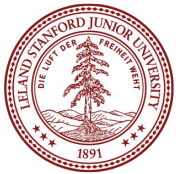
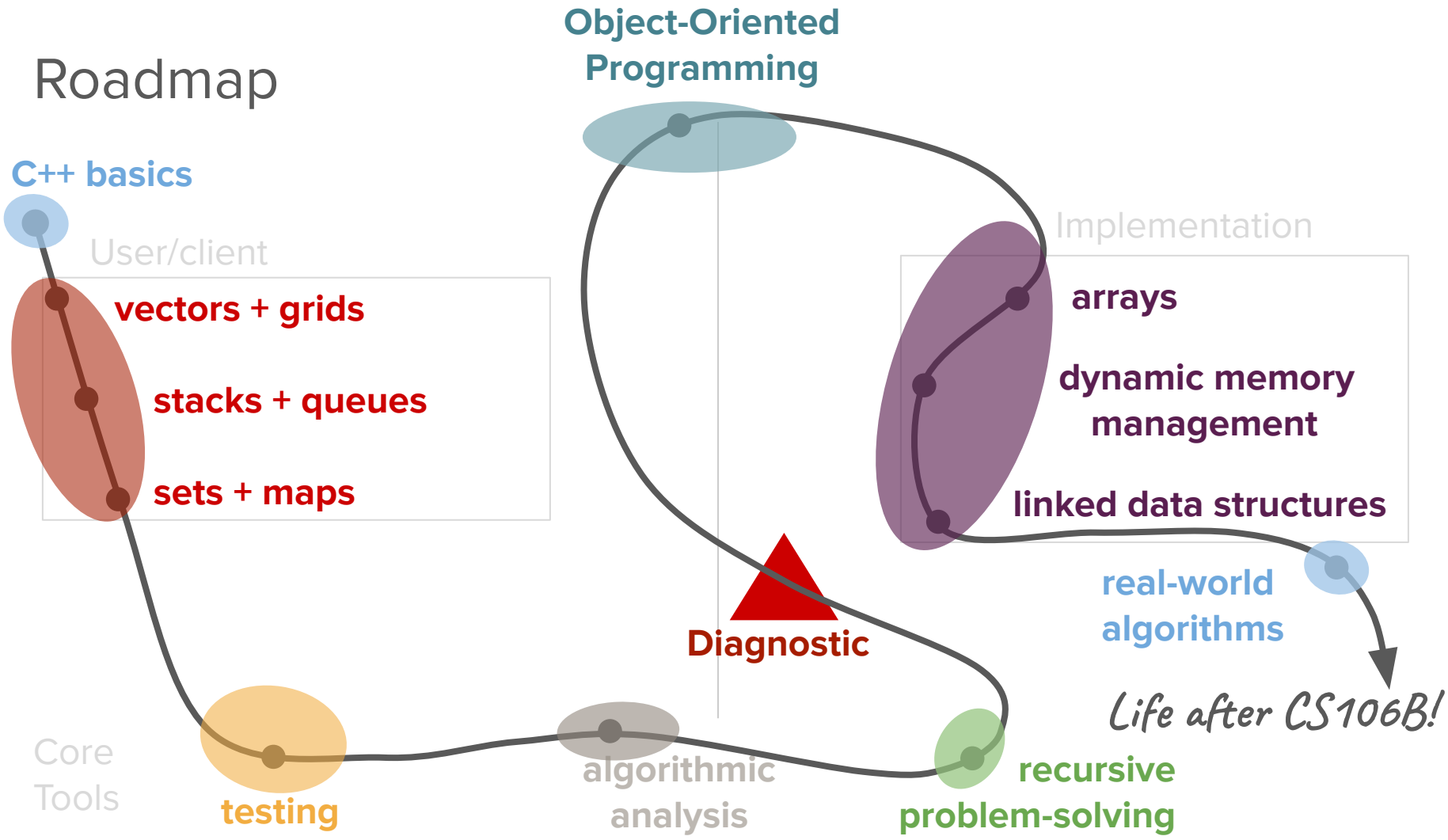
Console Programs, Vectors, and Grids

What is the first thing that comes to your mind when you think of the phrase "data structure"?

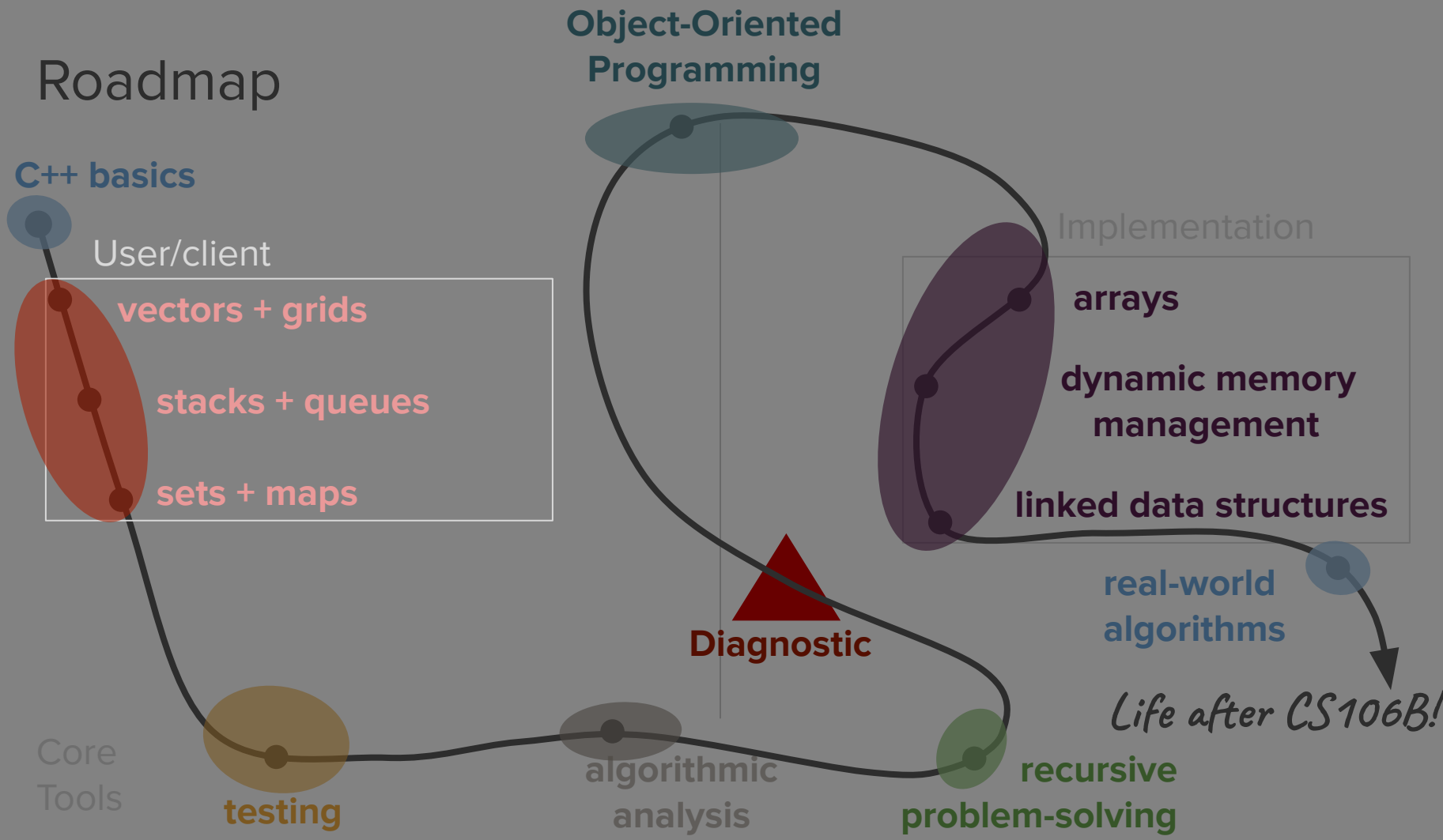
(put your answers the chat)



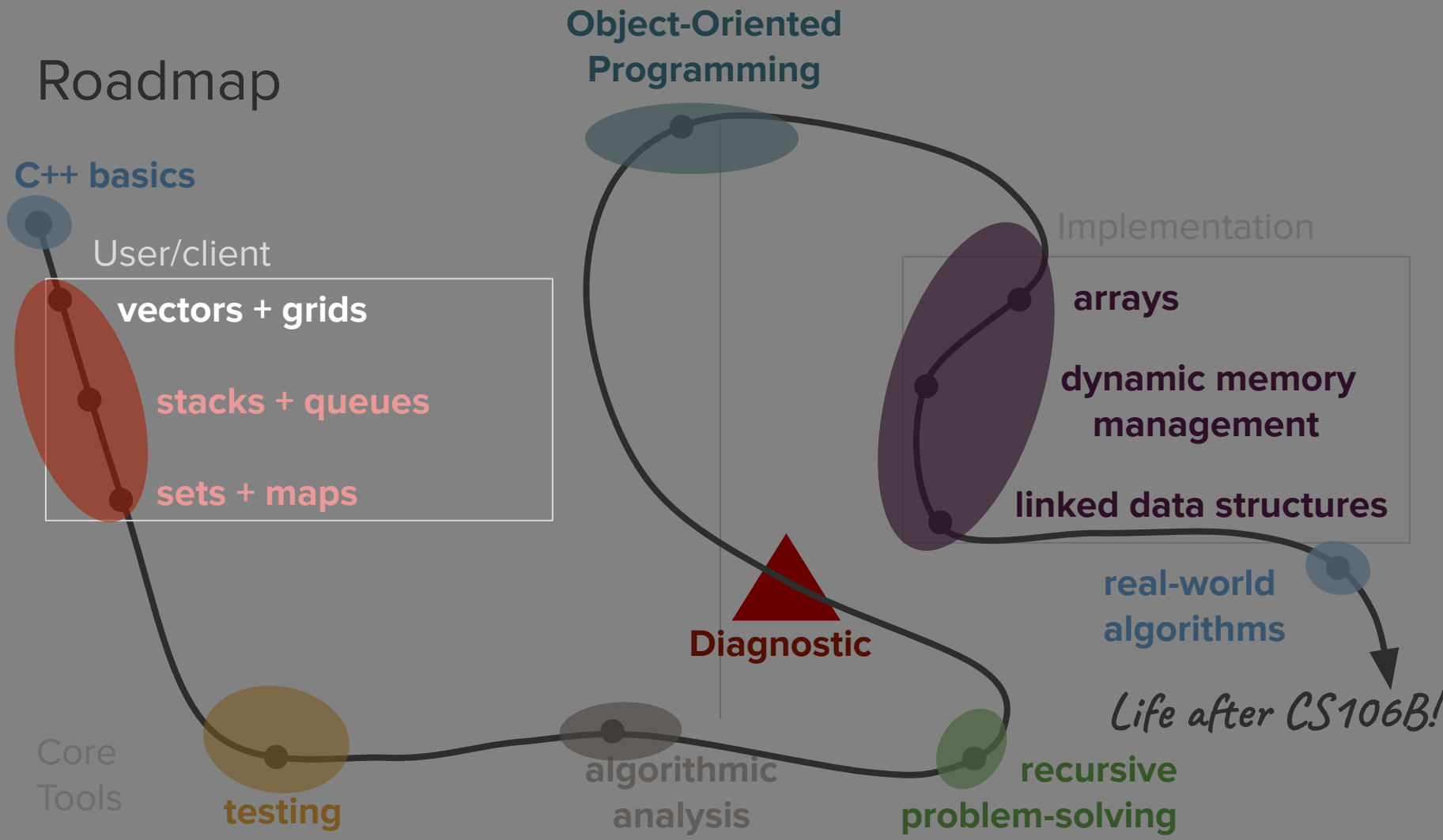
Roadmap


Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

Roadmap


Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

Roadmap


Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

Today's questions

How do we build programs that interact with users?

