# Testing, Vectors, and Grids

Elyse Cornwall

June 29th, 2023

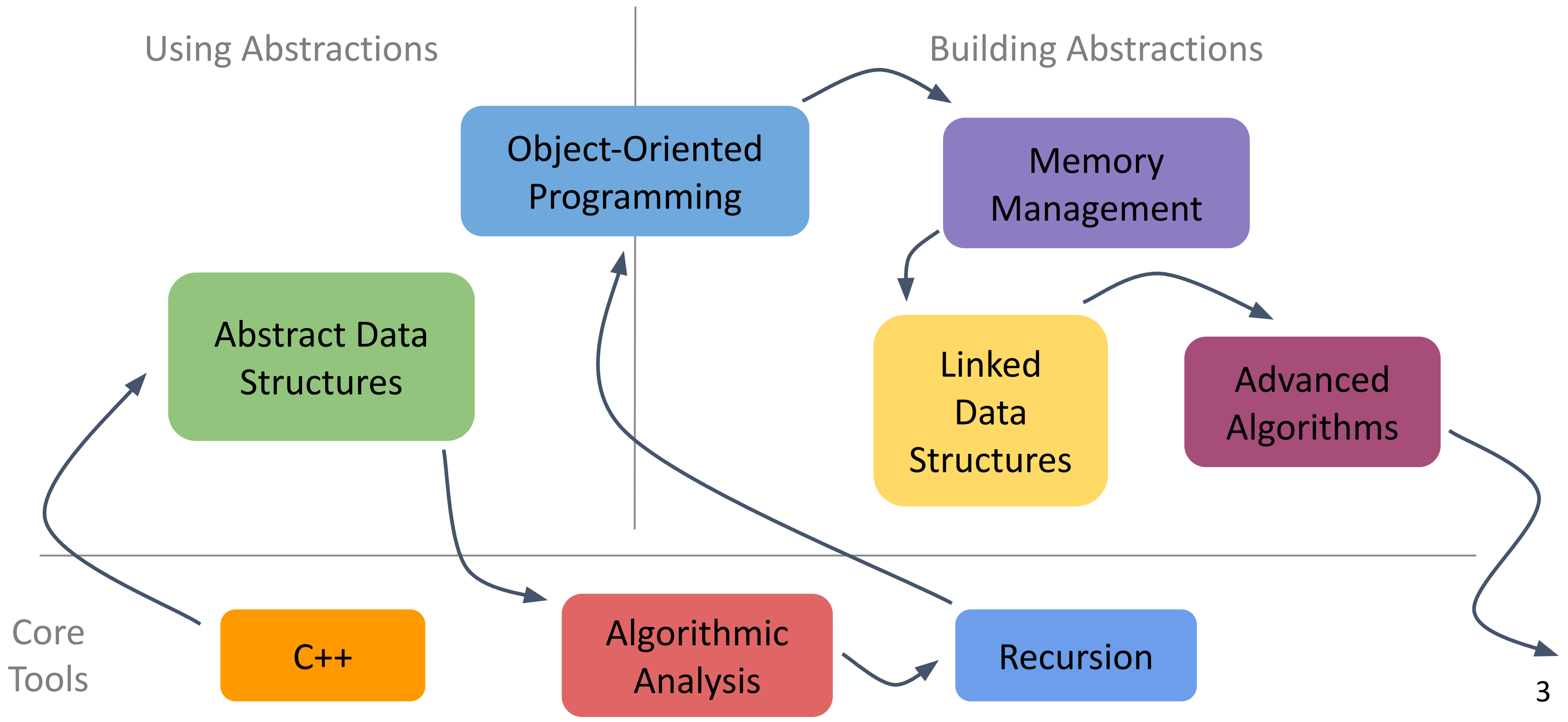
Contributions made from previous CS106B Instructors

