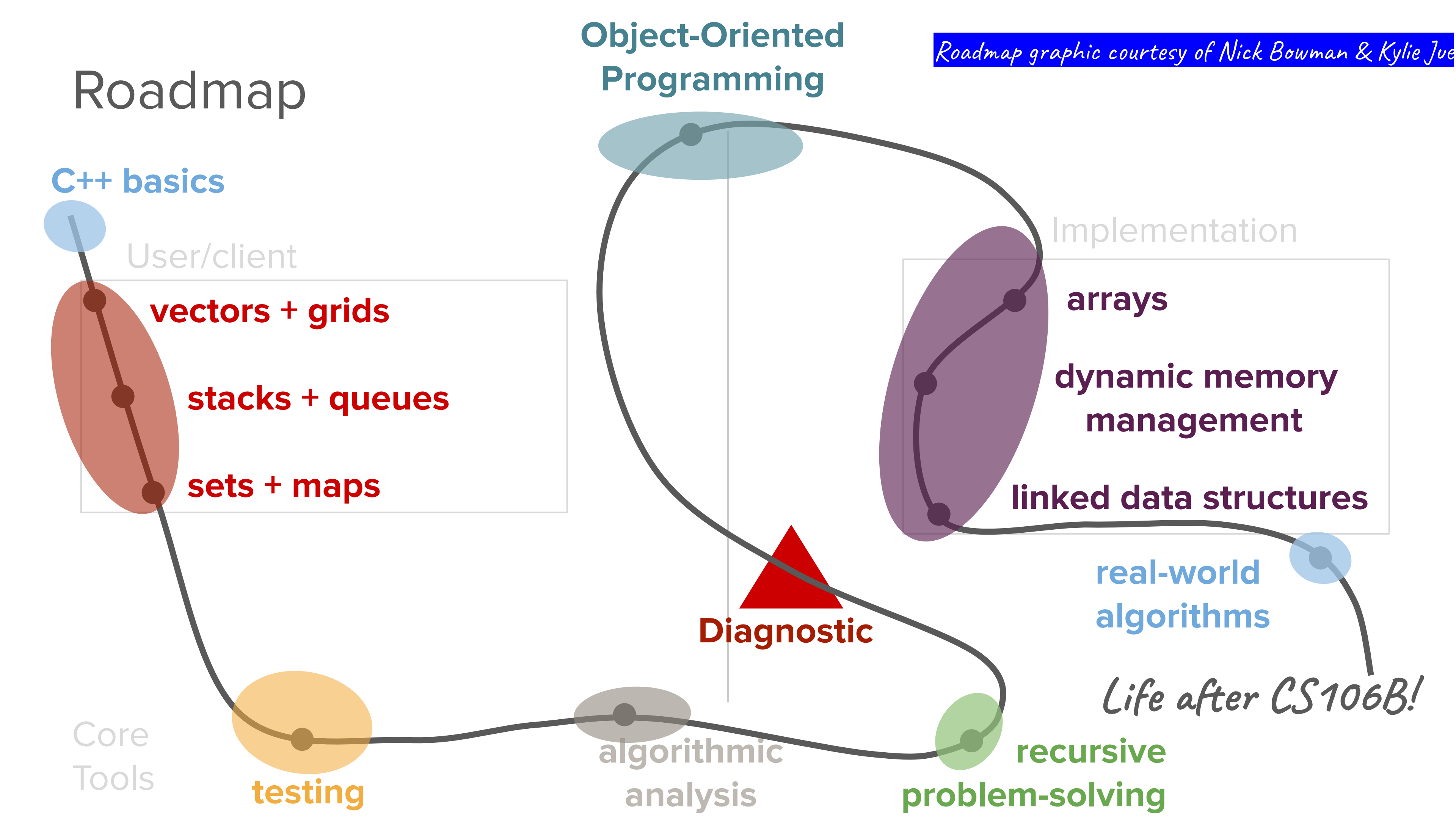
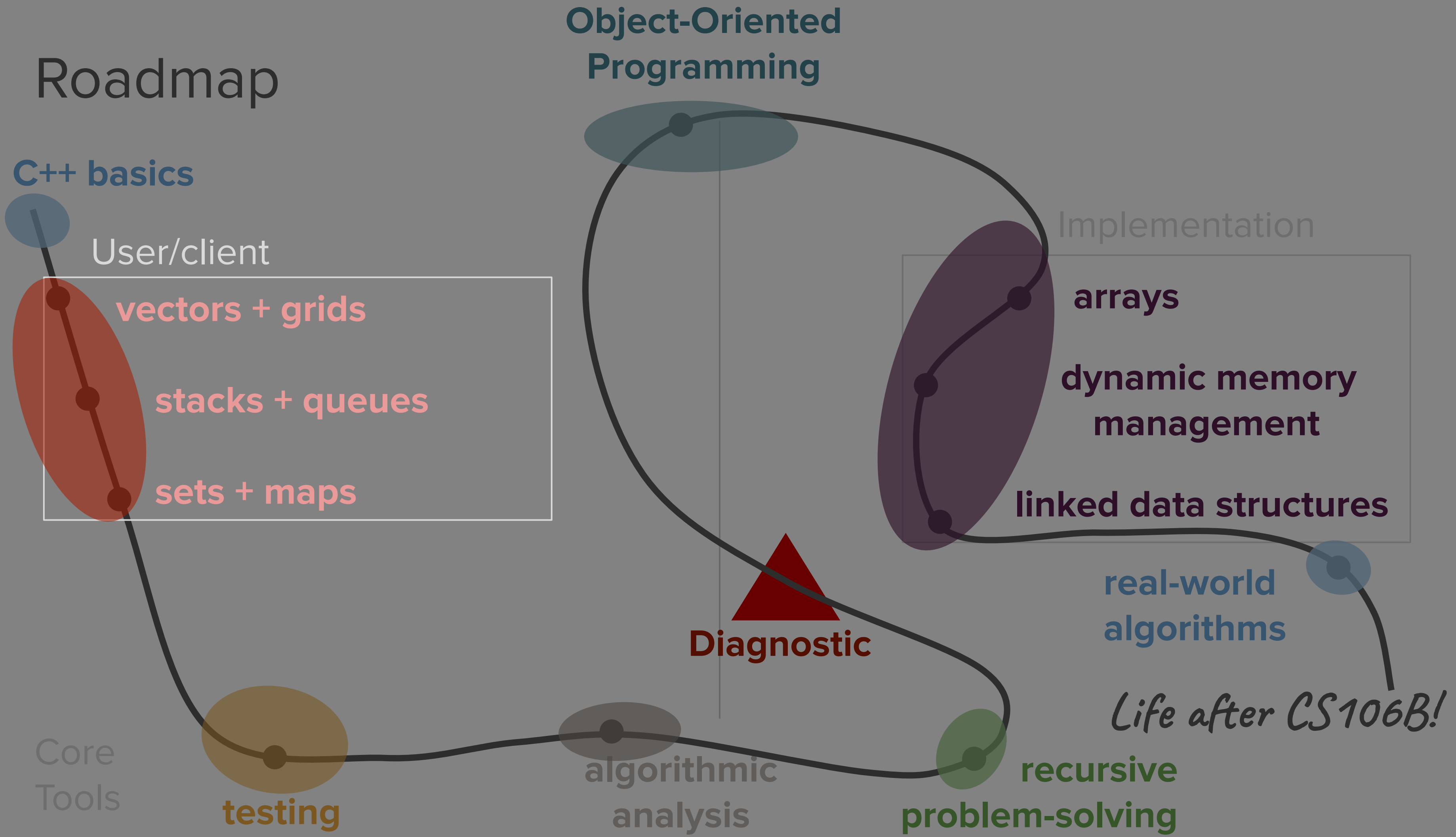
Ordered Data Structures: Grids, Queues, and Stacks

