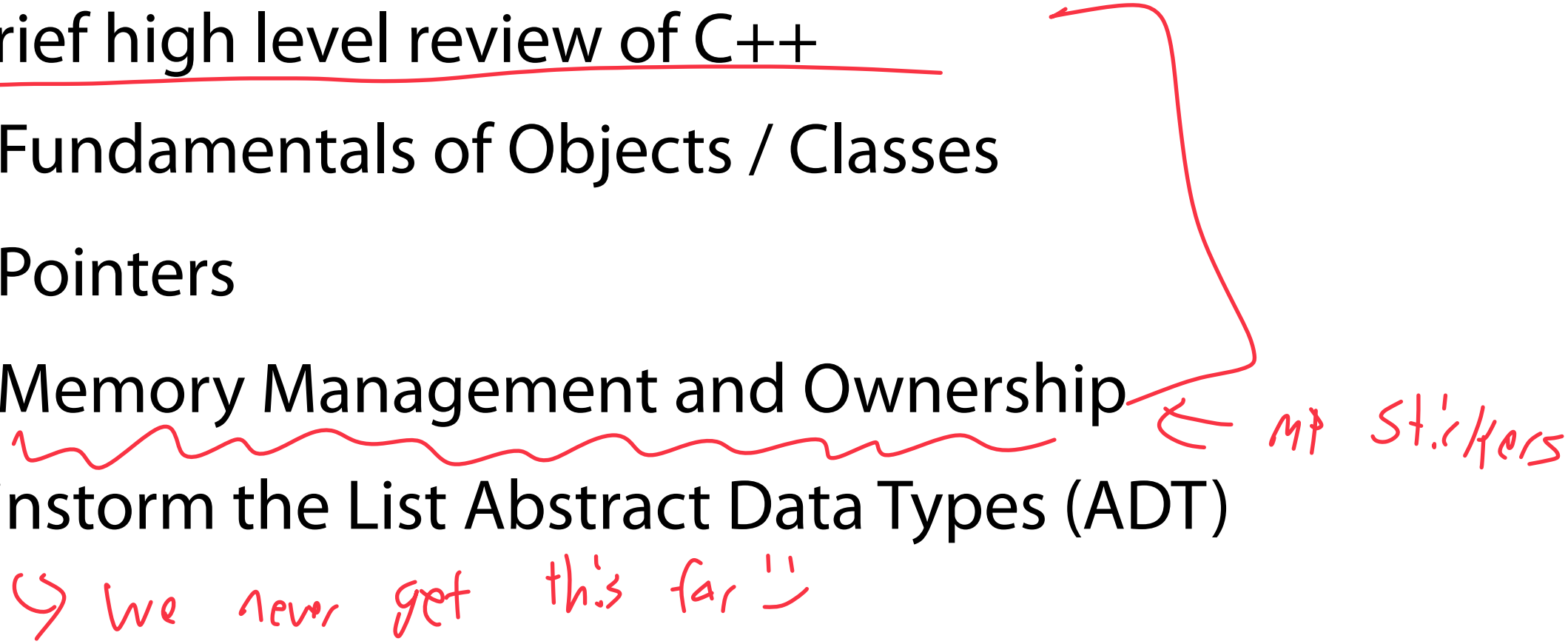
Registration open now :)

https://courses.engr.illinois.edu/cs225/exams/

Learning Objectives

A brief high level review of C++

- Fundamentals of Objects / Classes
- Pointers
- Memory Management and Ownership



Brainstorm the List Abstract Data Types (ADT)

↳ we never get this far :)

Encapsulation - Classes

Abstraction / organization separating:

Internal Implementation

↳ How this code does

↳ Left up to you!

Tip: Test each function 'Adp

External Interface

↳ What each function will do

↳ Doxygen (document)

↳ Input → output

tip: Always start here

↳ Draw it out!