Stanford University

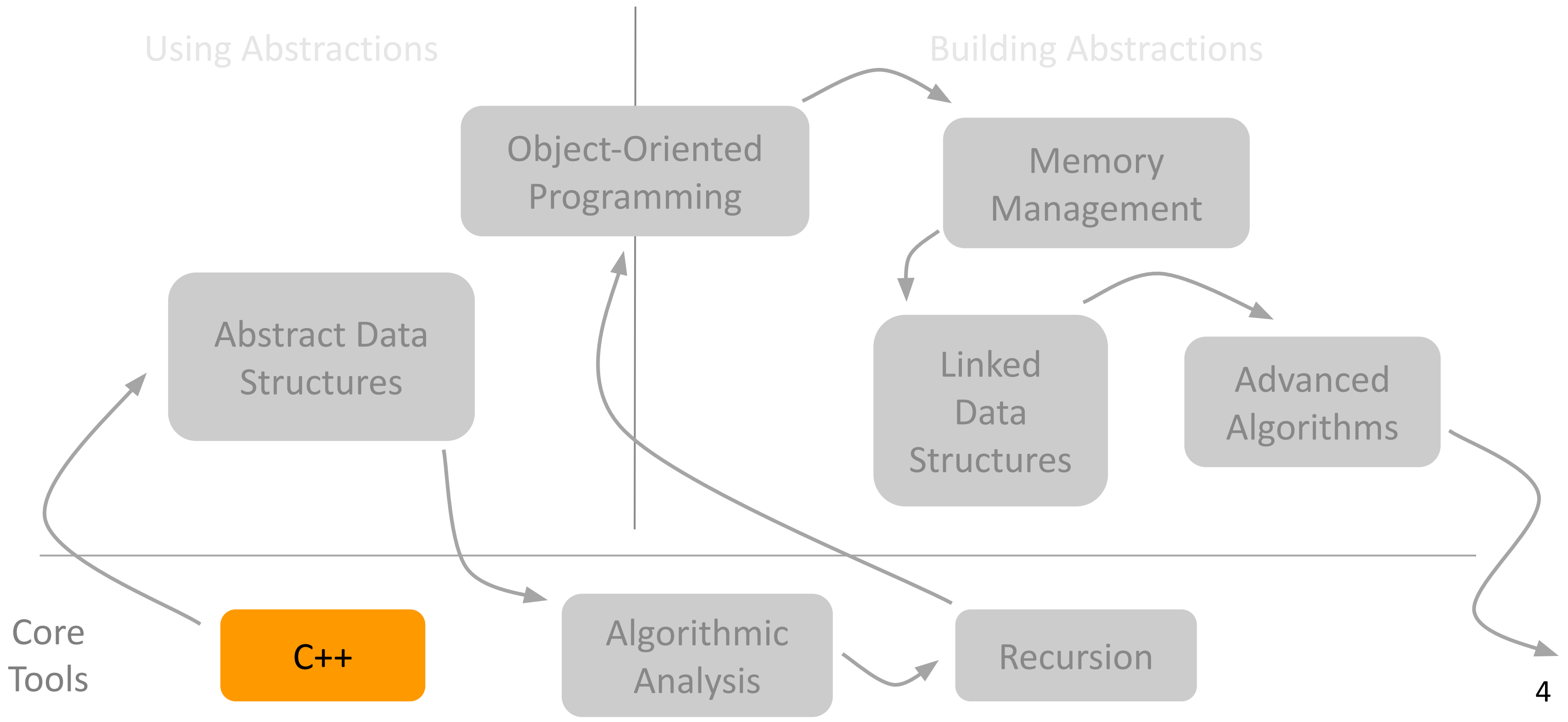
## Announcements

- Section policy reminder
  - You have one section absence - no need to ask for permission
  - You can attend another section if you have to miss your usual one
- Assignment 0 is due tomorrow
  - If you had install issues, go to LaIR, office hours, or chat with us after class
- Assignment 1 will be released tomorrow afternoon
- Assignment 1 YEAH Hours on Friday from 3-4pm at this Zoom link
  - Get started on the assignment early and ask any questions!

# CS106B Roadmap



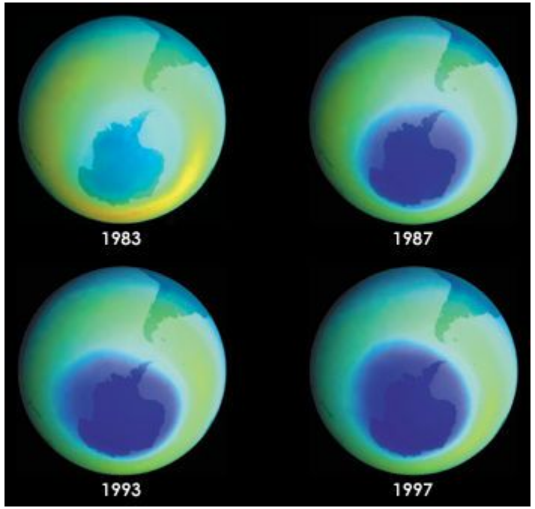
# CS106B Roadmap



## Testing

## Why is Testing Important?

For eight years, NASA's software discarded data that deviated from expected measurements, ignoring a growing hole in the ozone layer



source

## Why is Testing Important?

MCAS flight control software led to Boeing 737 MAX plane crashes

source

## Why is Testing Important?

- Software bugs have can have expensive, even deadly consequences
- As programmers, we take pride in building things that work well
- The key to writing robust, working code is writing good tests

## Testing Strategies

- "Test-as-you-go"
  - After each step, test thoroughly (don't wait until the end)

## Testing Strategies

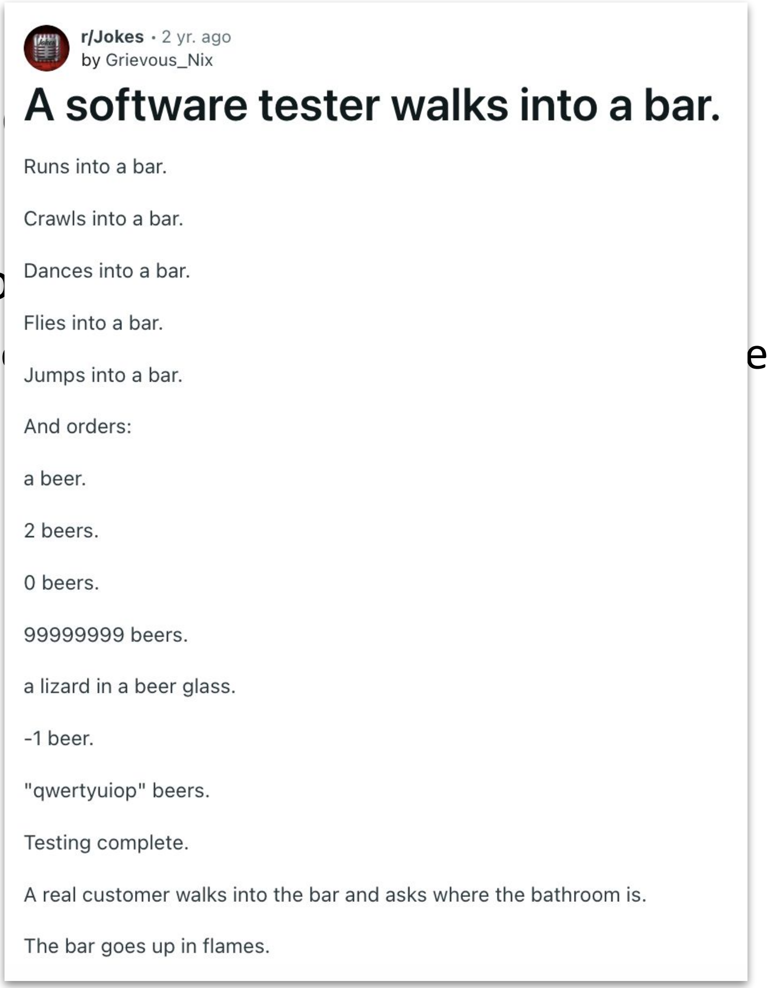
- "Test-as-you-go"
  - After each step, test thoroughly (don't wait until the end)
- Basic use cases