How do we structure data using abstractions in code?

Today's topics

1. Review (strings, testing, and SimpleTest)
2. Console Programs
3. Abstract Data Types
    a. Vectors
    b. Grids (time permitting)
4. Pass by reference

Review

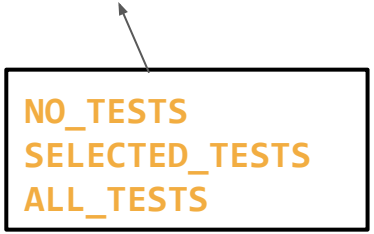
(strings, testing and SimpleTest)

SimpleTest

How does SimpleTest work?

main.cpp

```
#include "testing/SimpleTest.h"
#include "testing-examples.h"

int main()
{
    if (runSimpleTests(SELECTED_TESTS)) {
        return 0;
    }

    return 0;
}
```



How does SimpleTest work?

main.cpp

```
#include "testing/SimpleTest.h"
#include "testing-examples.h"

int main()
{
    if (runSimpleTests(SELECTED_TESTS)) {
        return 0;
    }

    return 0;
}
```

testing-examples.cpp

```
#include "testing/SimpleTest.h"

int factorial (int num);

int factorial (int num) {
     /* Implementation here */
}

PROVIDED_TEST("Some provided tests.") {
     EXPECT_EQUAL(factorial(1), 1);
     EXPECT_EQUAL(factorial(2), 2);
     EXPECT_EQUAL(factorial(3), 6);
     EXPECT_EQUAL(factorial(4), 24);
}

STUDENT_TEST("student wrote this test") {
     // student tests go here!
}
```

How does SimpleTest work?

main.cpp

```
#include "testing/SimpleTest.h"
#include "testing-examples.h"

int main()
{
    if (runSimpleTests(SELECTED_TESTS)) {
        return 0;
    }

    return 0;
}
```

testing-examples.cpp

```
#include "testing/SimpleTest.h"

int factorial (int num);

int factorial (int num) {
     /* Implementation here */
}

PROVIDED_TEST("Some provided tests.") {
     EXPECT_EQUAL(factorial(1), 1);
     EXPECT_EQUAL(factorial(2), 2);
     EXPECT_EQUAL(factorial(3), 6);
     EXPECT_EQUAL(factorial(4), 24);
}

STUDENT_TEST("student wrote this test") {
     // student tests go here!
}
```

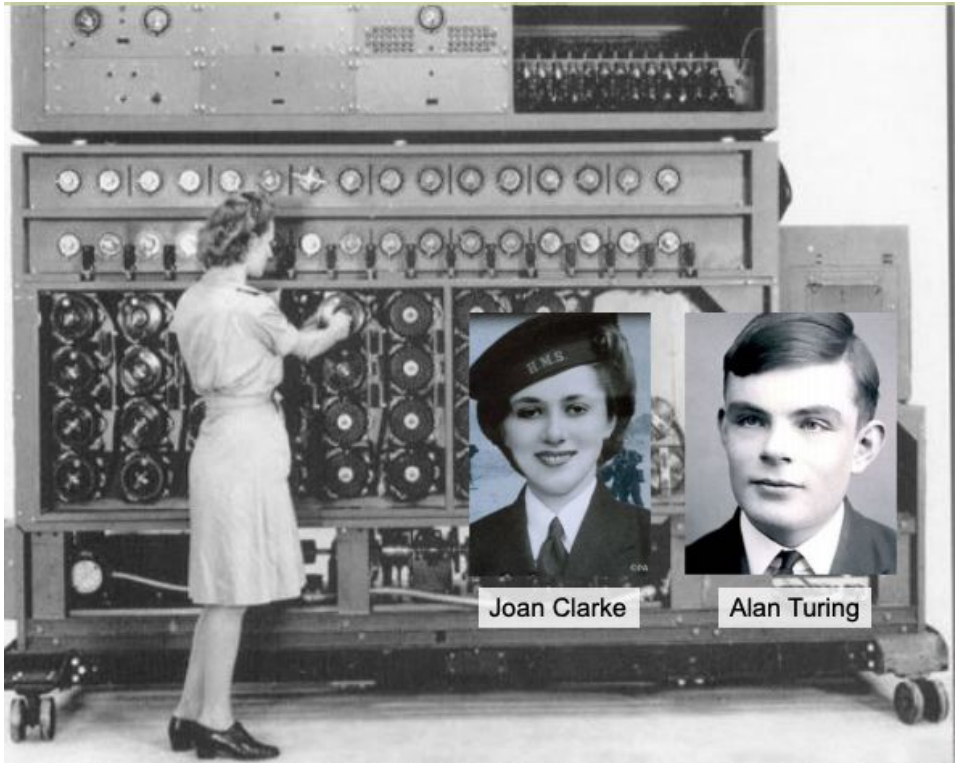
How do we solve interesting problems with strings?

Real-life problems involving strings

- Encryption and decryption

```
string encrypted = 'Jvkpun pz mbu';

string decrypted = 'Coding is fun';
```

Bonus: What cipher is this?



Real-life problems involving strings

- Encryption and decryption
- Language translation

```
string input = "¿Dónde está la
biblioteca?";
string output = "Where is the
library?";
```

*This result cost billions of dollars (adjusted for inflation)

Real-life problems involving strings

- Encryption and decryption
- Language translation
- DNA Analysis

```
string input = "ATGCCGATGTGC";

output = gene analysis,

homology score, etc.
```

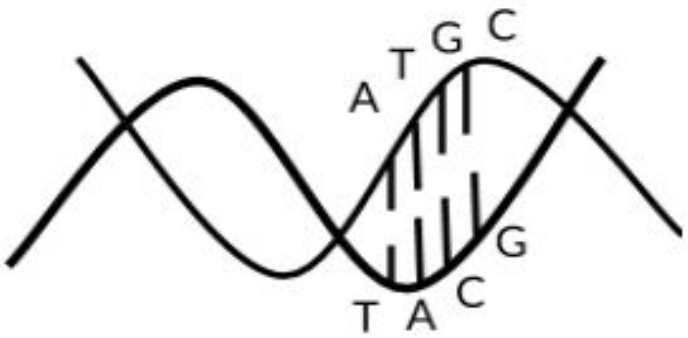
Real-life problems involving strings

- Encryption and decryption
- Language translation
- DNA Analysis

```
string input = "ATGCCGATGTGC";

output = gene analysis,

homology score, etc.
```

Generating DNA Complement Sequences

- In biology, you might have learned that the fundamental unit of DNA is a nucleotide, or base.
- The four possible bases for DNA are Guanine (G), Cytosine (C), Adenine (A), and Thymine (T).
- These nucleotides form “base pairs” that make up complementary strands of DNA (which create its double-helix structure).
- A pairs with T, and G pairs with C.



Generating DNA Complement Sequences

We want to write a function with the prototype

```
string complement (string dnaStrand)
```

which takes in a strand of DNA as a string and returns its complement as a string.

Generating DNA Complement Sequences

We want to write a function with the prototype

```
string complement (string dnaStrand)
```

which takes in a strand of DNA as a string and returns its complement as a string.

The function's output should be **case-insensitive**; that is, `complement("ATG")` and `complement("aTg")` should return the same result => `"TAC"`. All output should be in uppercase.

Generating DNA Complement Sequences

We want to write a function with the prototype

```
string complement (string dnaStrand)
```

which takes in a strand of DNA as a string and returns its complement as a string.

The function's output should be **case-insensitive**; that is, `complement("ATG")` and `complement("aTg")` should return the same result => `"TAC"`. All output should be in uppercase.

The function can assume that all of the base pairs of the input string are valid DNA base pairs– that is, the string consists only of the following characters: `'a'`, `'A'`, `'g'`, `'G'`, `'t'`, `'T'`, `'c'`, `'C'`

Your Task (`instructions.txt`)

- We've provided a buggy implementation of **`complement`** for you in the public Ed workspaces. We've also provided some tests, but all of the tests currently pass, so they haven't yet unearthed the bug in the code.
- You and your breakout room group members have three tasks:
  - Write at least one additional test that uncovers the bug in the provided implementation.
  - Fix the bug and confirm that your new test passes.
  - Make sure to add a more accurate name to the STUDENT_TEST identifier in the code. Discuss with your group what other tests/groups of tests you might add if you had more time to make the code more robust.

Breakout rooms! (5 minutes)

(Ed workspace)

DNA Exercise Recap

- What sort of test cases were not being covered?
  - Inputs with lowercase letters!
  - Example of a test that you could have added to surface an error
    - ```
      STUDENT_TEST ("DNA strand with lowercase letter") {
          EXPECT_EQUAL(complement("aTg"), "TAC");
      }
      ```
      ```

DNA Exercise Recap

- What sort of test cases were not being covered?
  - Inputs with lowercase letters!
  - Example of a test that you could have added to surface an error
    - ```
      STUDENT_TEST ("DNA strand with lowercase letter") {
          EXPECT_EQUAL(complement("aTg"), "TAC");
      }
      ```
- How do you fix the bug?
  - Need to do conversion of the characters in the string to lowercase!
  - Could add `ch = toupper(ch)` as the first line inside the for loop
  - Could convert the whole string to uppercase before starting the loop
  - Less optimal: check all 8 cases with if statements (for upper and lower case bases)
      ```

DNA Exercise Recap

- What sort of test cases were not being covered?
  - Inputs with lowercase letters!
  - Example of a test that you could have added to surface an error

```
STUDENT_TEST ("DNA strand with lowercase letter") {
    EXPECT_EQUAL(complement("aTg"), "TAC");
}
```

- How do you fix the bug?
  - Need to do conversion of the characters in the string to uppercase!
  - Could add `ch = toupper(ch)` as the first line inside the for loop
  - Could convert the whole string to uppercase before starting the loop
  - Less optimal: check all 8 cases with if statements (for upper and lower case bases)

Style tip: Minimize the number of hardcoded checks/conditional statements!

