

Data Structures

C++ Review

CS 225

Brad Solomon

August 28, 2023



UNIVERSITY OF
ILLINOIS
URBANA - CHAMPAIGN

Department of Computer Science



Theta Tau Fall Rush 2024

Freshly Theta Tau!



09/03 5:30-7 PM
INFO SESSION #1
CIF 3039



MIXER
09/05
5:30-7 PM
LOOMIS 151



COOKOUT
09/07 12:30-2:30 PM
Illini Grove



**INFO SESSION #2 &
Interview Workshop**
09/09 5-7 PM
David Kinley 114



JEOPARDY
09/12 5:30-7 PM • EVERITT 1306



HAPPY HOUR
09/13 5-6:30 PM
**ENGINEERING
QUAD**

.....09/14 & 09/15 10 AM-1 PM: INTERVIEWS (ONLINE).....

Companies our
consultants
have worked
with

Bloomberg



OTCR CONSULTING

UIUC's premier student-run consulting group

 [@otcr_consulting](https://www.instagram.com/otcr_consulting)

 [OTCR Consulting](https://www.facebook.com/OTCRConsulting)

 <https://www.otcrconsulting.com>

Quad Day

August 25th 12PM-4PM
Main Quad

Business Quad Day

August 27th 3:30PM-6PM
South Quad

Info Nights

August 29th 6PM-7PM
Location TBA

September 3rd 7PM-8PM
Location TBA

Meet & Greet and Case Training

September 4th 7PM-9PM
Location TBA

Please dress business casual for Info
Nights, Meet & Greet, and Case Training

Application Deadlines

1. August 30th @ 11:59pm
2. September 6th @ 11:59pm

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



Scan for:

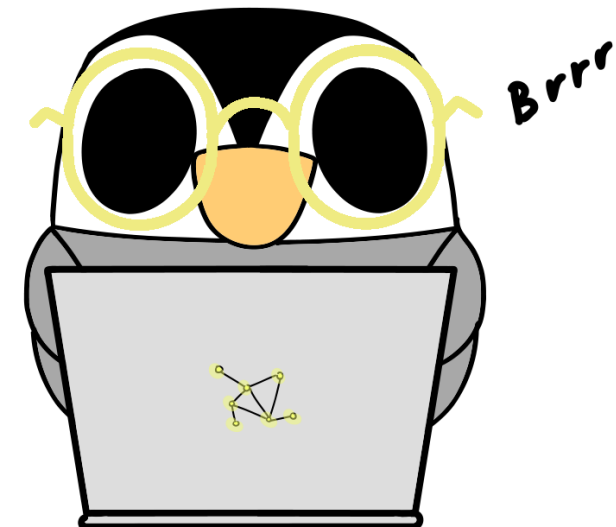
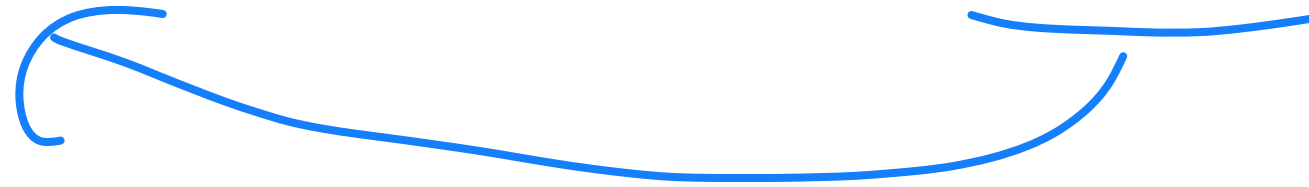
- Website
- Application
- Interest form

(Optional) Open Lab This Week

This week's lab is open office hours

Focus is making sure your machine is setup for semester

Installation information available on website



Office Hours

The office hour calendar will be populated next week

For now, please use Discord or Piazza

You can also stop by my regular office hours!

Thursday, 11 AM — 12 PM

Siebel 2233



Testing a 'Clicker' Set-up!

Have you signed up to take exam 0?

A) Yes!

B) No!