What's an example of "ordered data" that you've encountered in your life?

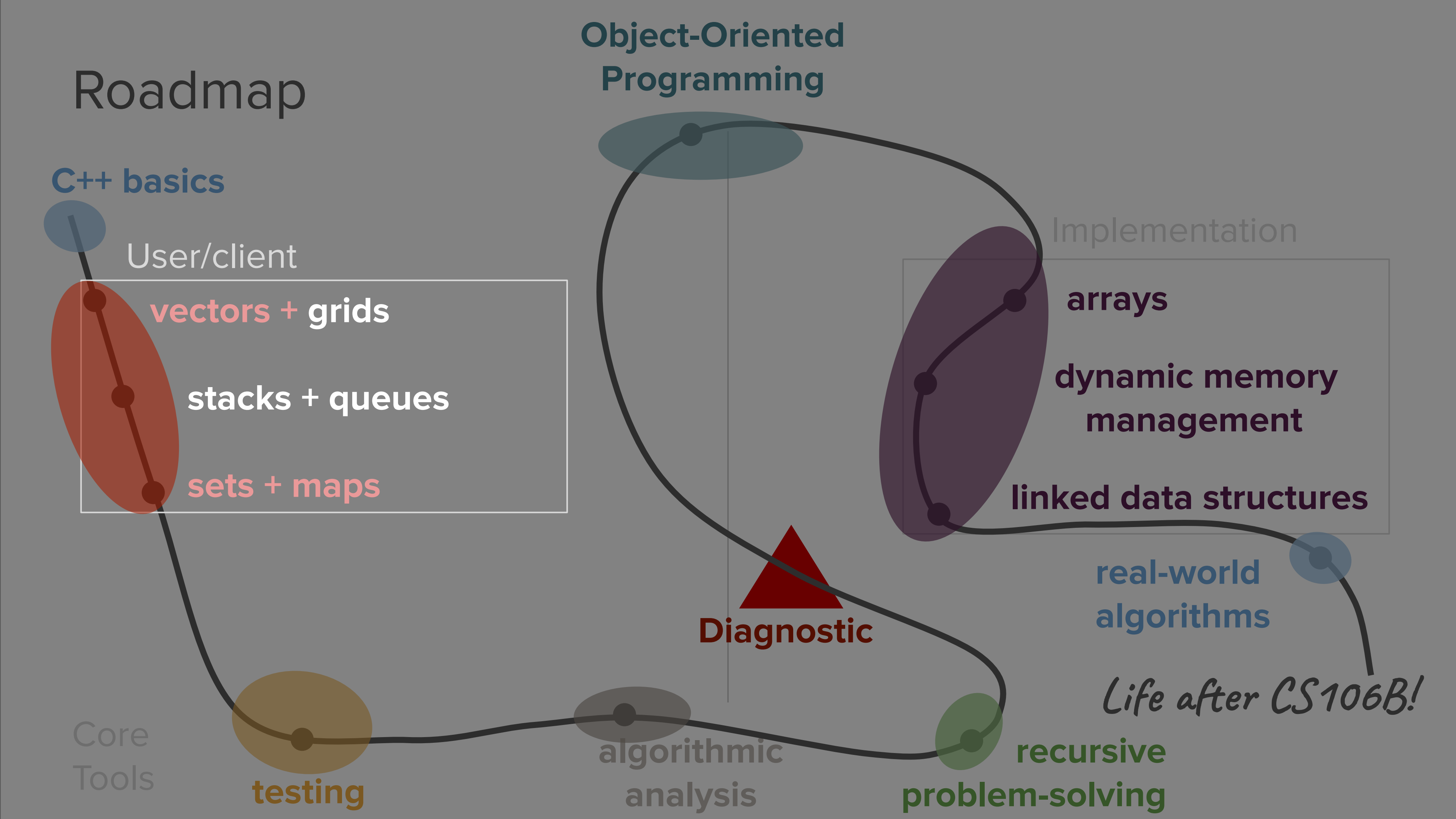
(put your answers in the chat)


Roadmap
Roadmap graphic courtesy of Nick Bowman & Kylie Jue
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
Implementation
vectors + grids
stacks + queues
sets + maps
arrays
dynamic memory management
linked data structures
real-world algorithms
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

Today's question

When is it appropriate to use different types of ordered data structures?

Today's topics

1. Review

2. Grids

[2.5 **`GridLocation`** + structs]

1. Queues

2. Stacks

Review

(vectors)

Abstract Data Types

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

- These ADTs give us certain guarantees about the organization and properties of our data

Abstract Data Types

*ADTs provide **interfaces** for data storage that programmers can use without understanding the underlying implementation!*

Our first ADT: **Vectors**

- 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.
- Vectors are flexible about the number of elements they can store. You can easily add and remove elements, and vectors also know their current size.

Grids

What is a Grid?

- A 2D array, defined with a particular width and height

0	0	0
0	0	0
0	0	0

What is a Grid?

- A 2D array, defined with a particular width and height

We say array instead of vector here because the dimensions are established when the Grid is created.

0	0	0
0	0	0
0	0	0

What is a Grid?

- A 2D array, defined with a particular width and height
- Useful for spreadsheets, game boards, graphics, cartography, geology, and more!

0	0	0
0	0	0
0	0	0

What is a Grid?

- A 2D array, defined with a particular width and height
- Useful for spreadsheets, game boards, graphics, cartography, geology, and more!
- Three ways to declare a Grid
  - Grid<type> gridName;
  - Grid<type> gridName(numRows, numCols);
  - Grid<type> gridName = {{r0c0, r0c1, r0c2}, {r1c0, r1c1, r1c2},...};

False	False	False
False	False	False
False	False	False

What is a Grid?

- A 2D array, defined with a particular width and height
- Useful for spreadsheets, game boards, graphics, cartography, geology, and more!
- Three ways to declare a Grid
  - Grid<type> gridName;

```
Grid<int> board;
board.resize(3, 3);
board[0][0] = 2;
board[1][0] = 6;
```

0	0	0
0	0	0
0	0	0

What is a Grid?