How do we build programs that interact with users?

How do we build programs that interact with users?

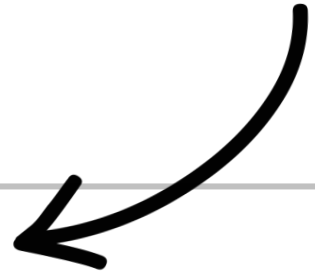
Console programs!

Definition

Console program

A program that uses the interactive terminal (console) as a communication boundary with the user.

An abstraction for the user!



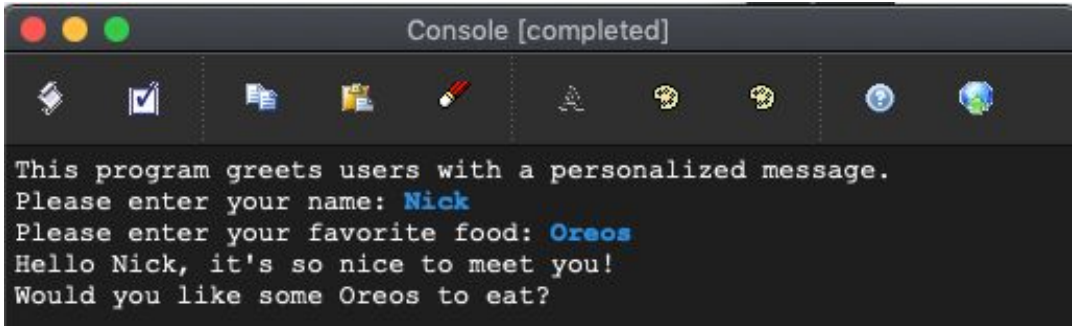
Definition

Console program

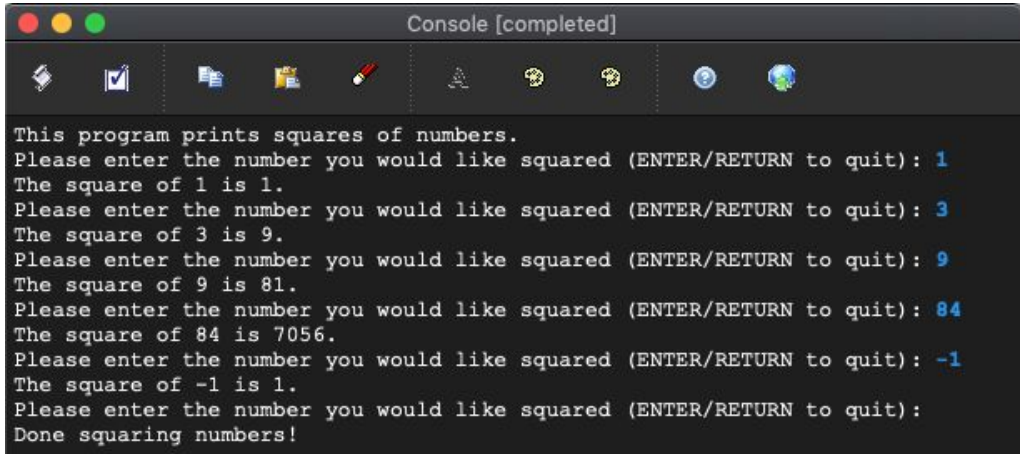
A program that uses the interactive terminal (console) as a communication boundary with the user.

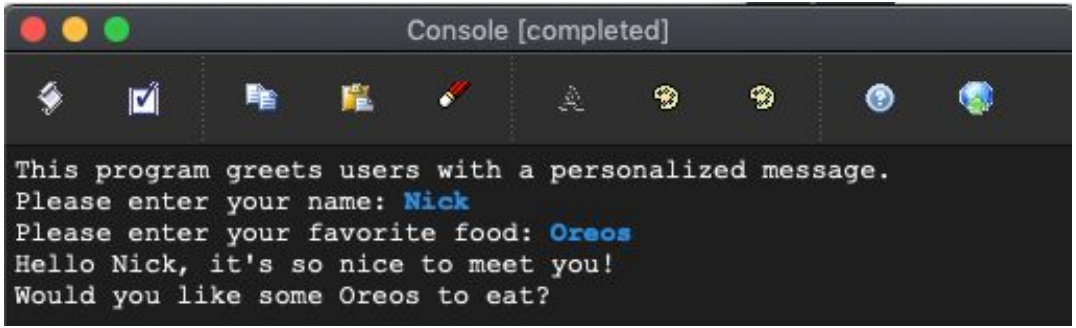
Some example console programs

Some example console programs



Some example console programs





How do we get information from the user?

How do we get information from the user?

The interactive terminal (console) and the **`getLine()`** *function!*

The console and the `getLine()` function

The console and the `getLine()` function

- The console is the text-output area that we have already seen when using `cout` to display information. In addition to displaying text, the console can also solicit text from a user.

The console and the `getLine()` function

- The console is the text-output area that we have already seen when using **`cout`** to display information. In addition to displaying text, the console can also solicit text from a user.

- The **`getLine()`** function takes in a single parameter, which is a prompt to show to the user.

The console and the `getLine()` function

- The console is the text-output area that we have already seen when using **`cout`** to display information. In addition to displaying text, the console can also solicit text from a user.

- The **`getLine()`** function takes in a single parameter, which is a prompt to show to the user.

- The function will then wait while the user types in text into the console.

The console and the `getLine()` function

- The console is the text-output area that we have already seen when using **`cout`** to display information. In addition to displaying text, the console can also solicit text from a user.

- The **`getLine()`** function takes in a single parameter, which is a prompt to show to the user.

- The function will then wait while the user types in text into the console.

- After the user submits their answer by hitting the “Enter/Return” key, the function returns the value that the user typed into the console.

Console Programs Demo

Console program summary

- Use **`getLine(prompt)`** to read in information from the user.
  - Make sure to convert the data to the correct type
  - You can also use functions from simpio.h to get data of other types

Console program summary

- Use **`getLine(prompt)`** to read in information from the user.
  - Make sure to convert the data to the correct type
  - You can also use functions from simpio.h to get data of other types

- Use a **`while`** loop to enable multiple runs of your program.
  - **`while(true)`** paired with **`break`** is a powerful construct

Console program summary

- Use **`getLine(prompt)`** to read in information from the user.
  - Make sure to convert the data to the correct type
  - You can also use functions from simpio.h to get data of other types

- Use a **`while`** loop to enable multiple runs of your program.
  - **`while(true)`** paired with **`break`** is a powerful construct

- Console programs must be run directly from **`main()`**
  - Doesn't make sense to write tests using SimpleTest because they don't have neatly defined "output" to compare against

Announcements

Announcements

- Sections started yesterday and are continuing for the rest of the week! Check cs198.stanford.edu to see your time.
  - Section attendance and engaged participation are a part of your grade, so make sure to attend!
- Assignment 1 is out and is due next Tuesday at 11:59pm in your local timezone.
  - The YEAH session recording from last night is posted on Canvas under the "Course Videos" tab (different from where lectures are).
- Ed workspace notes
  - If you had technical difficulties during yesterday's example, check out the last 5 minutes of the lecture recording for a summary of the activity.
  - For now, you cannot fork public workspaces, but you can download the contents for later use.
- C++ survey results
  - We'll be making a post tomorrow on Ed addressing common questions that came up in the C++ survey. Keep an eye out for that so that you can get your questions answered!

How do we structure data using abstractions in code?

Abstract Data Types

Abstract Data Types

- Data structures, or **abstract data types (ADTs)**, are powerful abstractions that allow programmers to store data in structured, organized ways

Abstract Data Types

- Data structures, or **abstract data types (ADTs)**, are powerful abstractions that allow programmers to store data in structured, organized ways

- These ADTs give us certain guarantees about the organization and properties of our data, without our having to worry about managing the underlying details

Abstract Data Types

- Data structures, or **abstract data types (ADTs)**, are powerful abstractions that allow programmers to store data in structured, organized ways

- These ADTs give us certain guarantees about the organization and properties of our data, without our having to worry about managing the underlying details

An abstraction for the programmer!

Abstract Data Types

- Data structures, or **abstract data types (ADTs)**, are powerful abstractions that allow programmers to store data in structured, organized ways

- These ADTs give us certain guarantees about the organization and properties of our data, without our having to worry about managing the underlying details

- While we specifically study implementations of ADTs from the Stanford C++ libraries, these principles transcend language boundaries

Abstract Data Types

- Data structures, or **abstract data types (ADTs)**, are powerful abstractions that allow programmers to store data in structured, organized ways

- These ADTs give us certain guarantees about the organization and properties of our data, without our having to worry about managing the underlying details

- While we specifically study implementations of ADTs from the Stanford C++ libraries, these principles transcend language boundaries
    - We will do our best to point out comparisons to Java and Python along the way.
    - We will not be learning how to use the standard C++ (STL) data structures. If you're interested in learning more about these, check out the CS106L course materials.

Vectors

What is a vector?

- At a high level, a vector is an ordered collection of elements of the same type that can grow and shrink in size.

What is a vector?

- At a high level, a vector is an **ordered** collection of elements of the same type that can grow and shrink in size.
  - Each element in the collection has a specific location, or **index**

What is a vector?

- At a high level, a vector is an ordered collection of **elements of the same type** that can grow and shrink in size.
  - Each element in the collection has a specific location, or index
  - All elements in a vector must be of the same type. Unlike in other programming languages, a single vector cannot contain elements of mixed types.

What is a vector?

- At a high level, a vector is an ordered collection of elements of the same type that can **grow and shrink in size**.
  - Each element in the collection has a specific location, or index
  - All elements in a vector must be of the same type. Unlike in other programming languages, a single vector cannot contain elements of mixed types.
  - Vectors are flexible when it comes to the number of elements they can store. You can easily add and remove elements from a vector. Vectors also know their size, meaning you can query them to see how many elements they currently contain.

What is a vector?

- At a high level, a vector is an ordered collection of elements of the same type that can grow and shrink in size.
  - Each element in the collection has a specific location, or index
  - All elements in a vector must be of the same type. Unlike in other programming languages, a single vector cannot contain elements of mixed types.
  - Vectors are flexible when it comes to the number of elements they can store. You can easily add and remove elements from a vector. Vectors also know their size, meaning you can query them to see how many elements they currently contain.
- Analogs in other languages: **`list`** in Python and **`ArrayList`** in Java

What is a vector?

