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 fall

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!

B) No!

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)

Encapsulation - Classes

Abstraction / organization separating:

Internal Implementation **External Interface**

Brainstorming a ‘Library’ class

```
class Library {
public:










private:







};
```

Memory Management — Ownership

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

```
class Library{
public:
    void addBook(Book * book);
    void removeBook(std::string title);
    void returnBook(Book * book);

private:
    std::vector<Book*> in;
    std::vector<Book*> out;
};

```

Memory Management — Ownership

Join Code: 225

Imagine I have a Library class:

```
class Library{
public:
    void addBook(Book * book);
    void removeBook(std::string title);
    void returnBook(Book * book);

private:
    std::vector<Book*> in;
    std::vector<Book*> out;
};

```

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

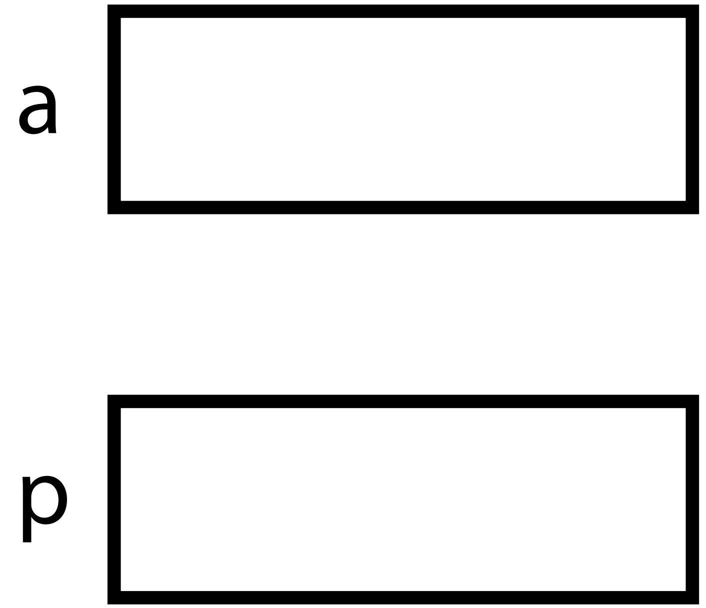
A) **Yes!**   B) **No!**   C) **Not sure**

Pointers

Pointers store memory addresses

```
int a = 3;

int *p = &a;
```



Pointers

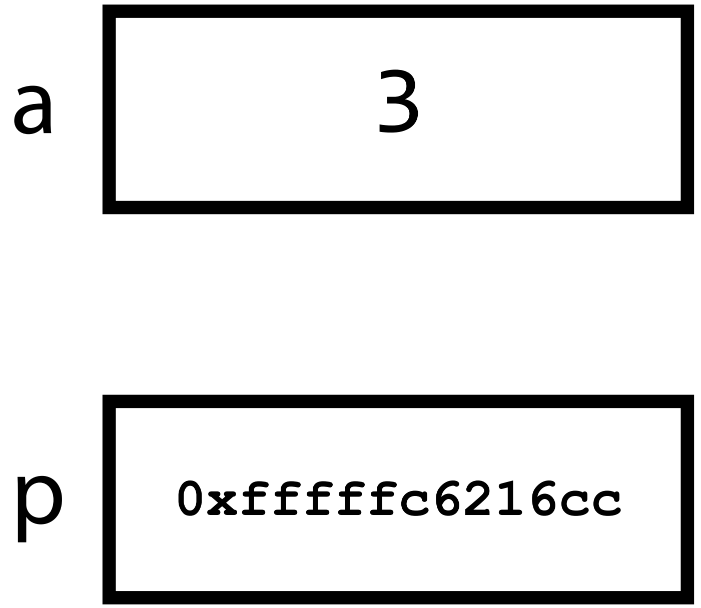
Pointers store memory addresses

```
int a = 3;

int *p = &a;

p++;
```

Does a change? Does p?