## Testing Strategies

- “Test-as-you-go”
    - After each step, test thoroughly (don’t wait until the end)
- Basic use cases
- Edge cases

## Testing Strat

- "Test-as-you-go
  - After each st e end)
- Basic use cases
- Edge cases



r/Jokes • 2 yr. ago
by Grievous_Nix

**A software tester walks into a bar.**

Runs into a bar.

Crawls into a bar.

Dances into a bar.

Flies into a bar.

Jumps into a bar.

And orders:

a beer.

2 beers.

0 beers.

99999999 beers.

a lizard in a beer glass.

-1 beer.

"qwertyuiop" beers.

Testing complete.

A real customer walks into the bar and asks where the bathroom is.

The bar goes up in flames.

## SimpleTest Library

Check out the SimpleTest guide

# What is SimpleTest?

- A library written by Stanford lecturers to make it easier to unit test your C++ code
  - Kind of like doctests in Python
  - `#include "testing/SimpleTest.h"`
- You’ll use SimpleTest a lot this quarter, starting with Assignment 1!

## Let’s Test Reversed

```
// reversed(str) returns copy of str with characters in reverse order

string reversed(string s) {
    string result;
    for (int i = s.length() - 1; i >= 0; i--) {
        result += s[i];
    }
    return result;
}
```

## Let’s Test Reversed

```
// reversed(str) returns copy of str with characters in reverse order

string reversed(string s) {
    string result;
    for (int i = s.length() - 1; i >= 0; i--) {
        result += s[i];
    }
    return result;
}
```

*Note, uninitialized strings get set to default value: empty string*

# Let’s Test Reversed

```
// reversed(str) returns copy of str with characters in reverse order

string reversed(string s) {
    string result;
    for (int i = s.length() - 1; i >= 0; i--) {
        result += s[i];
    }
    return result;
}
```

*Can it reverse the string “cat”?*

*What about “racecar”?*

*What should it return for “”?*

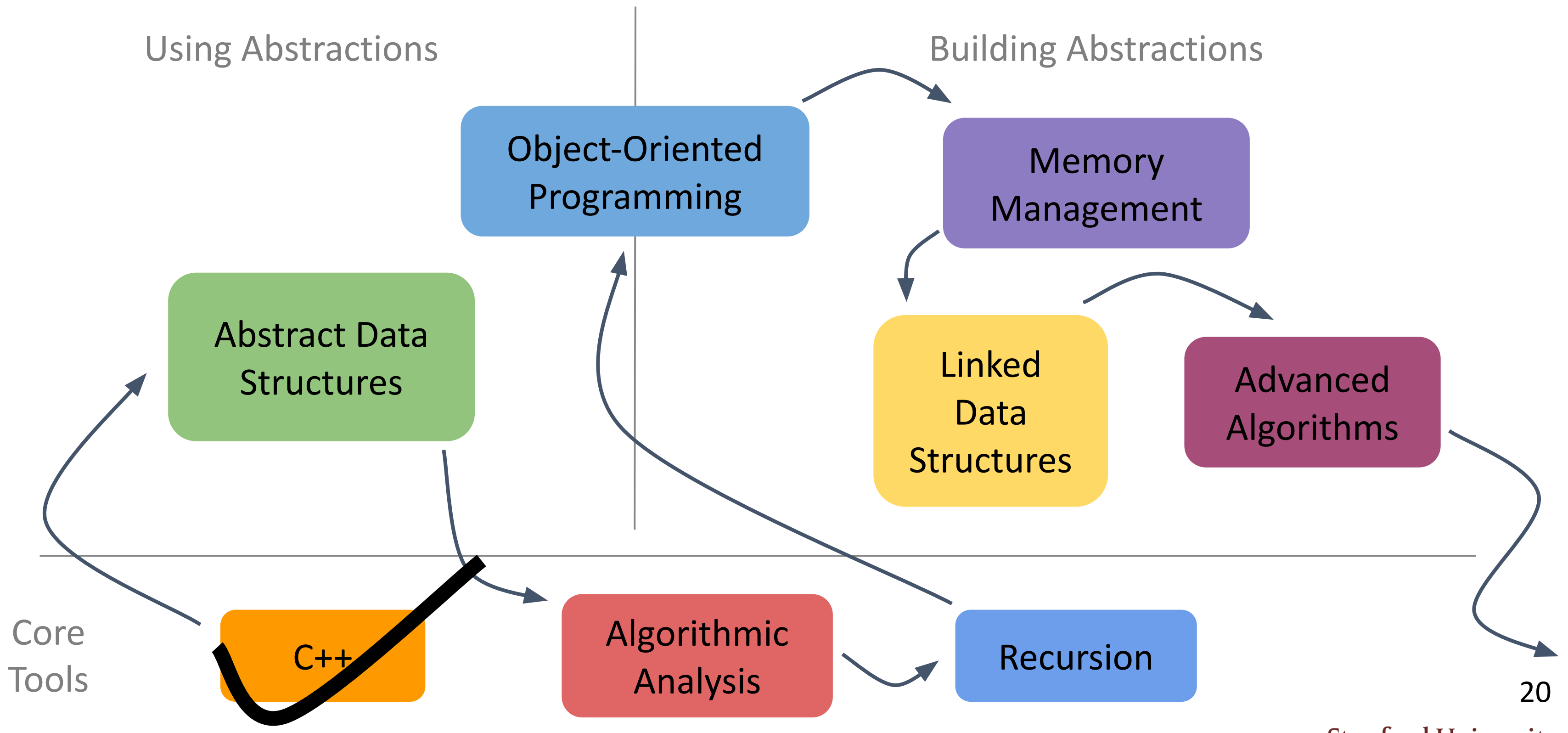
## Let’s Test Reversed

```
string reversed(string s) {
   // implementation here
}

/* * * * * * Test Cases * * * * * */
PROVIDED_TEST("Test reversed function") {
    EXPECT_EQUAL(reversed("cat"), "tac");
    EXPECT_EQUAL(reversed("racecar"), "racecar");
    EXPECT_EQUAL(reversed(""), "");
}
```