- At a high level, a vector is an ordered collection of elements of the same type that can grow and shrink in size.
  - Each element in the collection has a specific location, or index
  - All elements in a vector must be of the same type. Unlike in other programming languages, a single vector cannot contain elements of mixed types.
  - Vectors are flexible when it comes to the number of elements they can store. You can easily add and remove elements from a vector. Vectors also know their size, meaning you can query them to see how many elements they currently contain.
- Analogs in other languages: **`list`** in Python and **`ArrayList`** in Java
- Defined in the **`"vector.h"`** header file of the Stanford C++ libraries

What is a vector?

- At a high level, a vector is an ordered collection of elements of the same type that can grow and shrink in size.
  - Each element in the collection has a specific location, or index
  - All elements in a vector must be of the same type. Unlike in other programming languages, a single vector cannot contain elements of mixed types.
  - Vectors are flexible when it comes to the number of elements they can store. You can easily add and remove elements from a vector. Vectors also know their size, meaning you can query them to see how many elements they currently contain.
- Analogs in other languages: **`list`** in Python and **`ArrayList`** in Java
- Defined in the **`"vector.h"`** **header file** of the Stanford C++ libraries

A collection of function prototypes that allows for code sharing and reuse.

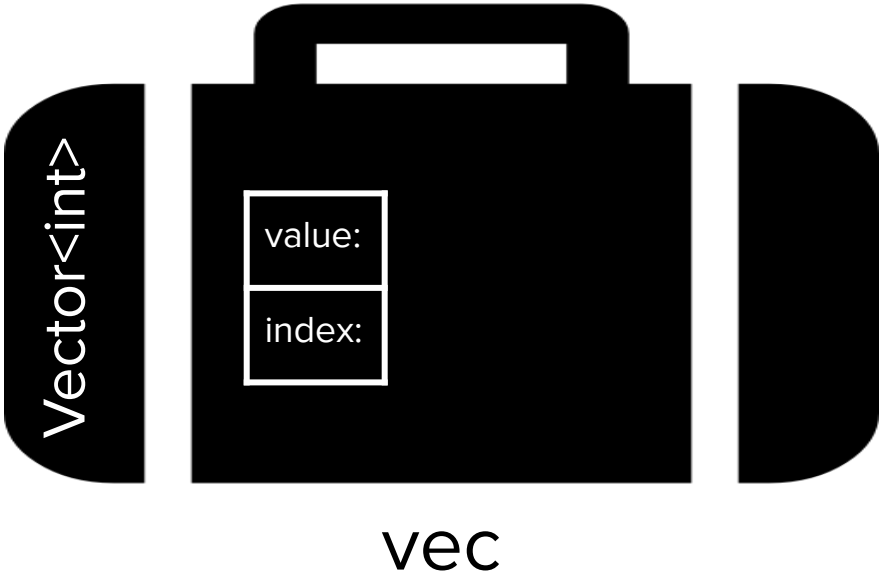
Basic Vector Operations: Creation

Basic Vector Operations: Creation

```
Vector<int> vec;
```

Basic Vector Operations: Creation

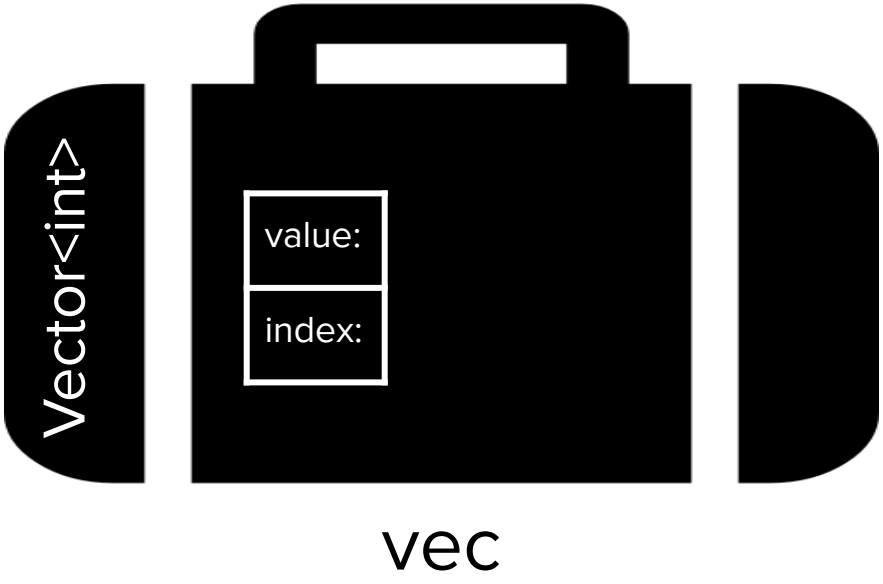
```
Vector<int> vec;
```



Basic Vector Operations: Creation

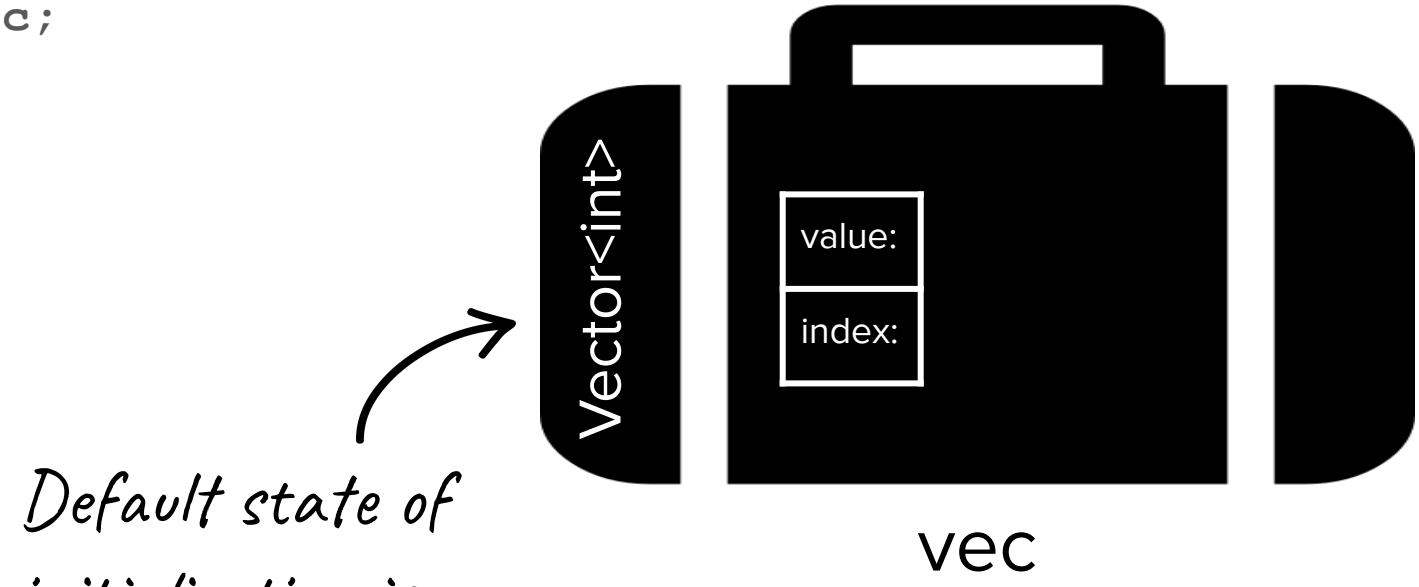
```
Vector<int> vec;
```

Must specify the type of values that will be held at creation time.



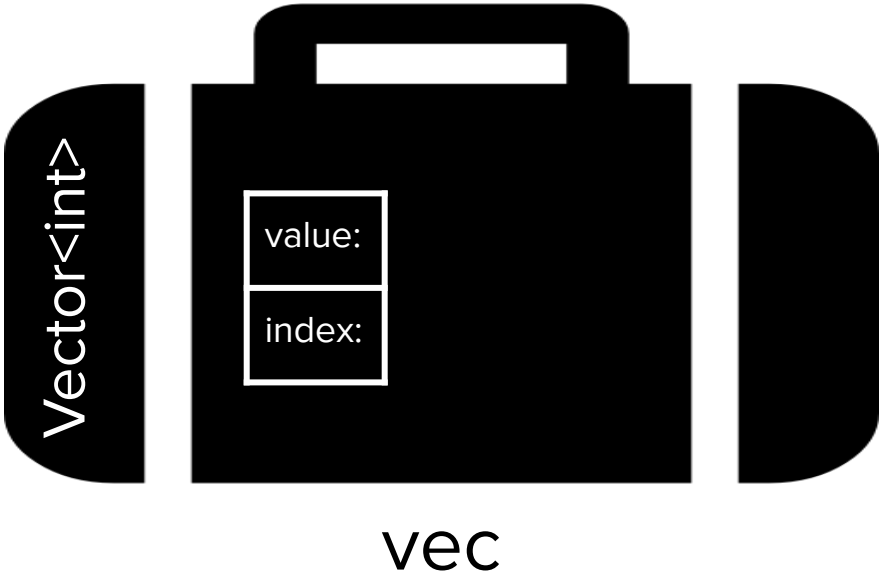
Basic Vector Operations: Creation

```
Vector<int> vec;
```



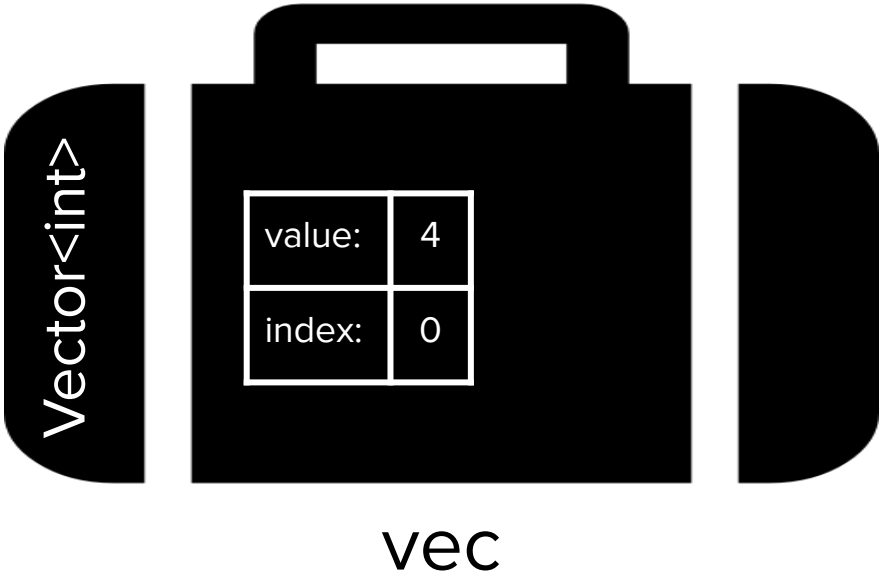
Basic Vector Operations: Adding Elements

```
Vector<int> vec;

vec.add(4);
```