Pointers

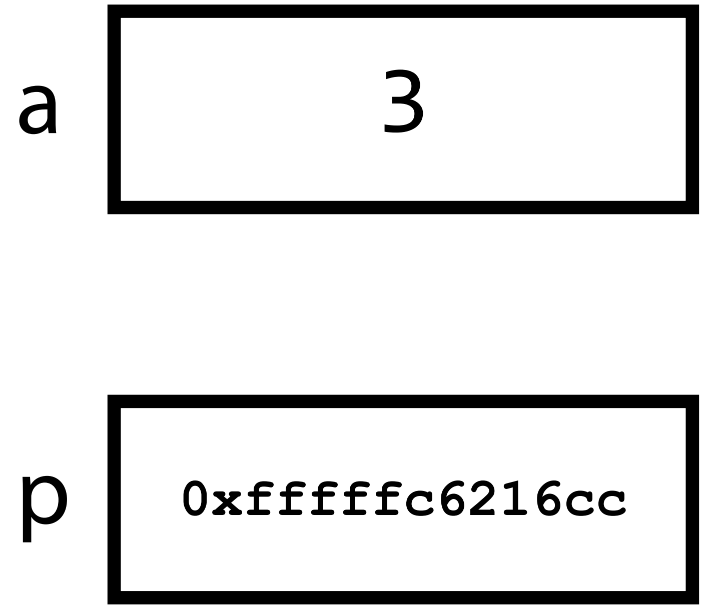
Pointers store memory addresses

```
int a = 3;

int *p = &a;

(*p)++;
```

Does a change? Does p?



Memory Management

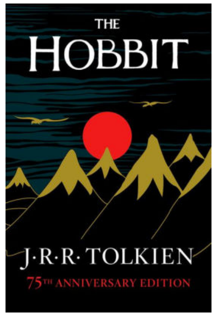
Stack: Local variable storage

Ex: `int x = 5;`

Heap: Dynamic storage

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

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)

Memory Management - Parameters

Which implementation do you prefer?

```
class Library {
public:
    int numBooks;
    std::string * titles;
};

// *** Function A ***
std::string getFirstBook(Library l){
    return (l.numBooks > 0) ? l.titles[0] : "None";
}

// *** Function B ***
std::string getFirstBook(Library * l){
    return(l->numBooks > 0) ? l->titles[0] : "None";
}

// *** Function C ***
std::string getFirstBook(Library & l){
    return (l.numBooks > 0) ? l.titles[0] : "None";
}
```

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++?

Memory Management — Ownership

```
class Library{
public:
    void addBook(Book * book);


    void removeBook(std::string title);


    void returnBook(Book * book);
private:

    std::vector<Book*> in;


    std::vector<Book*> out;


};
```

Does Library ‘own’ Books?

A) **Yes!**

B) **No!**

C) **Not sure**

Memory Management — Ownership

```cpp
class Library{
public:
    void addBook(Book * book);


    void removeBook(std::string title);


    void returnBook(Book * book);
private:

    std::vector<Book*> in;


    std::vector<Book*> out;


};
```

Does Library ‘own’ Books?

A) **Yes!**

B) **No!**

C) **Not sure**

Are they destroyed when the Library destructor is called?

Memory Management — Ownership

```
class Library{
public:
    void addBook(Book book);


    void removeBook(std::string title);


    void returnBook(Book book);
private:

    std::vector<Book> in;


    std::vector<Book> out;



};
```

Does Library 'own' Books?

A) **Yes!**

B) **No!**

C) **Not sure**

Memory Management — Ownership

```
class Library{
public:
    void addBook(Book book);


    void removeBook(std::string title);


    void returnBook(Book book);
private:

    std::vector<Book> in;


    std::vector<Book> out;



};
```

Does Library ‘own’ Books?

A) **Yes!**

B) **No!**

C) **Not sure**

Are they destroyed when the Library destructor is called?

Memory Management — Ownership

```
class Library{
public:
    void addBook(const Book& book);


    void removeBook(std::string title);


    void returnBook(const Book& book);
private:

    std::vector<Book*> in;


    std::vector<Book*> out;


};
```

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

```
class Library {
public:
    int numBooks;
    std::string * titles;
    ~Library();
    Library( int num, std::string* list );
};

Library::~Library(){
   delete titles;
   titles = nullptr;
}

Library::Library(int num, std::string* list){
    numBooks = inNum;
    titles = new std::string[ inNum ];
    std::copy(inList, inList + inNum, titles);
}

int main(){
   std::string myBooks[3] = {"A", "B", "C"};
   Library L1( 3, myBooks );
   Library L2( L1 );
   return 0;
}
```

```cpp
class Library {
public:
    int numBooks;
    std::string * titles;
    ~Library();
    Library( int num, std::string* list );
};

Library::~Library(){
   delete titles;
   titles = nullptr;
}

Library::Library(int num, std::string* list){
    numBooks = inNum;
    titles = new std::string[ inNum ];
    std::copy(inList, inList + inNum, titles);
}

int main(){
   std::string myBooks[3] = {"A", "B", "C"};
   Library L1( 3, myBooks );
   Library L2( L1 );
   return 0;
}
```