Brainstorming a 'Library' class

```
1  class Library {
2  public:
3
4
5
6
7
8
9
10
11
12
13 private:
14
15
16
17
18
19
20
21 };
```

accessible outside the class

↳ titles of books and their count

↳ checkout Book

↳ Public contact information / hours

Only accessible internally

↳ Vector<Employee>

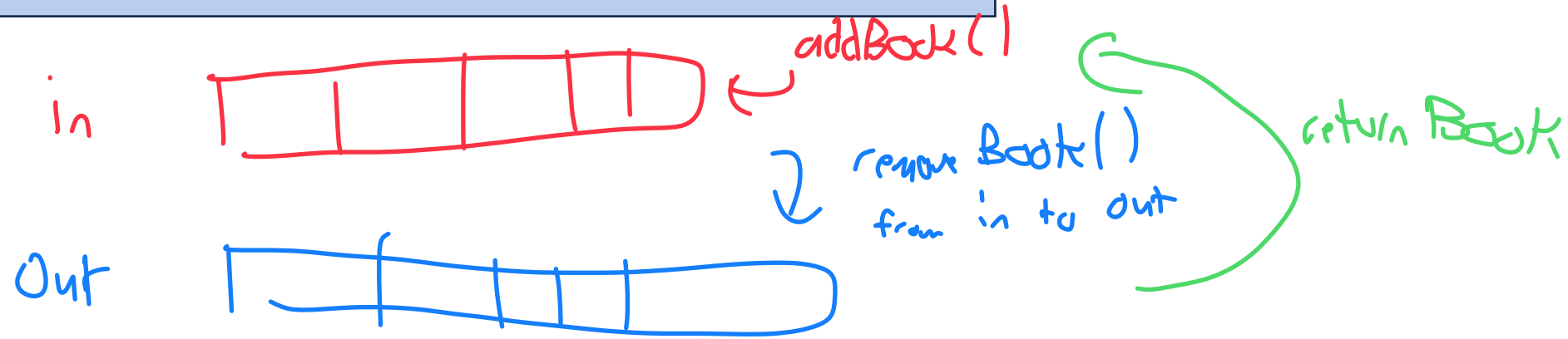
↳ bookOwner (who has a checked) out book

Interface

Memory Management — Ownership

Imagine I have a Library class (and hidden Book class):

```
 1 class Library{
 2 public:
 3     void addBook(Book * book);
 4     void removeBook(std::string title);
 5     void returnBook(Book * book);
 6
 7 private:
 8     std::vector<Book*> in;
 9     std::vector<Book*> out;
10 };
11
```



Memory Management — Ownership

Imagine I have a Library class:

```
 1 class Library{
 2 public:
 3     void addBook(Book * book);
 4     void removeBook(std::string title);
 5     void returnBook(Book * book);
 6
 7 private:
 8     std::vector<Book*> in;
 9     std::vector<Book*> out;
10 };
11
```

Join Code: 225

Pretest: Does Library class 'own' the Books it is storing?