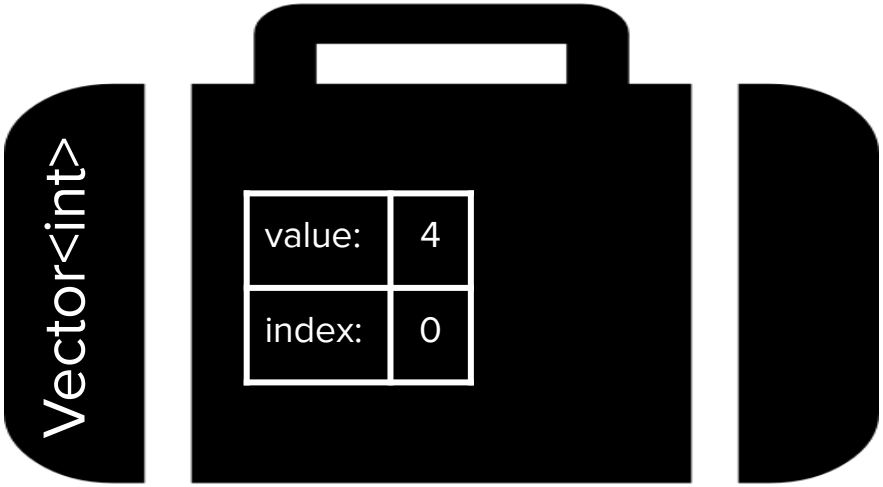
Basic Vector Operations: Adding Elements

```
Vector<int> vec;

vec.add(4);
```



Basic Vector Operations: Adding Elements

```
Vector<int> vec;

vec.add(4);
```

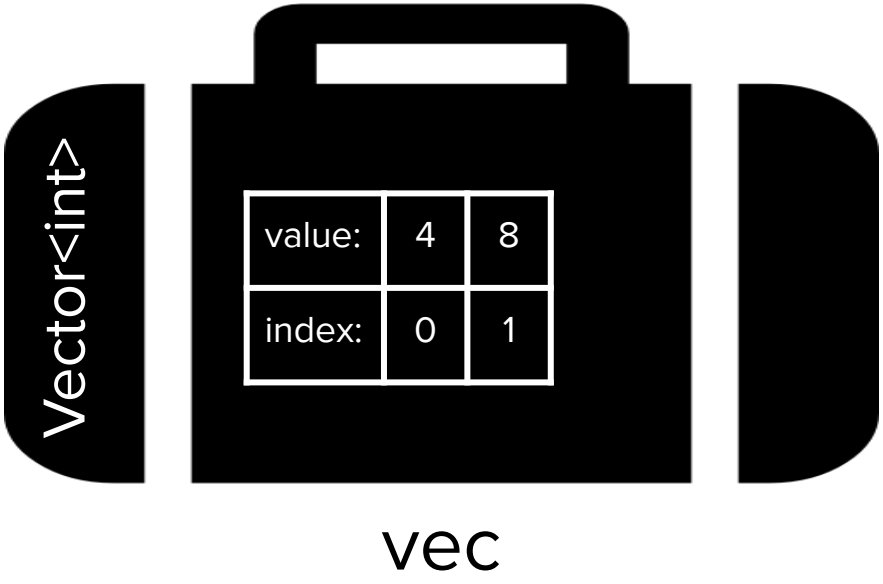


vec

Note: indexing starts at 0

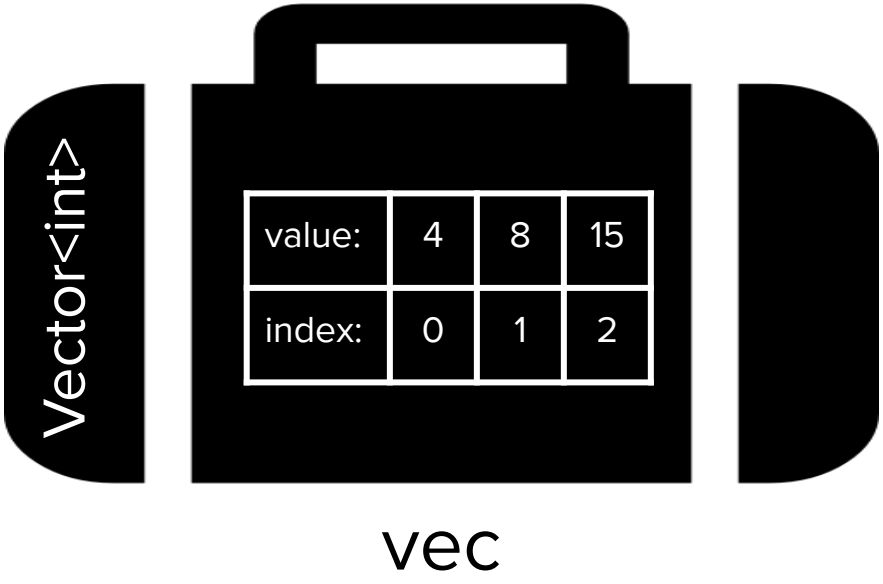
Basic Vector Operations: Adding Elements

```
Vector<int> vec;

vec.add(4);

vec.add(8);
```



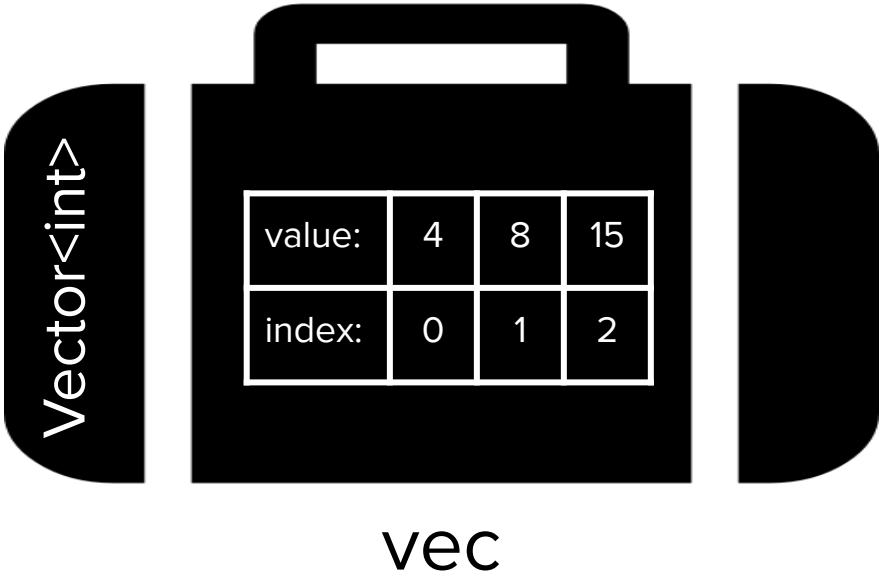
Basic Vector Operations: Adding Elements

```
Vector<int> vec;

vec.add(4);

vec.add(8);

vec.add(15);
```



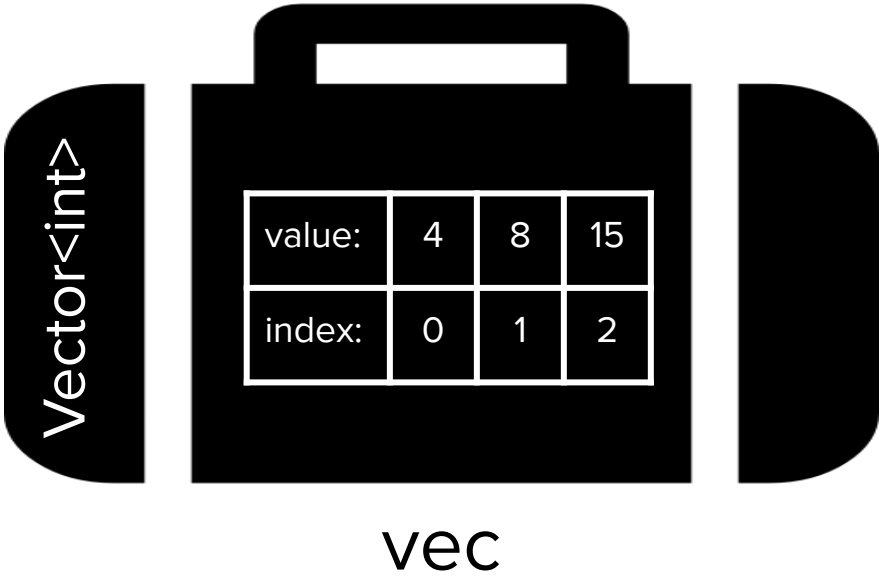
Basic Vector Operations: Creating + Adding Together

```
Vector<int> vec = {4, 8, 15};
```



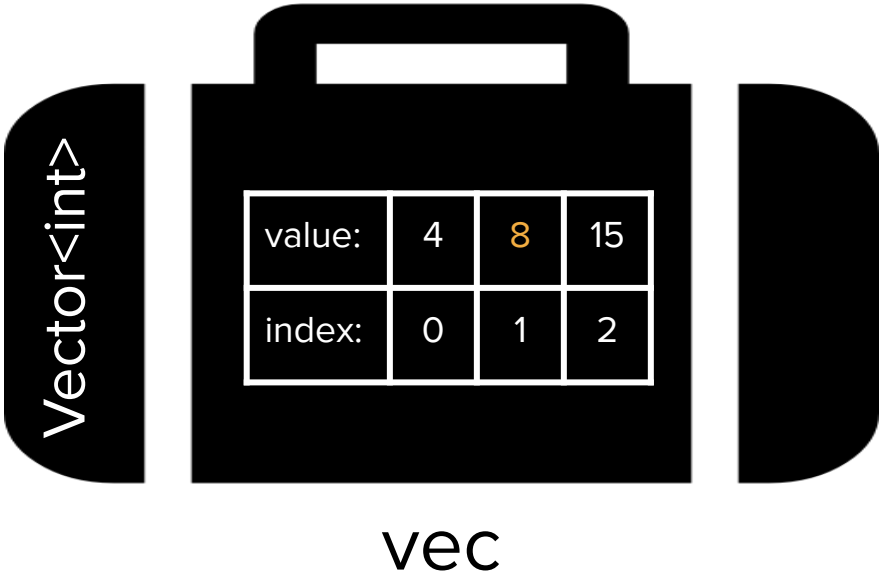
Basic Vector Operations: Accessing Elements

```
Vector<int> vec = {4, 8, 15};

cout << vec[1] << endl;
```