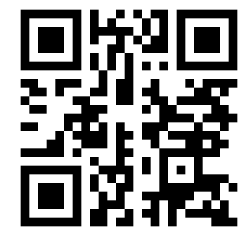


Join Code: 225

You can participate by going to website:

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

Exam 0 (9/4 — 9/6)



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 started August 22

<https://courses.engr.illinois.edu/cs225/fa2024/exams/>

Learning Objectives

A brief high level review of C++

Fundamentals of Objects / Classes

Memory Management and Ownership

Pointers and Const

Brainstorm the List Abstract Data Types (ADT)

Encapsulation - Classes

Abstraction / organization separating:

Internal Implementation

↳ How

↳ Your job

External Interface

↳ What

↳ Doxygen



Brainstorming a 'Library' class

Interface → Data
→ Functions

```
1 class Library {  
2 public:  
3  
4     ↳ Books in library  
5     ↳ add Books () / remove Books ()  
6  
7  
8  
9  
10  
11  
12  
13 private:  
14  
15     ↳ Library Cards  
16  
17     ↳ Books Overdue  
18     ↳ Books on Hold  
19  
20  
21 };
```

Implementation → How

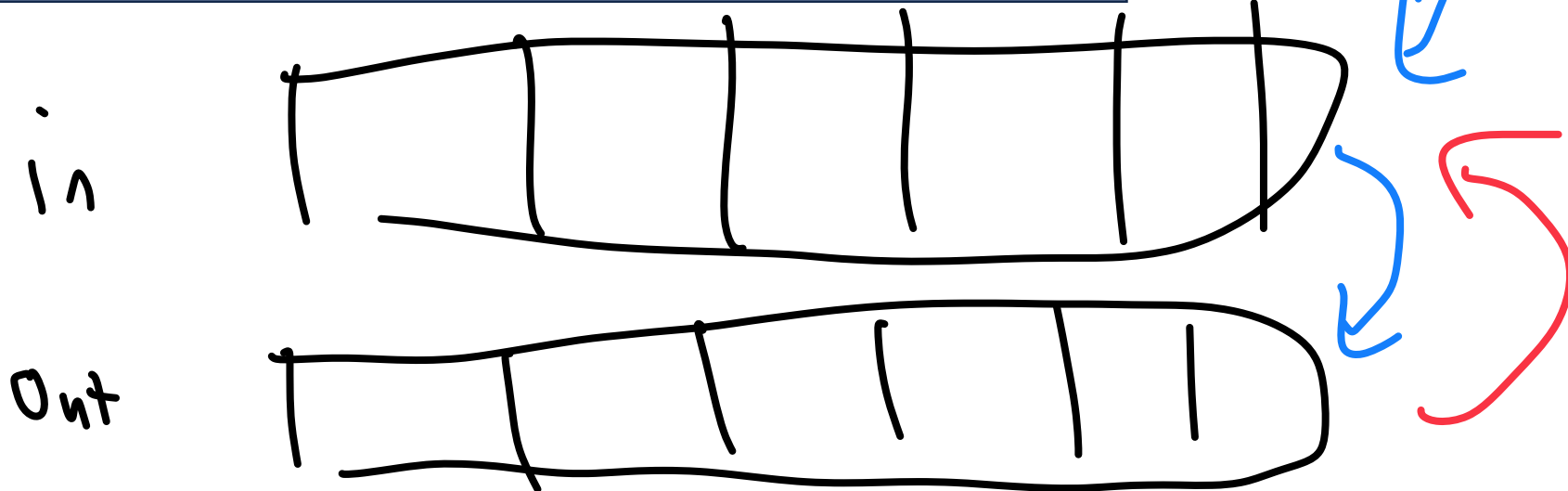
} who has what book

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
6 private:
7     std::vector<Book> in;
8     std::vector<Book> out;
9 };
```

return Book()



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
6 private:
7     std::vector<Book> in;
8     std::vector<Book> out;
9 };
```



Join Code: 225

Does my Library class 'own' the Books it is storing?

A) Yes!

B) No!