## SimpleTest Operations

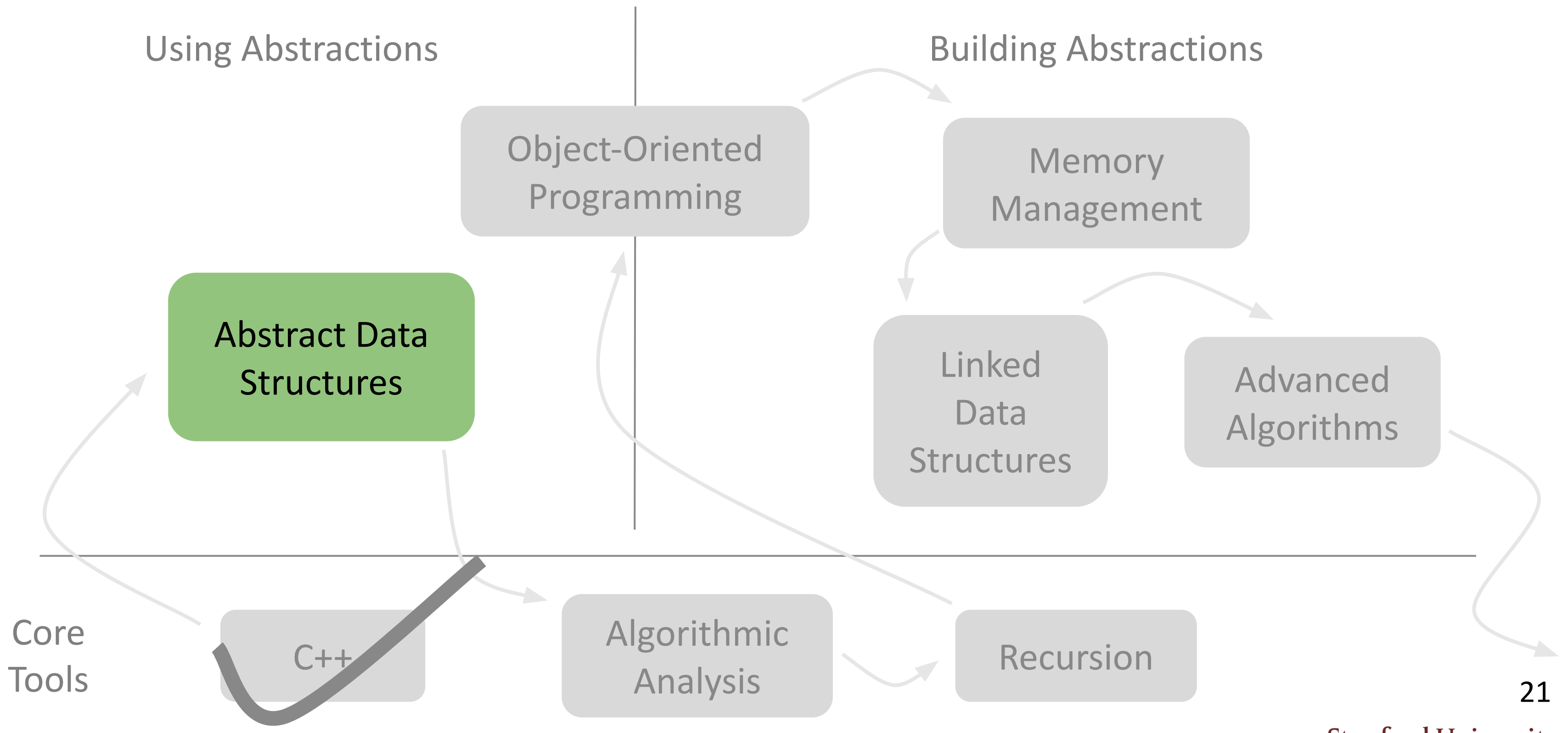
- `EXPECT_EQUAL(a, b)` - passes if a is equal to b
- `EXPECT(a)` - passes if the expression a is true
- `EXPECT_ERROR(a)` - passes if the expression raises an error
- `EXPECT_NO_ERROR(a)` - passes if the expression doesn’t raise an error
- `TIME_OPERATION(size, operation)` - times an operation

## Roadmap



Using Abstractions
Building Abstractions
Object-Oriented Programming
Memory Management
Abstract Data Structures
Linked Data Structures
Advanced Algorithms
Core Tools
C++
Algorithmic Analysis
Recursion

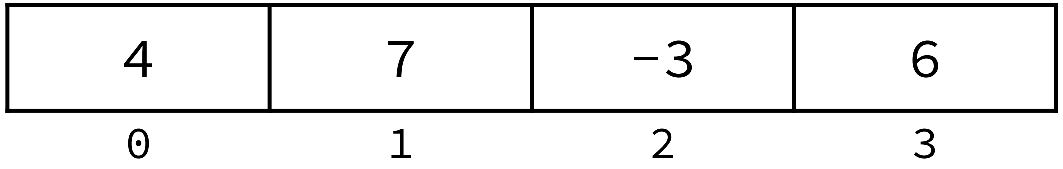
# Roadmap


Using Abstractions
Building Abstractions
Object-Oriented Programming
Memory Management
Abstract Data Structures
Linked Data Structures
Advanced Algorithms
Core Tools
C++
Algorithmic Analysis
Recursion