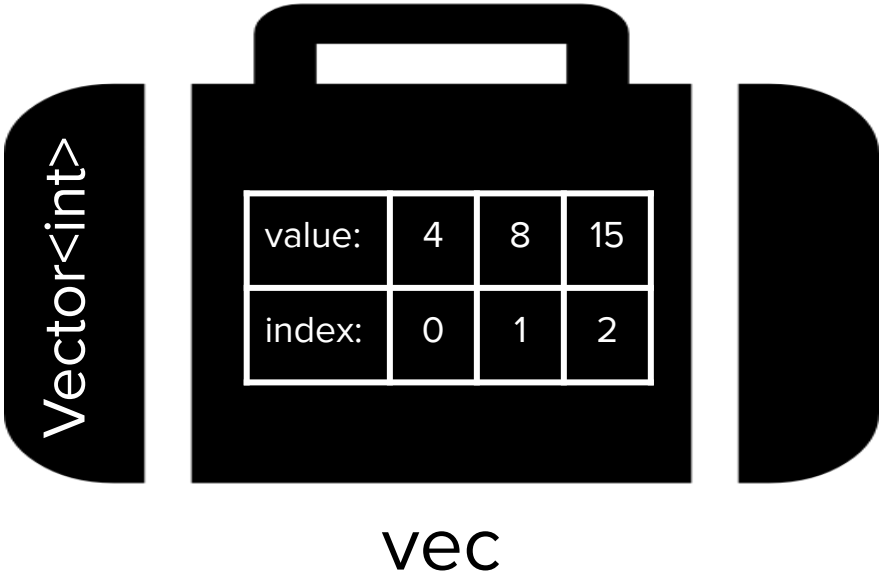
Basic Vector Operations: Accessing Elements

```
Vector<int> vec = {4, 8, 15};

cout << vec[1] << endl;
```



Basic Vector Operations: Accessing Elements

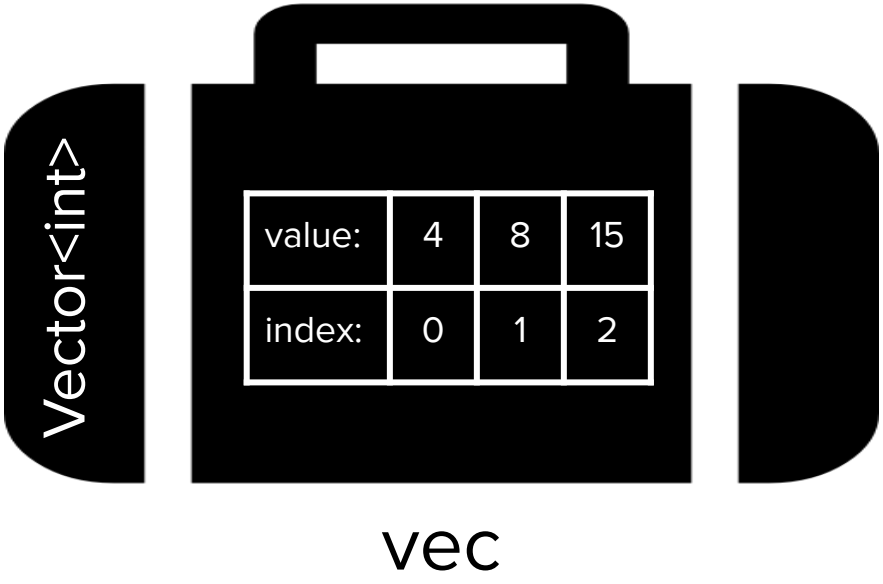
```
Vector<int> vec = {4, 8, 15};

cout << vec[3] << endl;
```



Basic Vector Operations: Accessing Elements

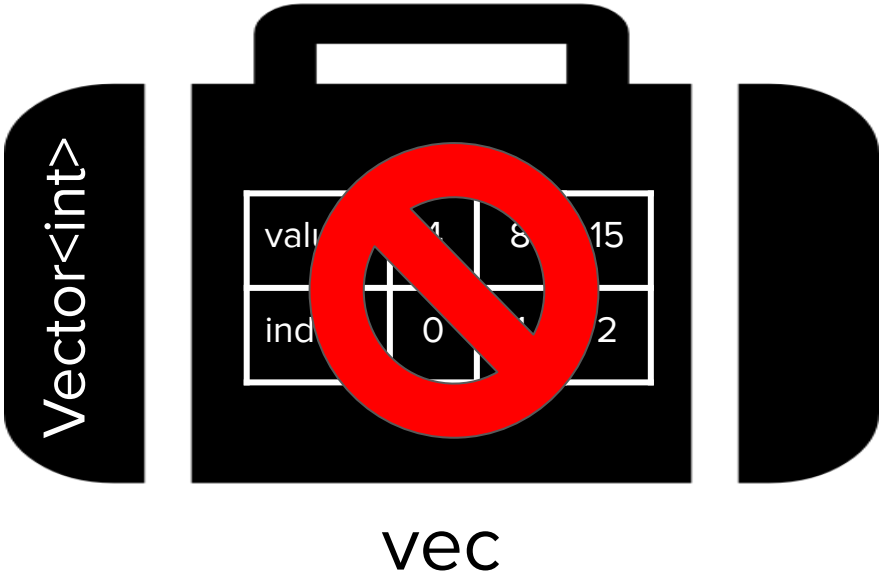
```
Vector<int> vec = {4, 8, 15};

cout << vec[3] << endl;
```

Poll: What will be the output of the above code snippet?



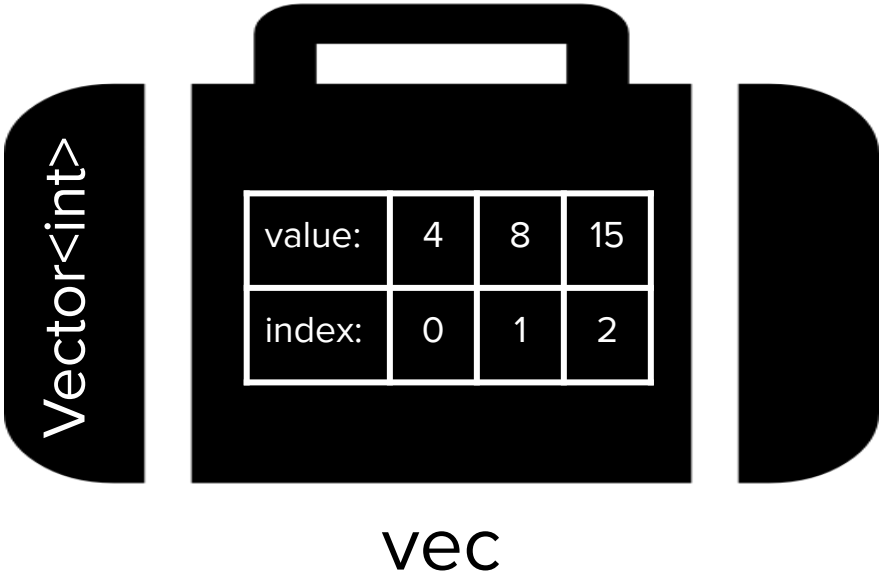
Basic Vector Operations: Accessing Elements

```
Vector<int> vec = {4, 8, 15};

cout << vec[3] << endl;

// this will throw an error!

// takeaway: Vector does
bounds checking and will not
allow you to access elements
that are out of bounds
```



Basic Vector Operations: Removing Elements

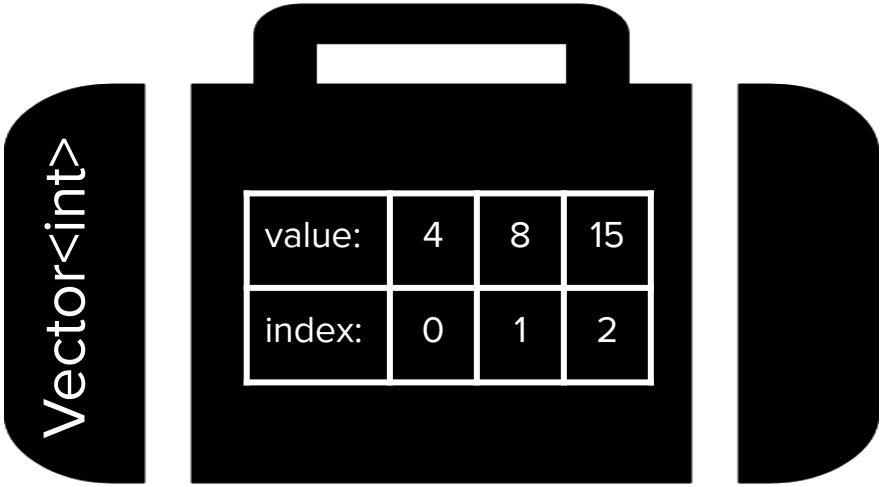
```
Vector<int> vec = {4, 8, 15};

cout << vec[1] << endl;

vec.remove(0);
```



Basic Vector Operations: Removing Elements

```
Vector<int> vec = {4, 8, 15};

cout << vec[1] << endl;

vec.remove(0);
```

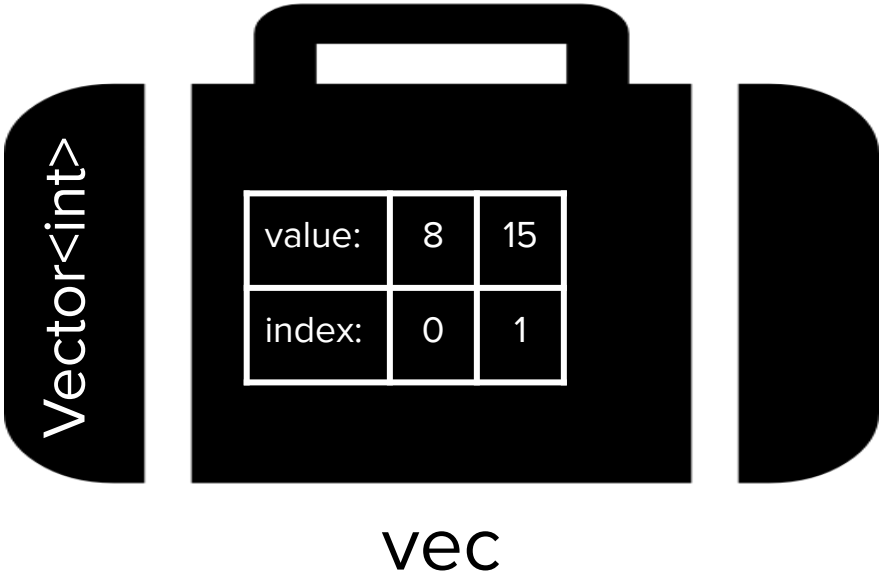
Specify the index to remove at



vec

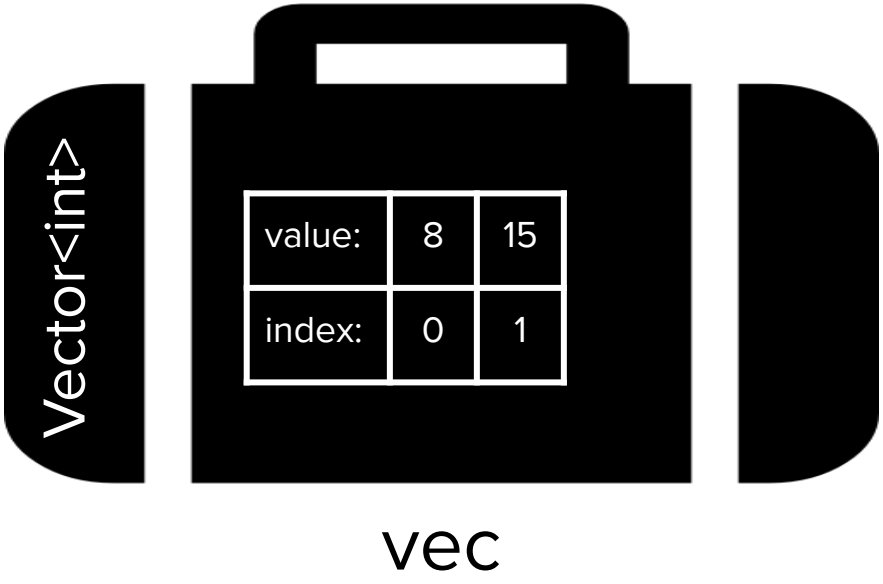
Basic Vector Operations: Removing Elements

```
Vector<int> vec = {4, 8, 15};

cout << vec[1] << endl;

vec.remove(0);
```