C) Not sure

Memory Management

Stack: Local variable storage

exists only in function
or
scope
defined

Ex: `int x = 5;`

Memory handled by computer

Heap: Dynamic storage

persist until deleted

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

☹ I Have to delete it!

Memory Management - Parameters

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

Ex: addBook(Book book)

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

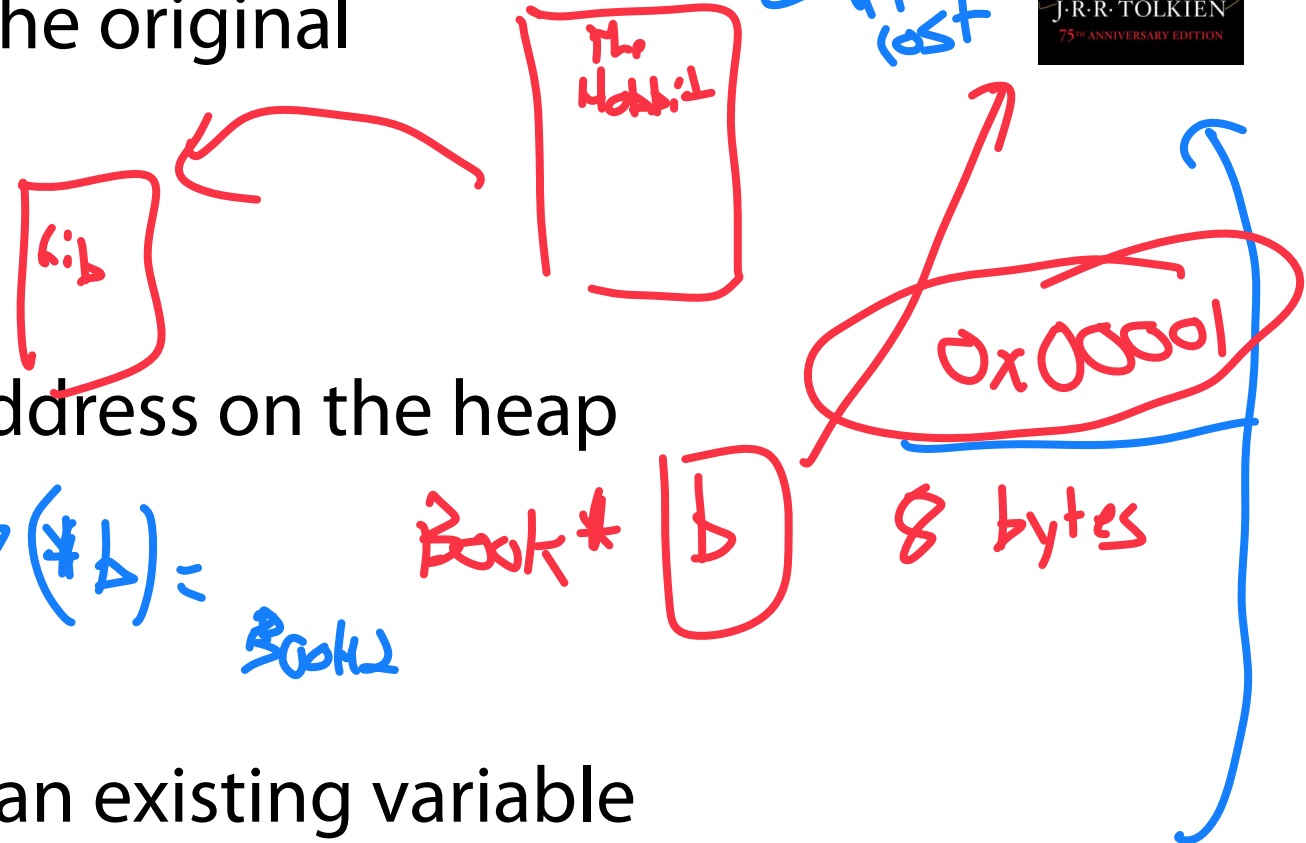
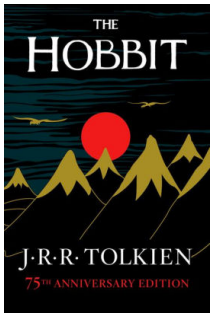
Ex: addBook(Book* book)

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

Ex: addBook(Book& book)

↳ (*b) = book
↳ const pointer

2+
data cost



Book & c = Hobbit

Memory Management - Parameters

Which implementation is 'best'? Why?