## Vectors

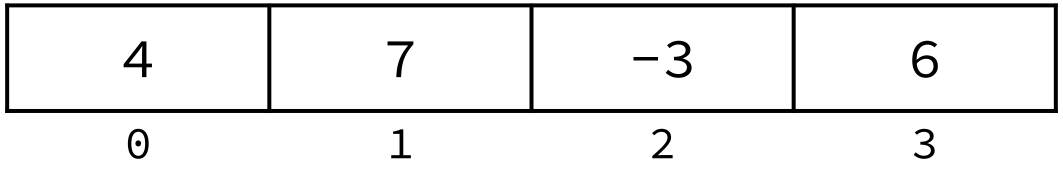
# What is a Vector?

- An abstract data type (ADT)
  - **Abstraction** that allows us to store data in an organized, structured way
- One of Stanford's C++ libraries (documentation here)
  - `#include "vector.h"`
- An ordered collection of elements that can grow and shrink in size
  - Like an `ArrayList` in Java or `list` in Python



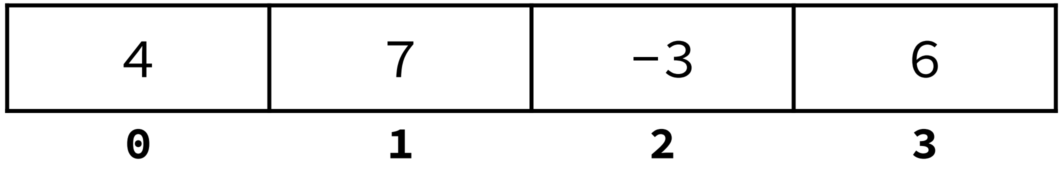
## Properties of a Vector

- Ordered (have indices)
- Can grow and shrink in size
- All elements must be of the same type



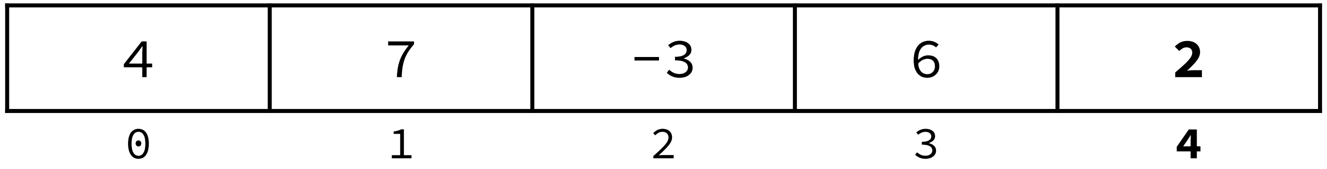
# Properties of a Vector

- **Ordered (have indices)**
- Can grow and shrink in size
- All elements must be of the same type



# Properties of a Vector

- Ordered (have indices)
- **Can grow and shrink in size**
- All elements must be of the same type



Stanford University

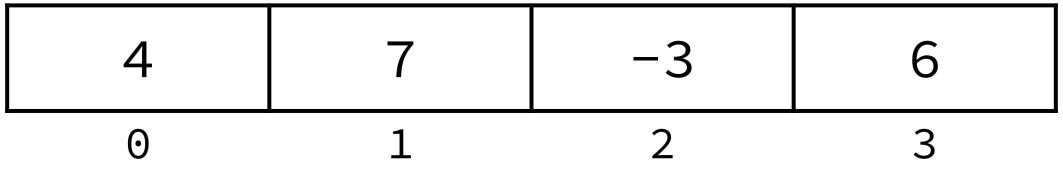
# Properties of a Vector

- Ordered (have indices)
- **Can grow and shrink in size**
- All elements must be of the same type

| | | | |
|---|---|---|---|
| 4 | 7 | -3 | 6 |
| 0 | 1 | 2 | 3 |

# Properties of a Vector

- Ordered (have indices)
- Can grow and shrink in size
- **All elements must be of the same type**



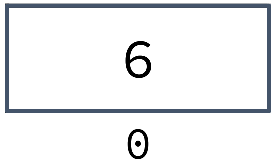
Stanford University

## Vector Operations: Creation

```
Vector<int> vec;  // creates an empty int vector
```

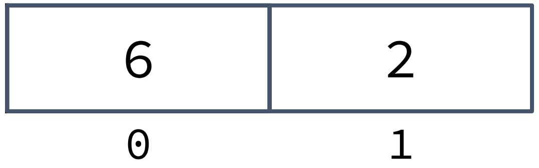
## Vector Operations: Adding Elements

```
Vector<int> vec;  // creates an empty int vector
vec.add(6);        // adds a new element
```



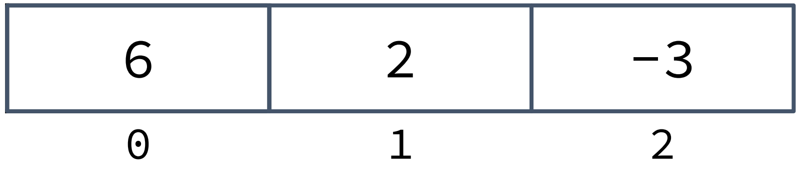
## Vector Operations: Adding Elements

```
Vector<int> vec;  // creates an empty int vector
vec.add(6);
vec.add(2);       // adds to end of vector
```