A) **Yes!** 25%

B) **No!** 55%

C) **Not sure** 20%

Pointers

Pointers store memory addresses

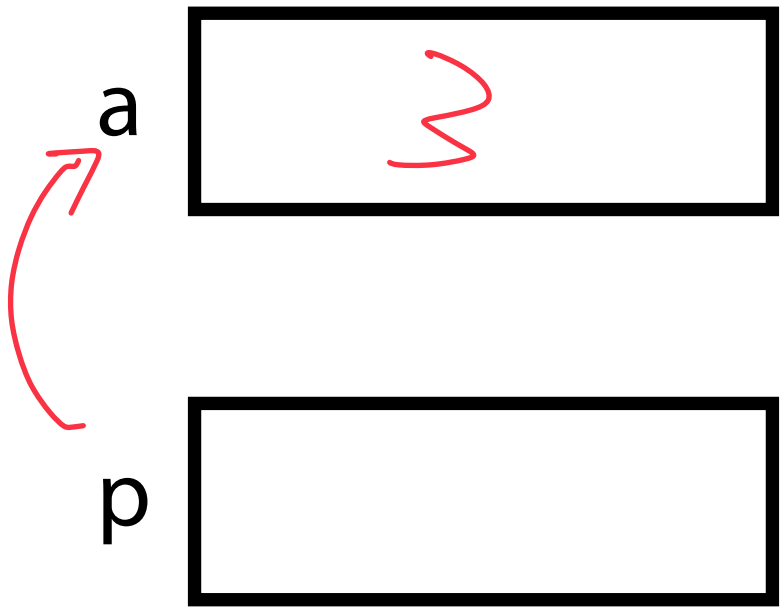
```
int a = 3;
```

local scope (stack)

```
int *p = &a;
```

address of

Also on the stack



Pointers

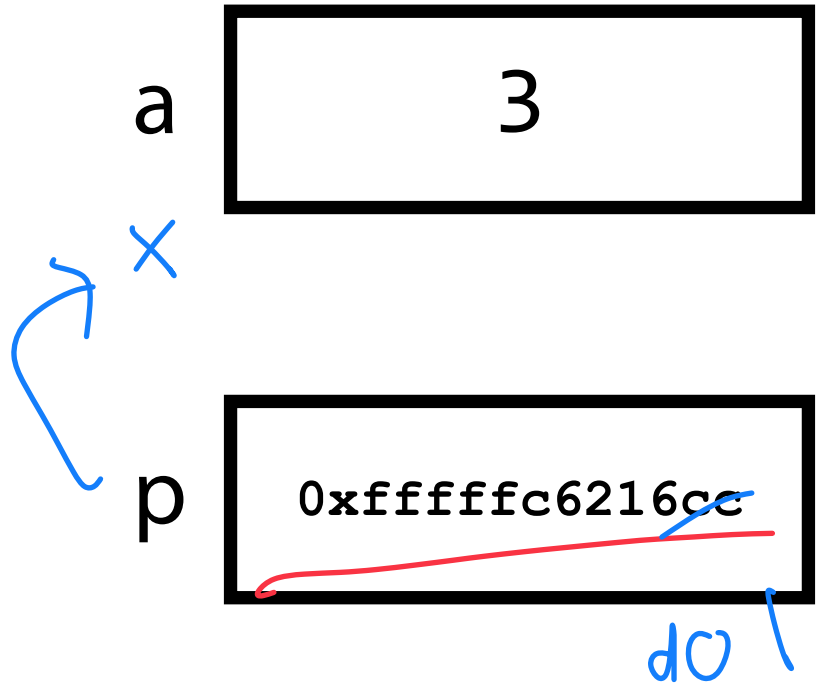
function:
a = 3
p = &a;
3 return p;

main:
p = function
*p++;
3

← This won't work!

Pointers store memory addresses

```
int a = 3;

int *p = &a;

p++;
```



Does a change? Does p?

↳ No

↳ Is being incremented by one

Unit of size(int)

Pointers

Pointers store memory addresses

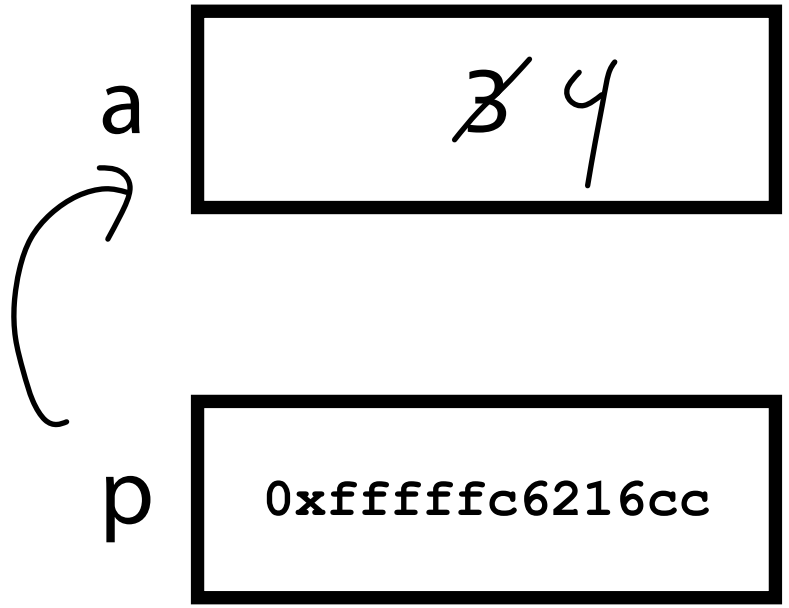
```
int a = 3;

int *p = &a;

(*p)++;
```

3++ → 4

dereference

Does a change? Does p?