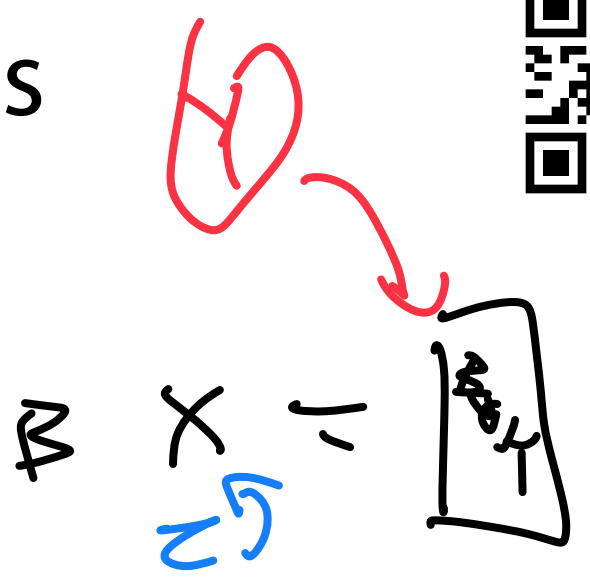
```
1 class Library {
2 public:
3     int numBooks;
4     std::string * titles;
5 };
6
7
8 // *** Function A ***
9 std::string getFirstBook(Library l){
10     return (l.numBooks > 0) ? l.titles[0] : "None";
11 }
12
13
14 // *** Function B ***
15 std::string getFirstBook(Library * l){
16     return (l->numBooks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

value

ptr

const

ref



~~X~~ Y = &X
Z = X

delete Y

Memory Management



Local memory on the stack is managed by the computer

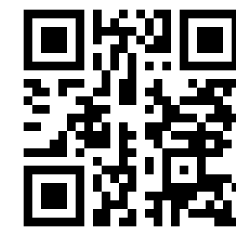
Heap memory allocated by **new** and freed by **delete**

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



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

Does Library 'own' Books?

A) **Yes!** 45%

B) No! 50%

C) **Not sure** 4%

Does my destructor need to delete them?

↳ No I don't have to delete

↳ Library has copy of every book it owns these books

Memory Management — Ownership

```
1 class Library{
2 public:
3
4     void addBook(Book book);
5
6
7     void removeBook(std::string title);
8
9
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!**

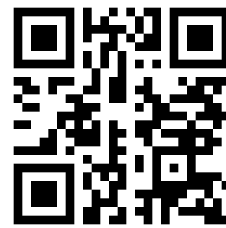
C) **Not sure**

Own books?

1) I must delete → No
[Memory management]

2) Library has storage (costs equal to Books) destructor removes → Yes

Memory Management — Ownership



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

Does Library 'own' Books?

A) Yes!

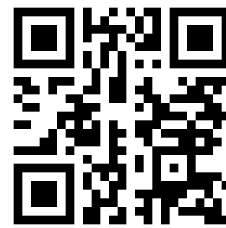
B) No!

C) Not sure

Does my destructor need to delete them?

↳ Yes, delete manually

Memory Management — Ownership



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

Does Library 'own' Books?

A) **Yes!**

B) No!

C) **Not sure**

Does my destructor need to delete them?

↳ No need to delete!

exists!

by ref this book exists

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

Memory Management — Ownership



If I don't have to allocate things, I should not allocate them!

Try to always use an existing class that handles ownership!