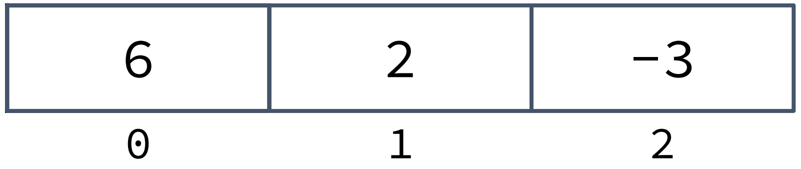
## Vector Operations: Adding Elements

```
Vector<int> vec;  // creates an empty int vector
vec.add(6);
vec.add(2);
vec.add(-3);
```



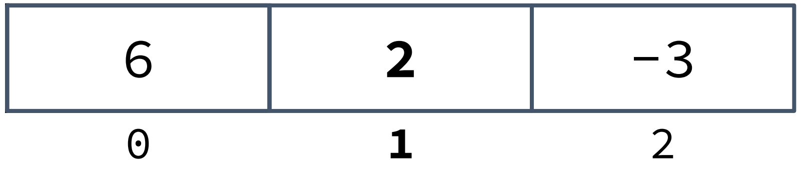
## Vector Operations: Creation with Elements

```
Vector<int> vec = {6, 2, -3};   // equivalent
```



## Vector Operations: Accessing Elements

```
Vector<int> vec = {6, 2, -3};   // equivalent
cout << vec[1] << endl;         // prints 2
```

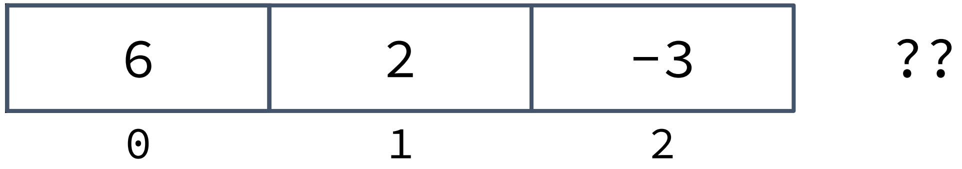


Stanford University

## Vector Operations: Accessing Elements(?)

```
Vector<int> vec = {6, 2, -3};   // equivalent
cout << vec[3] << endl;         // prints 2
```

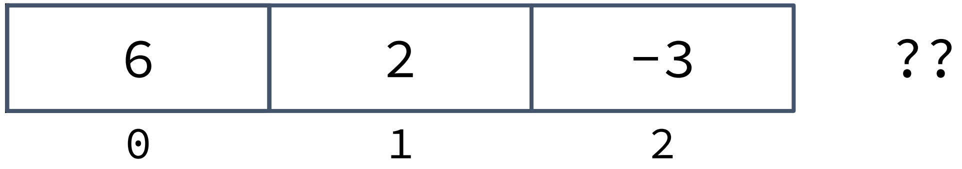
👥 *Talk with a neighbor, what will happen?*



## Vector Operations: Accessing Elements(?)

```
Vector<int> vec = {6, 2, -3};   // equivalent
cout <
```

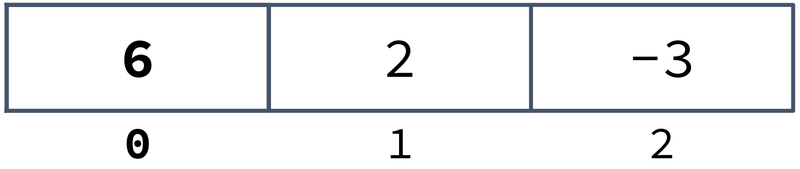
```
*** A fatal error was reported:

Vector::operator []: index of 3
is outside of valid range [0..2]
```



## Vector Operations: Removing Elements

```
Vector<int> vec = {6, 2, -3};   // equivalent
vec.remove(0);
```

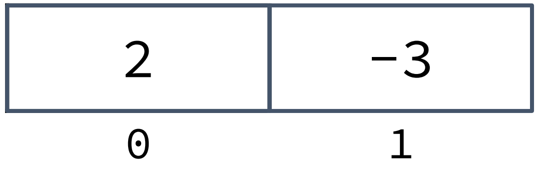
*Specify index to remove at*



## Vector Operations: Removing Elements

```
Vector<int> vec = {6, 2, -3};   // equivalent
vec.remove(0);
```

*Specify index to remove at*

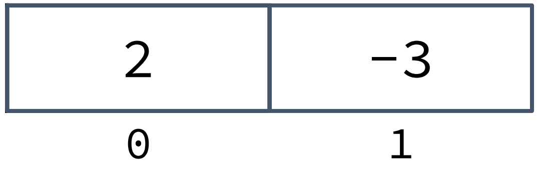


Stanford University

# Vector Operations: Getting Size

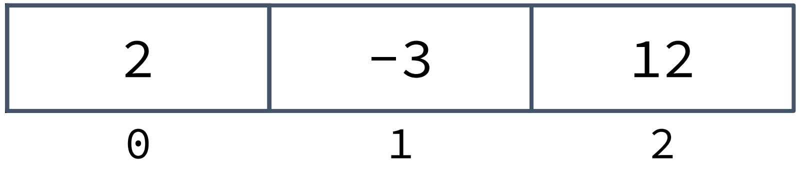
```
cout << vec.size() << endl;     // prints 2
```

*Number of elements currently in vector*



## Vector Operations: Getting Size

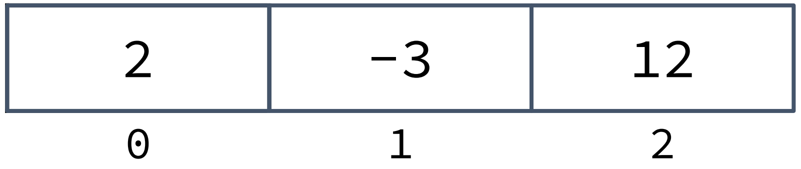
```
cout << vec.size() << endl;     // prints 2
vec.add(12);
```