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

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**

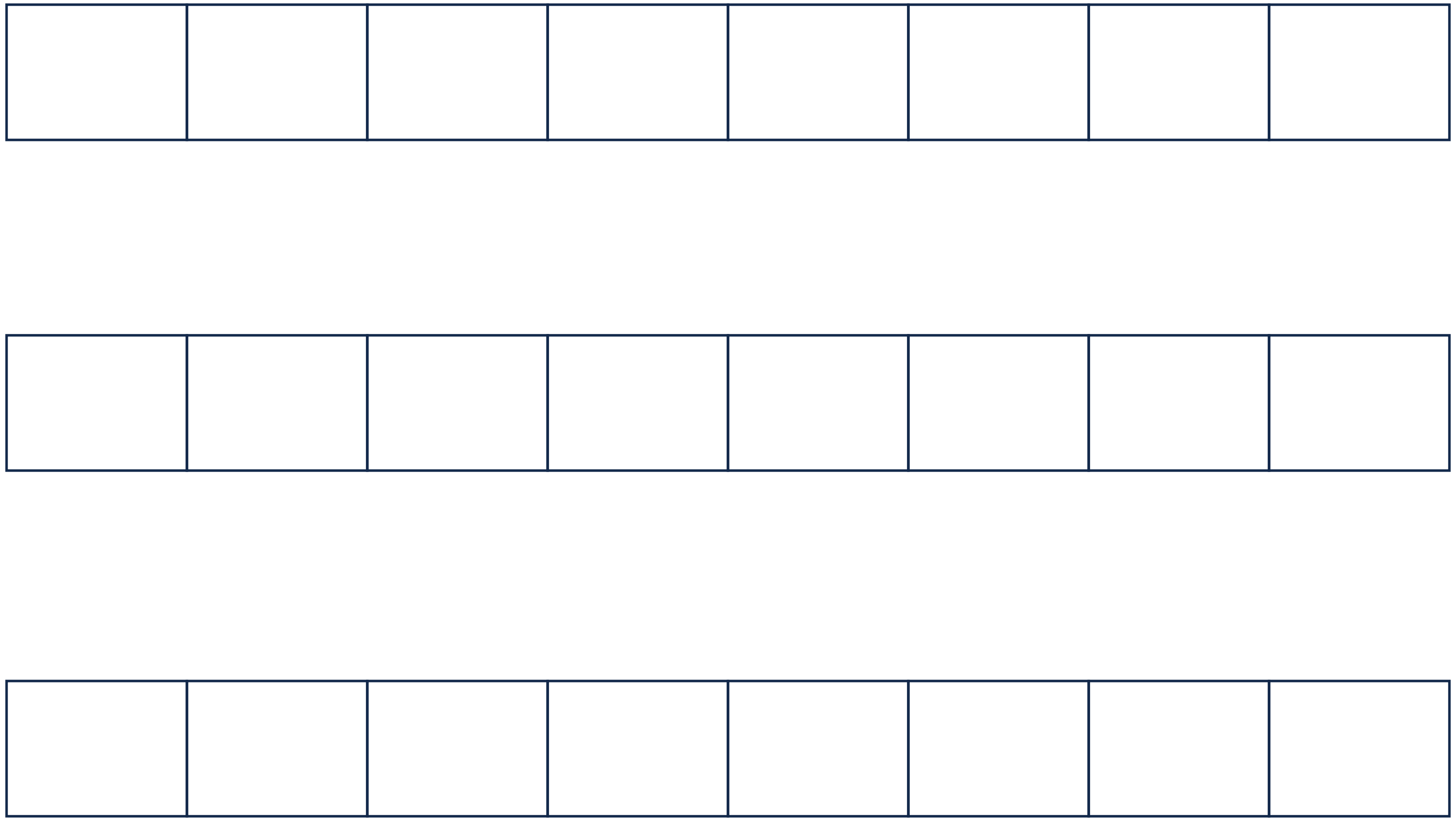
```
template <typename T>
T sum(T a, T b){
...
}
```

template1.cpp

```cpp
template <typename T>
T max(T a, T b) {
  T result;
  result = (a > b) ? a : b;
  return result;
}

```

Templates are very useful!



List Abstract Data Type

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