↳ Vector

Before you use keyword 'new', try everything else.

```
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 }
```

Variables

Destructor

Constructor

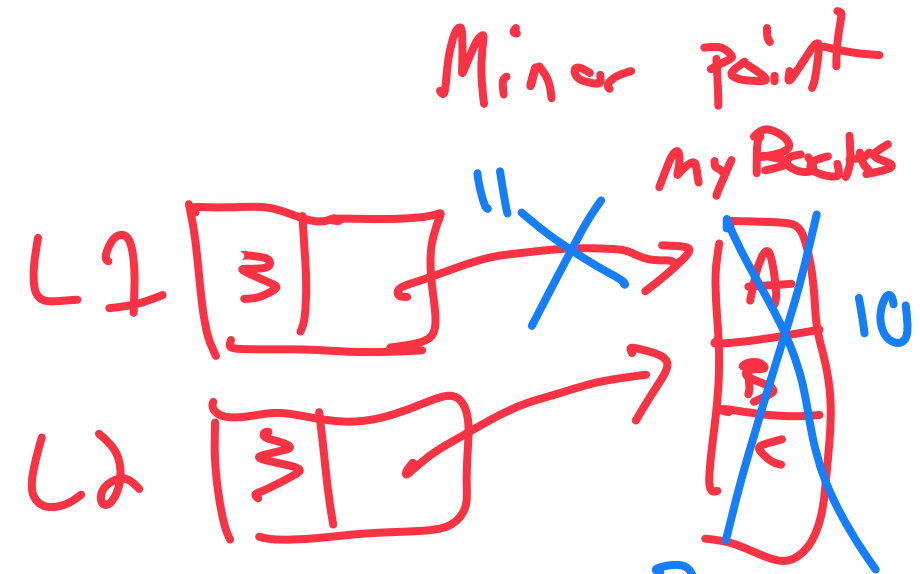
```

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. Deleting L1 deletes L2



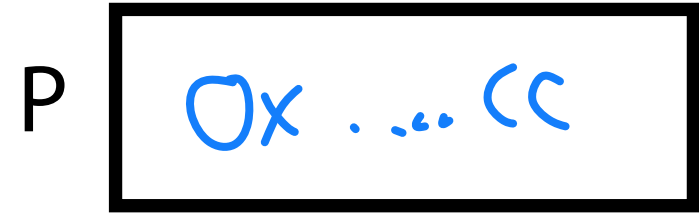
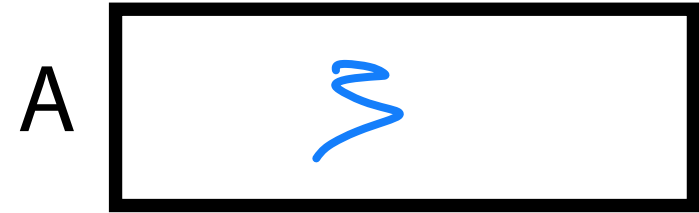
This is dangerous!
Avoid when possible



Pointers

```
1 int a = 3;  
2  
3 int *p = &a; // Value: 0xfffffc6216cc
```

↖ add res of

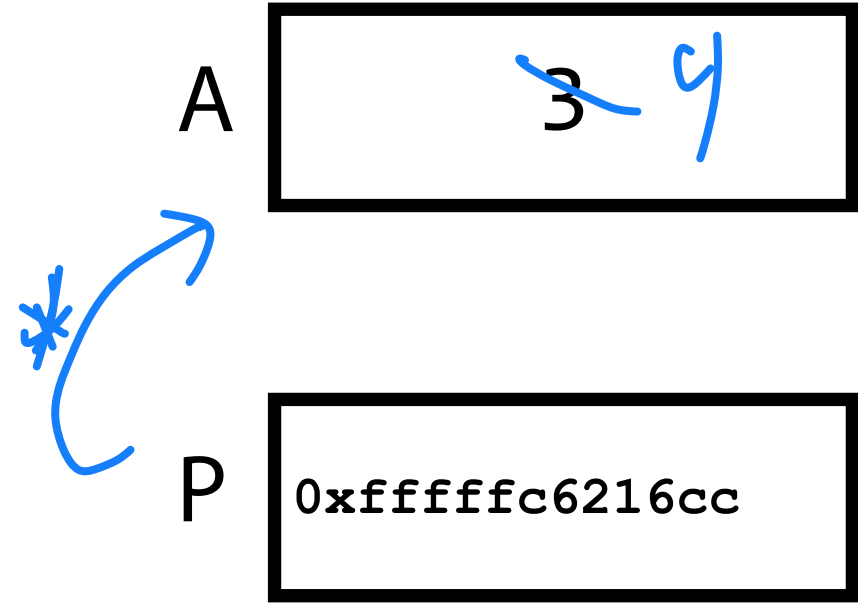


↪ value string address

Pointers