Stanford University

## Vector Operations: Getting Size

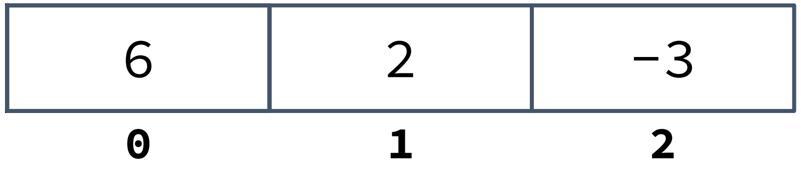
```
cout << vec.size() << endl;     // prints 2
vec.add(12);
cout << vec.size() << endl;     // prints 3
```



# Traversing a Vector

```
// Method 1: Traditional for loop
Vector<int> vec = {6, 2, -3};
for (int i = 0; i < vec.size(); i++) {
    cout << vec[i] << endl;
}
```

*Loops over indices: 0, 1, 2*

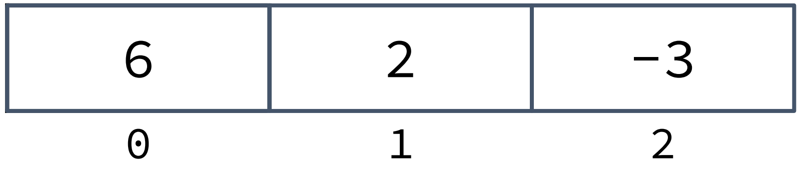


## Traversing a Vector

```
// Method 1: Traditional for loop
Vector<int> vec = {6, 2, -3};
for (int i = 0; i < vec.size(); i++) {
    cout << vec[i] << endl;
}
```

```
Output:
6
2
-3
```

*Loops over indices: 0, 1, 2*

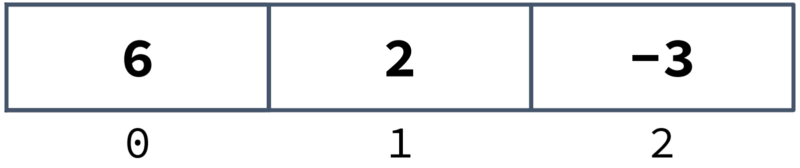


Stanford University

# Traversing a Vector

```
// Method 1: Traditional for loop
Vector<int> vec = {6, 2, -3};
for (int i = 0; i < vec.size(); i++) {
    cout << vec[i] << endl;
}

// Method 2: For-each loop
Vector<int> vec = {6, 2, -3};
for (int num: vec) {
    cout << num << endl;
}
```

*Loops over the elements*



Stanford University

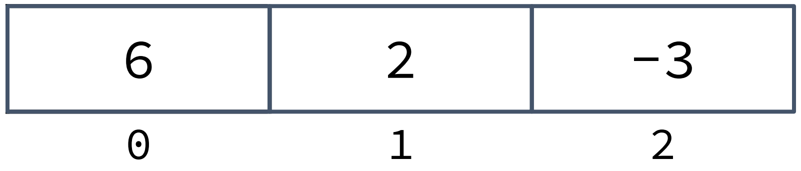
# Traversing a Vector

```
// Method 1: Traditional for loop
Vector<int> vec = {6, 2, -3};
for (int i = 0; i < vec.size(); i++) {
    cout << vec[i] << endl;
}
```

```
Output:
6
2
-3
```

```
// Method 2: For-each loop
Vector<int> vec = {6, 2, -3};
for (int num: vec) {
    cout << num << endl;
}
```

*Loops over the elements*



Stanford University

# The Stanford Vector Library

- `vec.size()`: Returns the number of elements in the vector.
- `vec.isEmpty()`: Returns true if the vector is empty, false otherwise.
- `vec[i]`: Selects the ith element of the vector.
- `vec.add(value)`: Adds a new element to the end of the vector.
- `vec.insert(index, value)`: Inserts the value before the specified index, and moves the values after it up by one index.
- `vec.remove(index)`: Removes the element at the specified index, and moves the rest of the elements down by one index.
- `vec.clear()`: Removes all elements from the vector.
- `vec.sort()`: Sorts the elements in the list in increasing order.

For more information, check out the Stanford Vector class documentation!

## Let’s Trace Some Code

```
void doubleVec(Vector<int> vec) {
    for (int i = 0; i < vec.size(); i++) {
        vec[i] = vec[i] * 2;
    }
}
```

## Let’s Trace Some Code

```
void doubleVec(Vector<int> vec) {
    for (int i = 0; i < vec.size(); i++) {
        vec[i] = vec[i] * 2;
    }
}
```

*This is a `void` function - it’s not returning anything.*

## Let's Trace Some Code

```
void doubleVec(Vector<int> vec) {
    for (int i = 0; i < vec.size(); i++) {
        vec[i] = vec[i] * 2;
    }
}

int main() {
    Vector<int> nums = {1, 2, 3, 4};
    doubleVec(nums);
    cout << nums << endl;
    return 0;
}
```

🎟️ *Attendance ticket: what gets printed in `main`?*
*(Let's test it!)*

# Let’s Trace Some Code

```
void doubleVec(Vector<int> vec) {
    for (int i = 0; i < vec.size(); i++) {
        vec[i] = vec[i] * 2;
    }
}

int main() {
    Vector<int> nums = {1, 2, 3, 4};
    doubleVec(nums);
    cout << nums << endl;
    return 0;
}
```

```
Output:
{1, 2, 3, 4}
```