↳ Yes! ↳ No!



Memory Management

Stack: Local variable storage

Ex: `int x = 5;`

exists in specific context

Heap: Dynamic storage

Ex: `int* x = new int[5];`

Anything I put on heap I must delete

Memory Management - Parameters

Pass by **Value:** A local copy of the original

Ex: addBook(Book book)

↳ changes to book don't persist

Pass by **Pointer to Value:** An address on the heap

Ex: addBook(Book* ~~book~~)

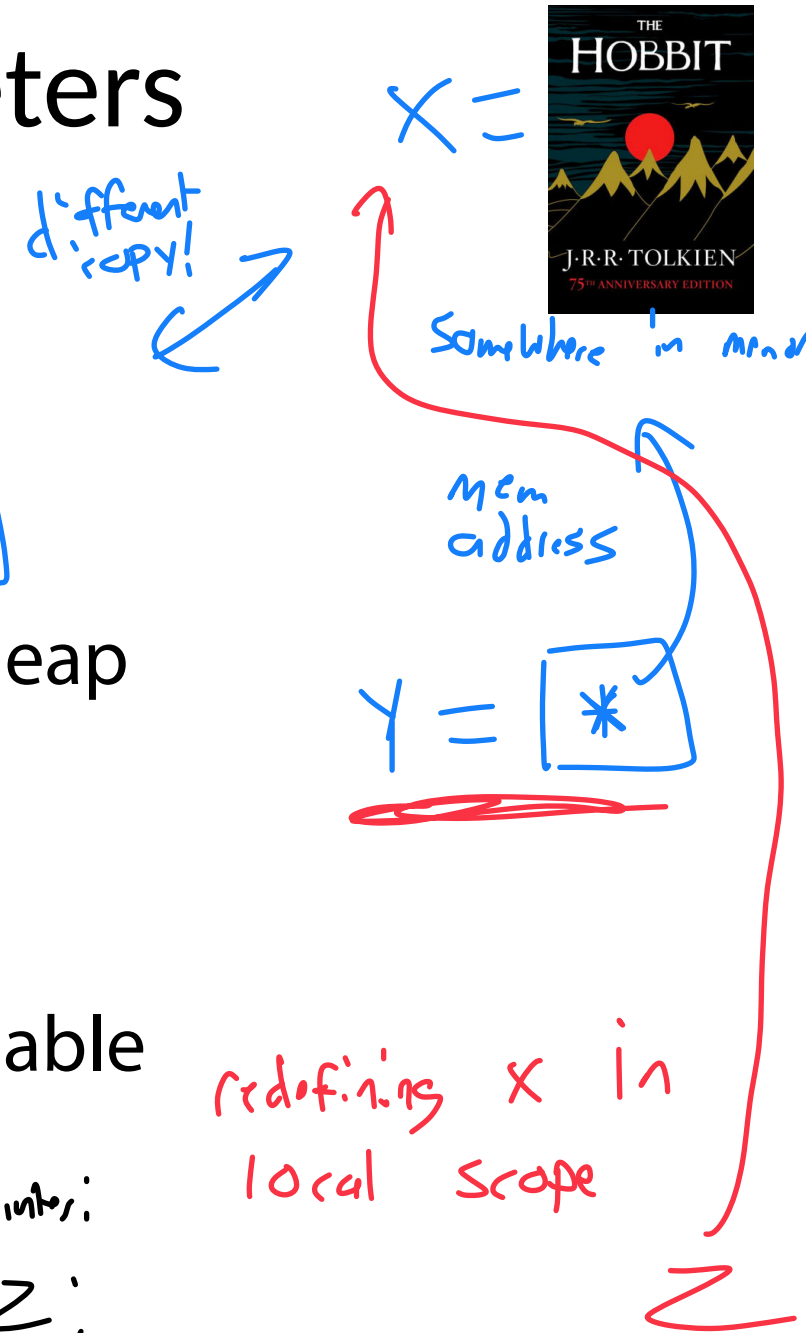
↳ changes here effect X

Pass by **Reference:** An *alias* to an existing variable

Ex: addBook(Book& ~~book~~)

If ref const to pointer:

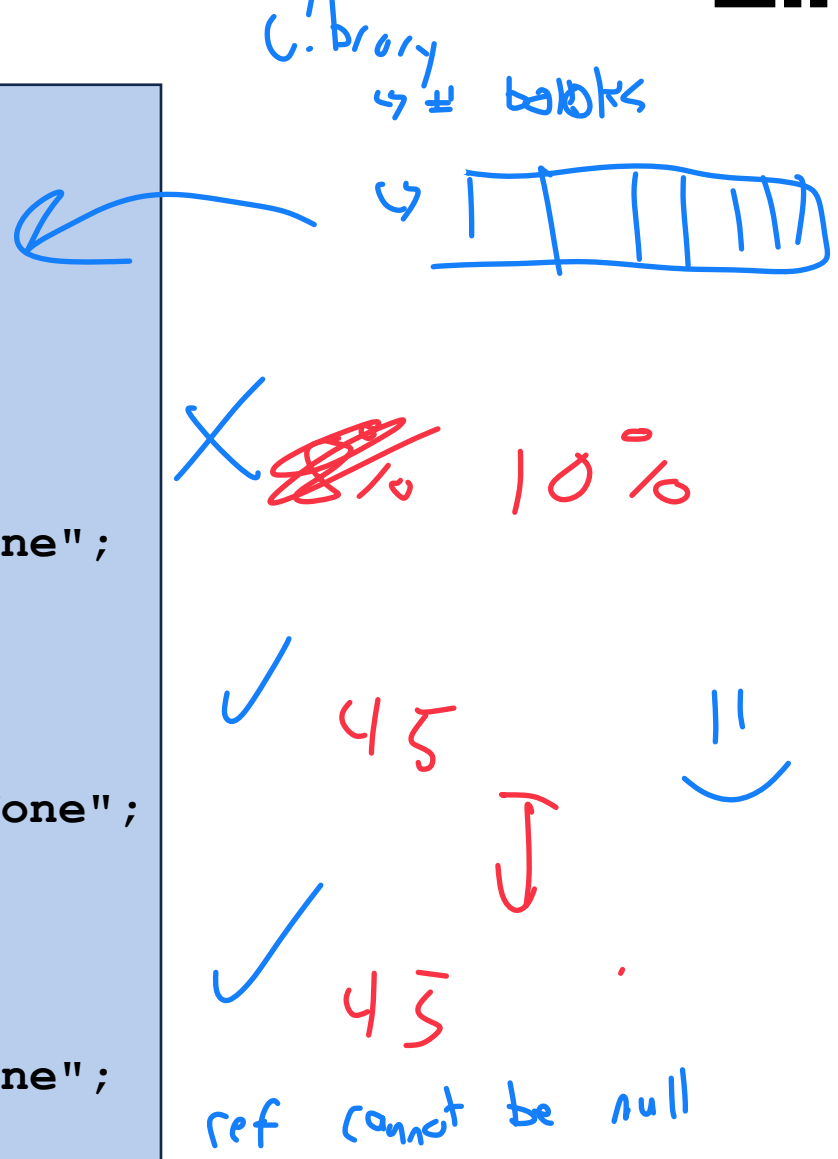
Book* p = &Z;



Memory Management - Parameters

Which implementation do you prefer?

```
1  class Library {
2  public:
3      int numBooks;
4      std::string * titles;
5  };
6
7
8  // *** Function A ***
9  std::string getFirstBook(Library l){
10     return (l.numBooks > 0) ? l.titles[0] : "None";
11 }
12
13
14 // *** Function B ***
15 std::string getFirstBook(Library * l){
16     return(l->numBooks > 0) ? l->titles[0] : "None";
17 }
18
19
20 // *** Function C ***
21 std::string getFirstBook(Library & l){
22     return (l.numBooks > 0) ? l.titles[0] : "None";
23 }
24
```