```
1 int a = 3;  
2  
3 int *p = &a; // Value: 0xfffffc6216cc  
4  
5 (*p)++;  
6  
7  
8  
9
```

dereference



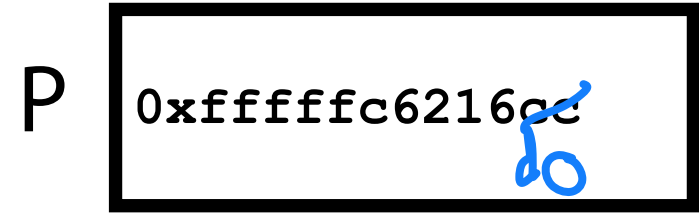
Pointers

```
1 int a = 3;  
2  
3 int *p = &a; // Value: 0xfffffc6216cc  
4  
5 (*p)++;  
6  
7 p++;  
8  
9
```

4 bytes

0x ~ ~ ~ do

↓

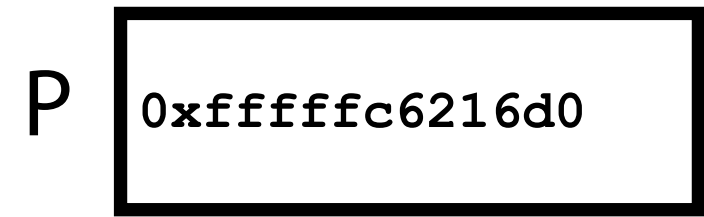
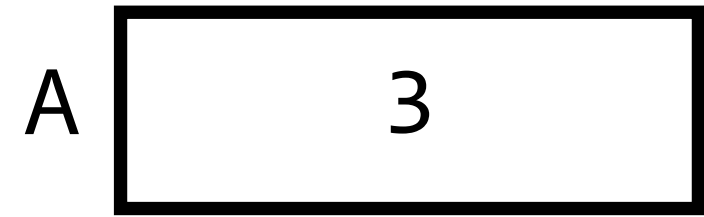


Pointers



```
1 int a = 3;
2
3 int *p = &a; // Value: 0xfffffc6216cc
4
5 (*p)++;
6
7 p++;
8
9 int *b;
```

modi 77
↓



←

(*b) = 5; (ash!)



Pointer-to-const vs constant pointer

```
1 int x = 3;
2 int y = 2;
3 // *** A ***
4 const int* a = &x;
5
6 a = &y;
7
8 // *** B ***
9 const int* b = &x;
10
11 *b = y;
12
13 // *** C ***
14 int* const c = &y;
15
16 c = &x;
17
18 // *** D ***
19 int* const d = &y;
20
21 *d = x;
```

`(const int)* a = &x;`

`(int)* const c = &y;`



Const pointers vs const methods

```
1 struct BlackBox {  
2     void update(const int & obj) {  
3         myVal = obj;  
4     A obj++;  
5     }  
6  
7  
8  
9     void update(int & obj) const {  
10        myVal = obj;  
11        B obj++;  
12    }  
13  
14  
15  
16     void update(const int & obj) const {  
17        myVal = obj;  
18        C obj++;  
19    }  
20  
21  
22  
23     int myVal;  
24 };  
25
```

Can't change value of obj;

Set values to this

Can't change member variables

The Const Keyword



Const means that an object cannot be modified

Variables: Can't change value

Pointers: Cant change value OR can't change pointer

Reference: Can't change value (address always fixed)

Method: Prevents non-mutable members from changing

Templates

Didn't get to!

will go over on Friday



template1.cpp



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

Templates in the context of Lists

--	--	--	--	--	--	--	--

--	--	--	--	--	--	--	--

--	--	--	--	--	--	--	--

List Abstract Data Type

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