Basic Vector Operations: Number of Elements

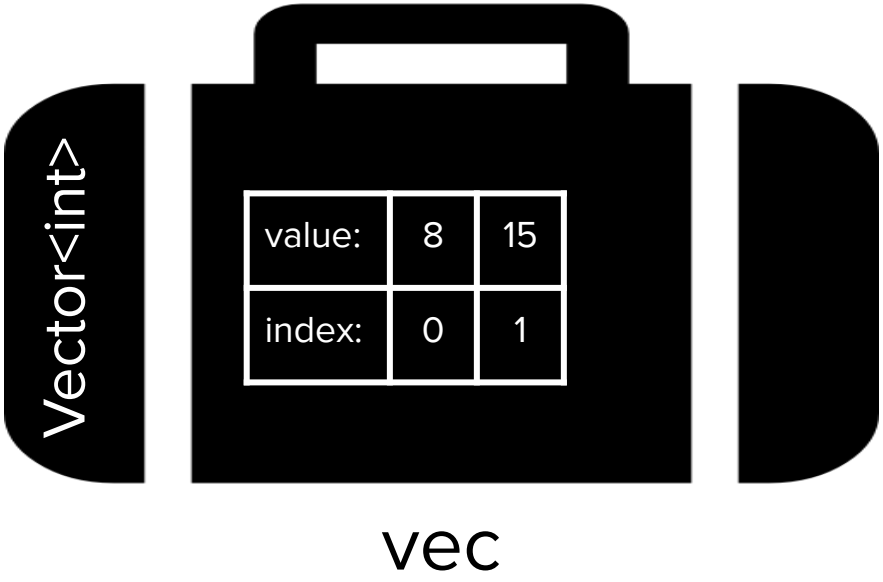
```
Vector<int> vec = {4, 8, 15};

cout << vec[1] << endl;

vec.remove(0);

cout << vec.size() << endl;
```



Basic Vector Operations: Number of Elements

```
Vector<int> vec = {4, 8, 15};

cout << vec[1] << endl;

vec.remove(0);

cout << vec.size() << endl;
```

Output:
2



Traversing a Vector

- Method 1: Traditional for loop

```
Vector<int> vec = {1, 0, 6};
for (int i = 0; i < vec.size(); i++) {
    cout << vec[i] << endl;
}
```

Traversing a Vector

- Method 1: Traditional for loop

```
Vector<int> vec = {1, 0, 6};
for (int i = 0; i < vec.size(); i++) {
    cout << vec[i] << endl;
}
```

Output:

1

0

6

Traversing a Vector

- Method 1: Traditional for loop

```
Vector<int> vec = {1, 0, 6};
for (int i = 0; i < vec.size(); i++) {
    cout << vec[i] << endl;
}
```

- Method 2: for-each loop

```
Vector<int> vec = {1, 0, 6};
for (int num: vec) {
    cout << num << endl;
}
```

Output:

1

0

6

Vector Functions

```
#include "vector.h"
```

- The following functions are part of the Vector collection, and can be useful:
  - **`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 the exhaustive list, check out the Stanford Vector class documentation

Vector Functions

```
#include "vector.h"
```

- The following functions are part of the Vector collection, and can be useful:
  - `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 the exhaustive list, check out the Stanford Vector class documentation

A vector example

[demo + poll]

Eliminating Negativity

- Consider the following task: Given a Vector of integers, write a function that eliminates negativity from the vector by changing the sign of all negative values to turn them into their positive equivalents

Eliminating Negativity

- Consider the following task: Given a Vector of integers, write a function that eliminates negativity from the vector by changing the sign of all negative values to turn them into their positive equivalents
- **Poll: What is the output of the code snippet?**

```
void eliminateNegativity(Vector<int> v){
    for (int i = 0; i < v.size(); i++){
        if (v[i] < 0){
            v[i] = -1 * v[i];
        }
    }
}

int main(){
    Vector<int> nums = {1, -4, 18, -11};
    eliminateNegativity(nums);
    cout << nums << endl;
}
```

Eliminating Negativity

- Consider the following task: Given a Vector of integers, write a function that eliminates negativity from the vector by changing the sign of all negative values to turn them into their positive equivalents
- Result: The vector is passed by value, so a copy is modified, and no changes persist.

```
void eliminateNegativity(Vector<int> v){
    for (int i = 0; i < v.size(); i++){
        if (v[i] < 0){
            v[i] = -1 * v[i];
        }
    }
}

int main(){
    Vector<int> nums = {1, -4, 18, -11};
    eliminateNegativity(nums);
    cout << nums << endl;
}
```

Eliminating Negativity

- Consider the following task: Given a Vector of integers, write a function that eliminates nega
vector by changing
negative values to t
their positive equiva
- Result: The vector is
value, so a copy is n
changes persist.

```
void eliminateNegativity(Vector<int> v){
    for (int i = 0; i < v.size(); i++){
        if (v[i] < 0){
            [i] = -1 * v[i];
```

```
t> nums = {1, -4, 18, -11};
Negativity(nums);
ums << endl;
```

So how do we allow functions to modify vectors?

Pass by reference

(i.e. How do we efficiently and effectively handle data structures in functions?)

Definition

pass by value

When a parameter is passed into a function, the new variable *stores a copy* of the passed in value in memory

Definition

pass by reference

When a parameter is passed into a function, the new variable stores a *reference* to the passed in value, which allows you to directly edit the original value

What exactly is a reference?

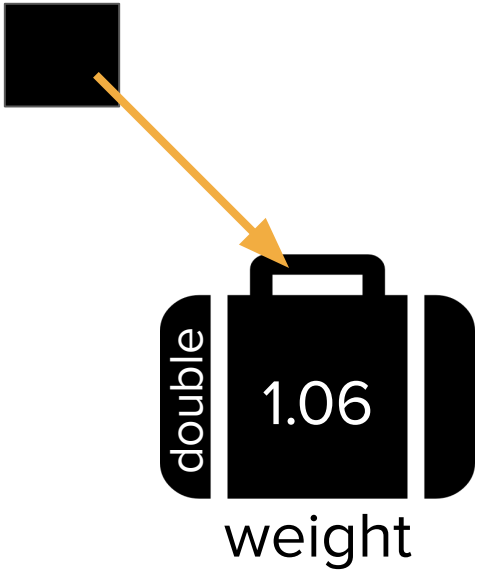
- Regular variables look like this:

We will think of a variable as a named container storing a value.

What exactly is a reference?

- References look like this:

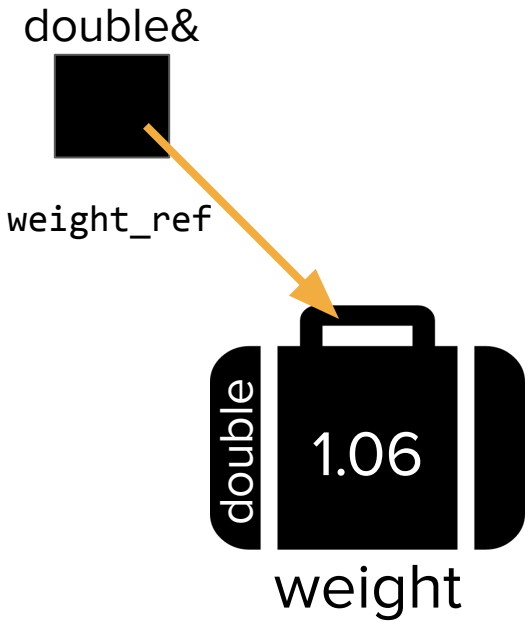
We will think of a reference as a box that just refers to an existing variable.



What exactly is a reference?

- References look like this:

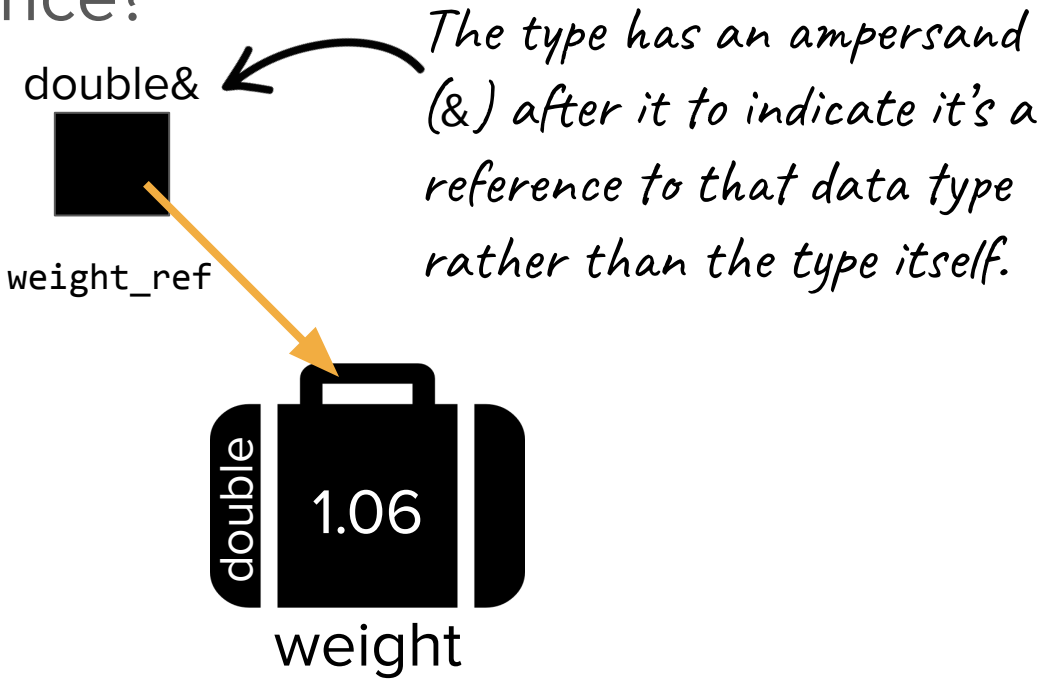
References have names and types, just like regular variables.