*Remember, by default, parameters are passed by value in C++.*

## Let’s Trace Some Code

```
void doubleVec(Vector<int> vec) {
    for (int i = 0; i < vec.size(); i++) {
        vec[i] = vec[i] * 2;
    }
}

int main() {
    Vector<int> nums = {1, 2, 3, 4};
    doubleVec(nums);
    cout << nums << endl;
    return 0;
}
```

*How would we pass a parameter so that the callee could modify it?*

# Pass by Reference

## Let’s Compare

Pass by value

- Callee gets a copy of a variable from the caller function
- Changes to that variable that occur in callee do not persist in caller

# Let’s Compare

Pass by value

- Callee gets a copy of a variable from the caller function
- Changes to that variable that occur in callee do not persist in caller

Pass by reference

- Callee gets a **reference** to a variable from the caller function
- Now, the callee can directly modify the original variable

## Let’s Edit Some Code

```
void doubleVec(Vector<int> vec) {
    for (int i = 0; i < vec.size(); i++) {
        vec[i] = vec[i] * 2;
    }
}

int main() {
    Vector<int> nums = {1, 2, 3, 4};
    doubleVec(nums);
    cout << nums << endl;
    return 0;
}
```

## Let’s Edit Some Code

```
void doubleVec(Vector<int>& vec) {
    for (int i = 0; i < vec.size(); i++) {
        vec[i] = vec[i] * 2;
    }
}

int main() {
    Vector<int> nums = {1, 2, 3, 4};
    doubleVec(nums);
    cout << nums << endl;
    return 0;
}
```

*We add an ampersand after the type to indicate that it’s a reference (Let’s test it!)*

## Passing Other Types by Reference

```
void tripleWeight(double& weightRef) {
    weightRef *= 3; // triple the weight
}

int main() {
    double weight = 1.06;
    tripleWeight(weight);
    cout << weight << endl;  // prints 3.18
}
```

## Passing Other Types by Reference

```
void tripleWeight(double& weightRef) {
    weightRef *= 3; // triple the weight
}

int main() {
    double weight = 1.06;
    tripleWeight(weight);
    cout << weight << endl;  // prints 3.18
}
```

*However, this isn’t great style…*

# When Do We Pass by Reference?

Yes:

- When we want the callee function to edit our data
- To avoid making copies of large data structures
- When we need to return multiple values

## When Do We Pass by Reference?

Yes:

- When we want the callee function to edit our data
- To avoid making copies of large data structures
- When we need to return multiple values

No:

- Just because
  - Passing by reference is risky because another function can modify your data!
- When the data we're passing to the callee is small, and thus copying isn't expensive

## Grids

# What is a Grid?

- Another one of Stanford’s C++ libraries (documentation here)
  - `#include “grid.h”`
- A 2D array with fixed dimensions
  - *Array* not *Vector*, because it cannot grow or shrink; dimensions are set

| | | |
|---|---|---|
| 2 | 5 | -1 |
| 10 | 11 | 3 |
| 19 | -4 | -2 |
| 4 | 6 | 2 |

## Grid Operations: Creation

```
// Option 1: No initialization
Grid<int> grid;
```

## Grid Operations: Creation

```
// Option 1: No initialization
Grid<int> grid;
grid.resize(4, 3);  // must resize or reassign before using
```

|   |   |   |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

## Grid Operations: Creation

```
// Option 1: No initialization
Grid<int> grid;
grid.resize(4, 3);  // must resize or reassign before using
```

*Notice the grid has been filled with default values for this type*

| | | |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

# Grid Operations: Creation

```
// Option 2: Specify number of rows and columns
Grid<int> grid(4, 3);
```

*Notice the grid has been filled with default values for this type*

| | | |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

# Grid Operations: Creation

```
// Option 3: Fill in all elements
Grid<int> grid = {{2, 5, -1}, {10, 11, 3}, ... }
```

| | | |
|---|---|---|
| 2 | 5 | -1 |
| 10 | 11 | 3 |
| 19 | -4 | -2 |
| 4 | 6 | 2 |

# Grid Operations: Accessing Elements

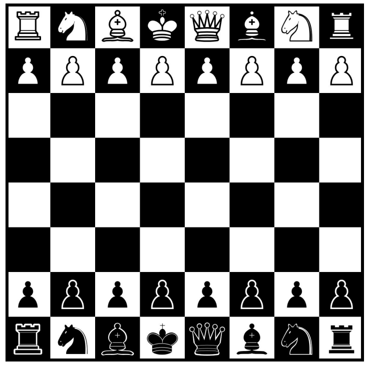
```
// Option 3: Fill in all elements
Grid<int> grid = {{2, 5, -1}, {10, 11, 3}, ... }
cout << grid[2][1] << endl;  // we do [row][col]
```

| | | |
|---|---|---|
| 2 | 5 | -1 |
| 10 | 11 | 3 |
| 19 | **-4** | -2 |
| 4 | 6 | 2 |

# The Stanford Grid Library

- `grid.numRows()`: Returns the number of rows in the grid.
- `grid.numCols()`: Returns the number of columns in the grid.
- `grid[i][j]`: selects the element in the ith row and jth column.
- `grid.resize(rows, cols)`: Changes the dimensions of the grid and re-initializes all entries to their default values.
- `grid.inBounds(row, col)`: Returns true if the specified row, column position is in the grid, false otherwise.

For more information, check out the Stanford Grid documentation!



👥 What kind of data might you store in a Vector? What about a Grid?

## Recap

- Testing
    - Test incrementally and often!
    - We’ll be using SimpleTest this quarter
- Vectors
    - Ordered data, grows and shrinks, all one type
- Pass by reference &
    - Allows us to modify the original variable when passed as parameter
- Grids
    - 2D arrays, fixed size, all one type

Thanks! See you next week 😎