Memory Management

Local memory on the stack is managed by the computer

Heap memory allocated by **new** and freed by **delete**

Pass by value makes a copy of the object

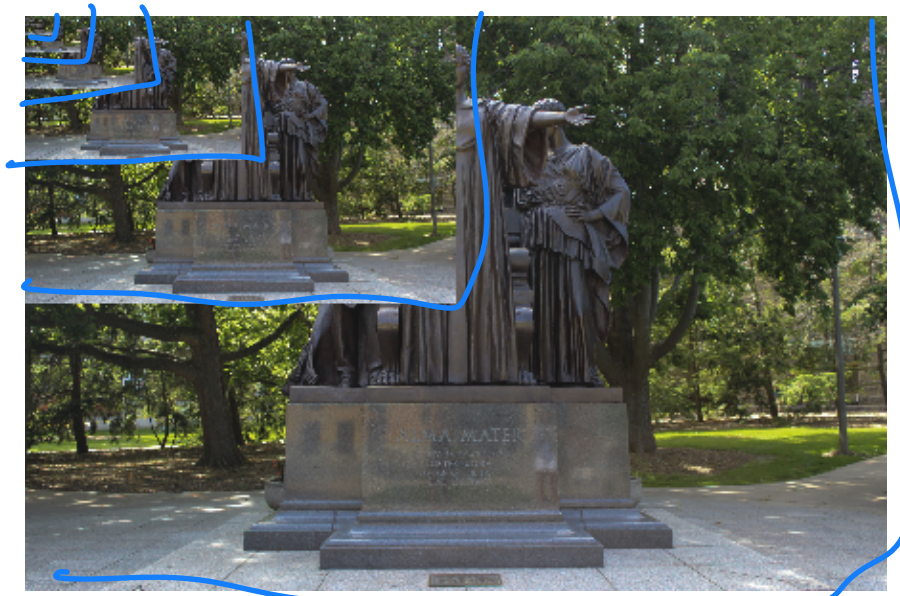
Pass by pointer can be dereferenced to modify an object

Pass by reference modifies the object directly

Memory Management — Ownership

What does **ownership** mean in C++?

↳ who allocates/ deallocates the item? memory



which fuss owns that image

PNG

Imagesp

Stickersheet

Memory Management — Ownership

```
1  class Library{
2  public:
3      void addBook(Book * book);
4
5
6      void removeBook(std::string title);
7
8
9      void returnBook(Book * book);
10 private:
11
12     std::vector<Book*> in;
13
14
15     std::vector<Book*> out;
16
17
18 };
```

Does Library 'own' Books?

A) **Yes!** ๒๐%

B) **No!** ๘๐%

C) **Not sure** ๐%

Memory Management — Ownership

```
1  class Library{
2  public:
3      void addBook(Book * book);
4
5
6      void removeBook(std::string title);
7
8
9      void returnBook(Book * book);
10 private:
11
12     std::vector<Book*> in;
13
14
15     std::vector<Book*> out;
16
17
18 };
```

Does Library 'own' Books?

A) **Yes!**

B) **No!**

C) **Not sure**

Are they destroyed when the Library destructor is called?

↳ Book * is pointer to Book which exists elsewhere

Memory Management — Ownership

```
1  class Library{
2  public:
3      void addBook(Book book);
4
5
6      void removeBook(std::string title);
7
8
9      void returnBook(Book book);
10 private:
11
12     std::vector<Book> in;
13
14
15     std::vector<Book> out;
16
17
18 };
```

Does Library 'own' Books?

A) **Yes!** 80%

B) **No!** ~20%

C) **Not sure** ~

Memory Management — Ownership

```
1  class Library{
2  public:
3      void addBook(Book book);
4
5
6      void removeBook(std::string title);
7
8
9      void returnBook(Book book);
10 private:
11
12     std::vector<Book> in;
13
14
15     std::vector<Book> out;
16
17
18 };
```

Does Library 'own' Books?

A) **Yes!**

B) **No!**

C) **Not sure**

Are they destroyed when the Library destructor is called?

Memory Management — Ownership

```
1  class Library{
2  public:
3      void addBook(const Book& book);
4
5
6      void removeBook(std::string title);
7
8
9      void returnBook(const Book& book);
10 private:
11
12     std::vector<Book*> in;
13
14
15     std::vector<Book*> out;
16
17
18 };
```

Does Library 'own' Books?

A) **Yes!**

B) **No!**

C) **Not sure**

Are they destroyed when the Library destructor is called?

Memory Management — Ownership

The owner of an object is responsible for its resource management (particularly allocation / deallocation)

A ‘litmus test’ of ownership — who handles destruction?