What exactly is a reference?

- References look like this:

References have names and types, just like regular variables.

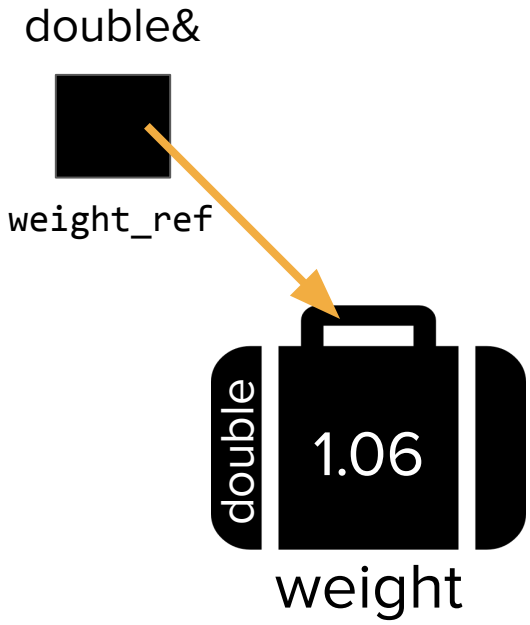


What exactly is a reference?

- References look like this:

Here's what this would look like in code:

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

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

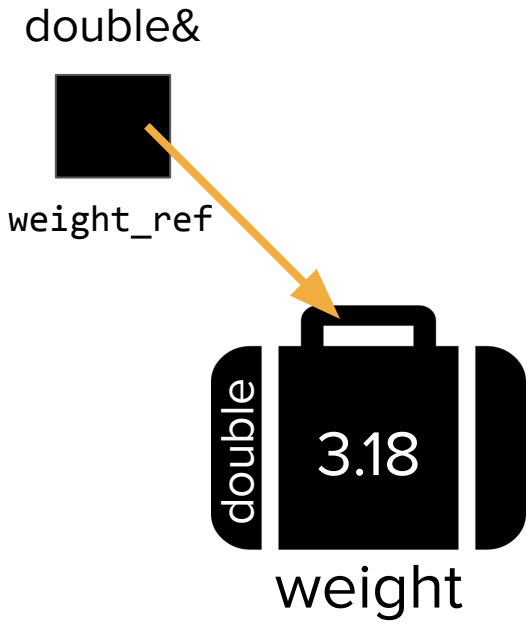


What exactly is a reference?

- References look like this:

Here's what this would look like in code:

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

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

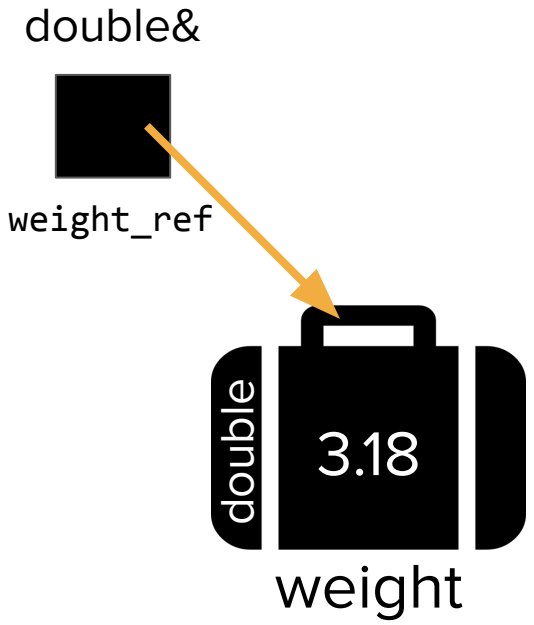


What exactly is a reference?

- References look like this:

Here's what this would look like in code:

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

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



But we don't usually write code this way...

When we use references

- To allow helper functions to edit data structures in other functions

When we use references

- To allow helper functions to edit data structures in other functions
  - But why don’t we just return a copy of the data structure?

When we use references

- To allow helper functions to edit data structures in other functions
  - But why don't we just return a copy of the data structure?

- To avoid making new copies of large data structures in memory
  - Passing data structures by reference makes your code more efficient!

When we use references

- To allow helper functions to edit data structures in other functions
  - But why don't we just return a copy of the data structure?
- To avoid making new copies of large data structures in memory
  - Passing data structures by reference makes your code more efficient!
- References also provide a workaround for **multiple return values**
  - Your function can both have a return value and also directly edit a Vector object passed in as a parameter. This makes it as if your function is returning both the vector and the actual return value!

When we use references

- To allow helper functions to edit data structures in other functions
  - But why don’t we just return a copy of the data structure?
- To avoid making new copies of large data structures in memory
  - Passing data structures by reference makes your code more efficient!
- References also provide a workaround for **multiple return values**
  - Your function can take in multiple pieces of information by reference and modify them all. In this way you can "return" both a modified Vector and some auxiliary piece of information about how the structure was modified. This makes it as if your function is returning two updated pieces of information to the function that called it!

Revisiting eliminateNegativity

[demo]

When we *don't* use references

- If we always used references, functions would all be able to edit one another's variables, and scoping would get confusing!
  - This would also make bugs much more likely. Unexpected and unintended changes to variables could persist across functions.

When we *don’t* use references

- If we always used references, functions would all be able to edit one another’s variables, and scoping would get confusing!
  - This would also make bugs much more likely. Unexpected and unintended changes to variables could persist across functions.

- When the data itself is small (i.e. the cost of *copying by value* is low), then we don’t need to use a reference.

When we *don't* use references

- If we always used references, functions would all be able to edit one another's variables, and scoping would get confusing!
  - This would also make bugs much more likely. Unexpected and unintended changes to variables could persist across functions.

- When the data itself is small (i.e. the cost of *copying by value* is low), then we don't need to use a reference.

- Note: You can't provide a literal as an argument if you are passing a parameter by reference.

```
void tripleWeight(double& weight_ref);
...
tripleWeight(1.06);
```

Don't do this!
Compiler error!

When we *don’t* use references

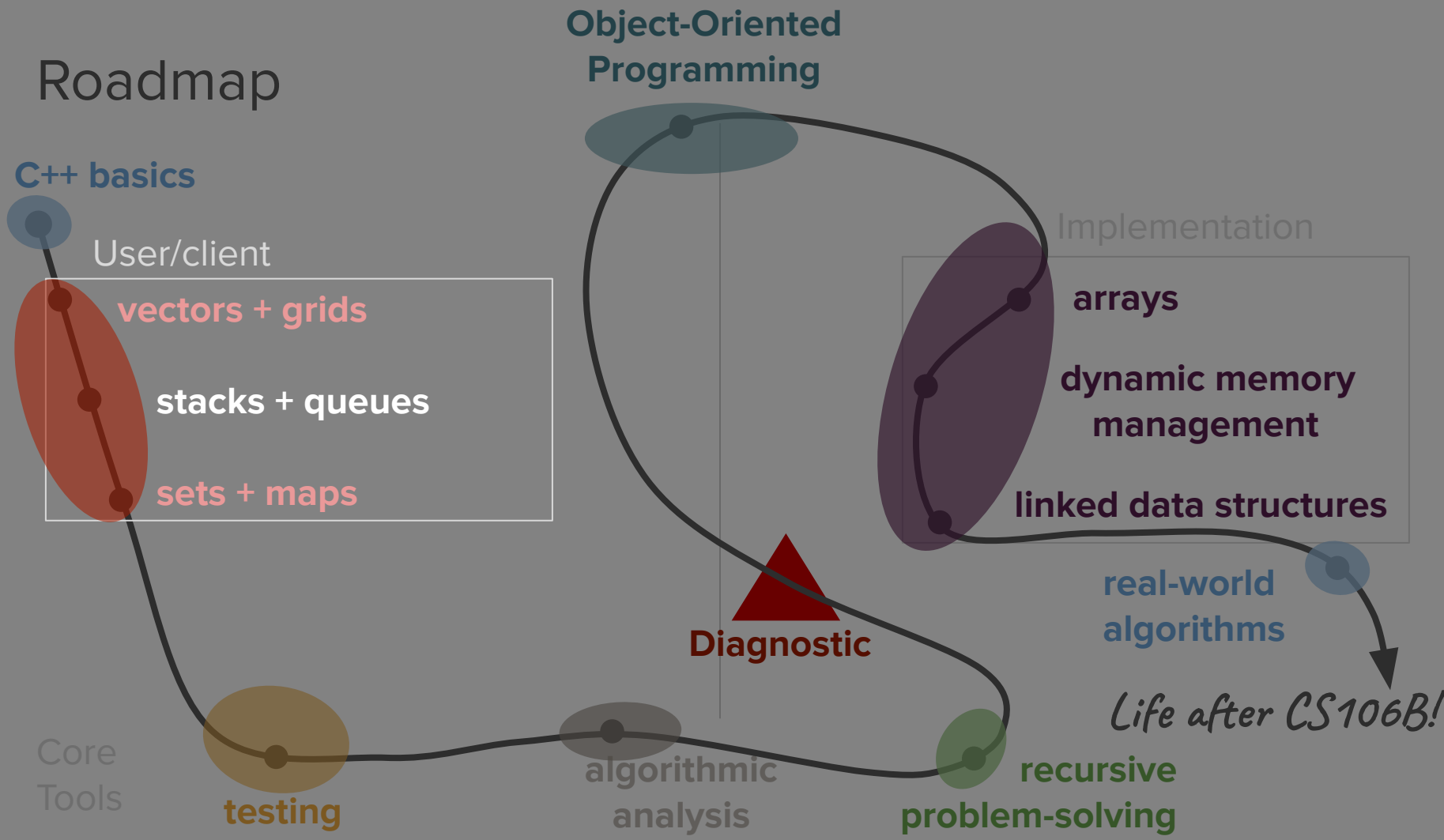
- If we always used references, functions would all be able to edit one another’s variables, and scoping would get confusing!
  - This would also make bugs much more likely. Unexpected and unintended changes to variables could persist across functions.

- When the data itself is small (i.e. the cost of *copying by value* is low), then we don’t need to use a reference.

- Note: You can't provide a literal as an argument if you are passing a parameter by reference.

What’s next?

Roadmap


Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

Stacks and Queues