If we are storing pointers or references, not our problem!

Vector’s consolation prize — vector handles destruction

The Rule of Three

If it is necessary to **define any one** of these three functions in a class, it will be necessary to **define all three** of these functions:

1. Destructor — Called when we delete object

2. Copy Constructor — Make a new object as a copy of an existing one

3. Copy assignment operator — Assign value from existing X to Y

‘The Rule of Zero'

A corollary to Rule of Three

Classes that **declare** custom destructors, copy/move constructors or copy/move assignment operators should deal exclusively with ownership. Other classes **should not declare** custom destructors, copy/move constructors or copy/move assignment operators

— Scott Meyers

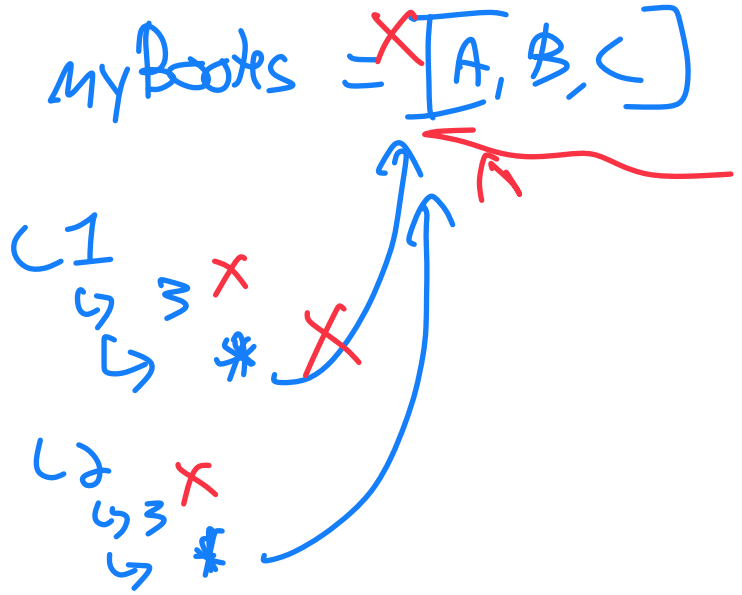
If I didn't allocate not my problem to deallocate

```
1  class Library {
2  public:
3      int numBooks;
4      std::string * titles;
5      ~Library();
6      Library( int num, std::string* list );
7  };
8
9  Library::~Library(){
10     delete titles;
11     titles = nullptr;
12 }
13
14 Library::Library(int num, std::string* list){
15     numBooks = inNum;
16     titles = new std::string[ inNum ];
17     std::copy(inList, inList + inNum, titles);
18 }
19
20 int main(){
21     std::string myBooks[3] = {"A", "B", "C"};
22     Library L1( 3, myBooks );
23     Library L2( L1 );
24     return 0;
25 }
```

dumb 'if'

destructor

You defined destructor
C++ compiler will auto create
a copy constructor
↳ shallow



```
 1 class Library {
 2 public:
 3     int numBooks;
 4     std::string * titles;
 5     ~Library();
 6     Library( int num, std::string* list );
 7 };
 8
 9 Library::~Library(){
10    delete titles;
11    titles = nullptr;
12 }
13
14 Library::Library(int num, std::string* list){
15     numBooks = inNum;
16     titles = new std::string[ inNum ];
17     std::copy(inList, inList + inNum, titles);
18 }
19
20 int main(){
21    std::string myBooks[3] = {"A", "B", "C"};
22    Library L1( 3, myBooks );
23    Library L2( L1 );
24    return 0;
25 }
```

Whats wrong with this code?

A. Can't create L2 Library obj

B. Don't delete either Library

C. The second object being deleted crashes

I said the wrong thing to student question
↳ there is a new

So the better fix is make a deep copy

Questions?

Templates

A way to write generic code whose type is determined during completion

Templates

A way to write generic code whose type is determined during completion

1. Templates are a recipe for code using generic types

Templates

A way to write generic code whose type is determined during completion

1. Templates are a recipe for code using generic types

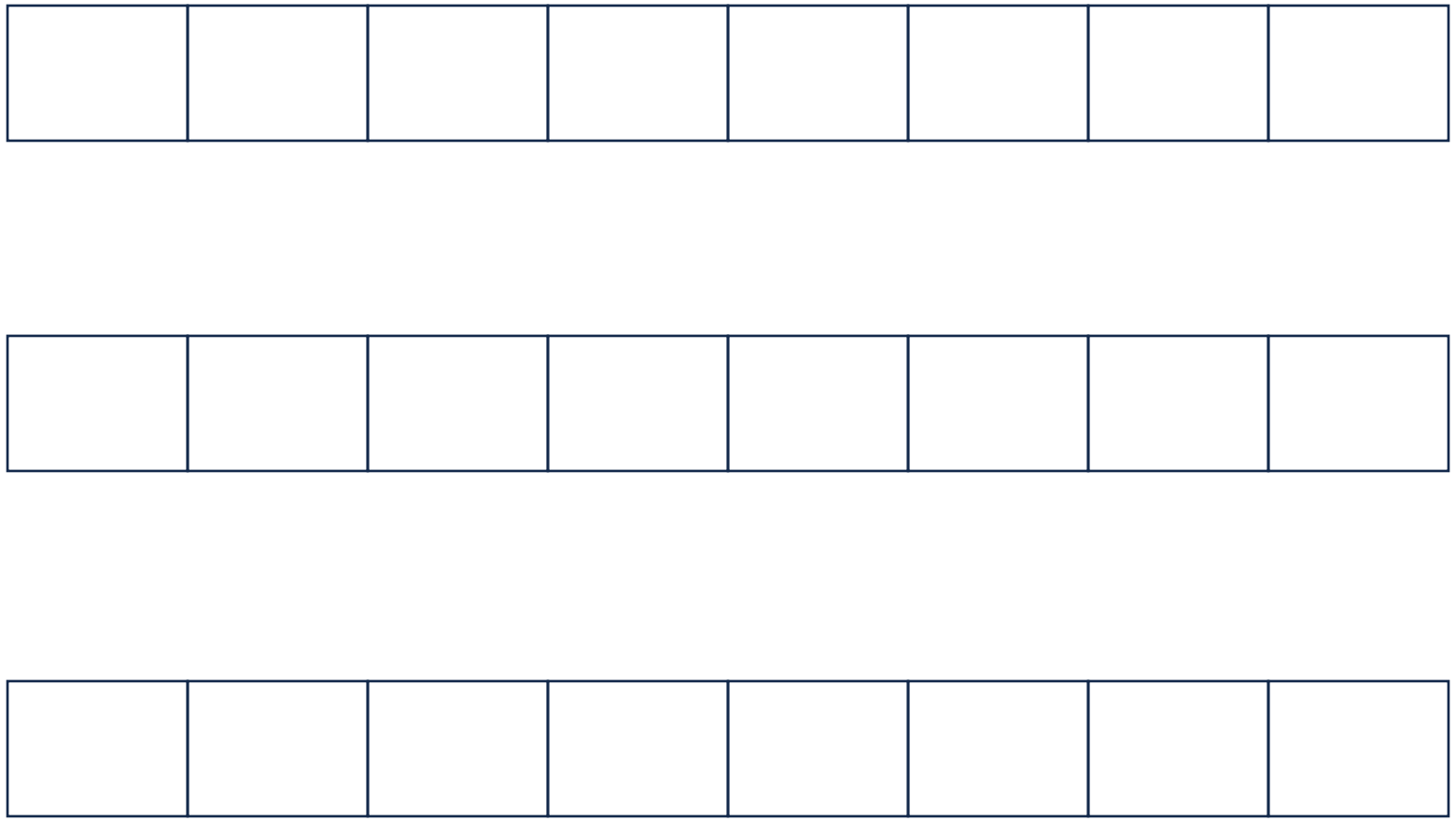
2. The compiler uses templates to generate C++ code **when needed**

```
template <typename T>
T sum(T a, T b){
...
}
```

template1.cpp

```
1  template <typename T>
2  T max(T a, T b) {
3    T result;
4    result = (a > b) ? a : b;
5    return result;
6  }
7
```

Templates are very useful!



List Abstract Data Type

What is the expected **interface** for a list?