- A 2D array, defined with a particular width and height
- Useful for spreadsheets, game boards, graphics, cartography, geology, and more!
- Three ways to declare a Grid
    - Grid<type> gridName;

```
Grid<int> board;
board.resize(3, 3);
board[0][0] = 2;
board[1][0] = 6;
```

0	0	0
0	0	0
0	0	0

If you declare a board with no initialization, you must resize it or reassign it before using it. Resizing will fill it with default values for that type.

What is a Grid?

- A 2D array, defined with a particular width and height
- Useful for spreadsheets, game boards, graphics, cartography, geology, and more!
- Three ways to declare a Grid
  - Grid<type> gridName(numRows, numCols);

```
Grid<int> board(3, 3);
board[0][0] = 2;
board[1][0] = 6;
```

2	0	0
6	0	0
0	0	0

What is a Grid?

- A 2D array, defined with a particular width and height
- Useful for spreadsheets, game boards, graphics, cartography, geology, and more!
- Three ways to declare a Grid
  - Grid<type> gridName = {{r0c0, r0c1, r0c2}, {r1c0, r1c1, r1c2},...};

```
Grid<int> board = {{2}, {6}};
```

2	0	0
6	0	0
0	0	0

What is a Grid?

- A 2D array, defined with a particular width and height
- Useful for spreadsheets, game boards, graphics, cartography, geology, and more!
- Three ways to declare a Grid
  - Grid<type> gridName;
  - Grid<type> gridName(numRows, numCols);
  - Grid<type> gridName = {{r0c0, r0c1, r0c2}, {r1c0, r1c1, r1c2},...};

2	0	0
6	0	0
0	0	0

Grid methods

- The following methods are part of the Grid collection and can be useful:
    - **`gname.numRows()`**: Returns the number of rows in the Grid.
    - **`gname.numCols()`**: Returns the number of columns in the Grid.
    - **`gname[i][j]`**: selects the element in the **`i`**th row and **`j`**th column.
    - **`gname.resize(rows, cols)`**: Changes the dimensions of the Grid and re-initializes all entries to their default values.
    - **`gname.inBounds(row, col)`**: Returns **`true`** if the specified row, column position is in the Grid, **`false`** otherwise.

- For the exhaustive list, check out the Stanford Grid documentation.

Grid methods

- The following methods are part of the Grid collection and can be useful:
  - `gname.numRows()`: Returns the number of rows in the Grid.
  - `gname.numCols()`: Returns the number of columns in the Grid.
  - `gname[i][j]`: selects the element in the `i`th row and `j`th column.
  - `gname.resize(rows, cols)`: Changes the dimensions of the Grid and re-initializes all entries to their default values.
  - `gname.inBounds(row, col)`: Returns **`true`** if the specified row, column position is in the Grid, **`false`** otherwise.

- For the exhaustive list, check out the Stanford Grid documentation.

How to traverse a Grid

```
void printGrid(Grid<char>& grid) {
    for(int r = 0; r < grid.numRows(); r++) {
        for(int c = 0; c < grid.numCols(); c++) {
            cout << grid[r][c];
        }
        cout << endl;
    }
}
```

How to traverse a Grid

```
void printGrid(Grid<char>& grid) {
    for(int r = 0; r < grid.numRows(); r++) {
        for(int c = 0; c < grid.numCols(); c++) {
            cout << grid[r][c];
        }
        cout << endl;
    }
}
```

'y'	'e'
'e'	'h'
'a'	'w'

Poll: What is the output of this function called on the provided grid going to be?

A. yeehaw

B. yea
ehw

C. ye
eh
aw

D. None of the above

How to traverse a Grid

```
void printGrid(Grid<char>& grid) {
    for(int r = 0; r < grid.numRows(); r++) {
        for(int c = 0; c < grid.numCols(); c++) {
            cout << grid[r][c];
        }
        cout << endl;
    }
}
```

Output:

```
Variables:
r = 0
c = 0
```

'y'	'e'
'e'	'h'
'a'	'w'

How to traverse a Grid

Variables:
r = 0
c = 0

```
void printGrid(Grid<char>& grid) {
    for(int r = 0; r < grid.numRows(); r++) {
        for(int c = 0; c < grid.numCols(); c++) {
            cout << grid[r][c];
        }
        cout << endl;
    }
}
```

'y'	'e'
'e'	'h'
'a'	'w'

Output:

y

How to traverse a Grid

Variables:
r = 0
c = 1

```
void printGrid(Grid<char>& grid) {
    for(int r = 0; r < grid.numRows(); r++) {
        for(int c = 0; c < grid.numCols(); c++) {
            cout << grid[r][c];
        }
        cout << endl;
    }
}
```

'y'	'e'
'e'	'h'
'a'	'w'

Output:
ye

How to traverse a Grid

Variables:
r = 1
c = 0

```
void printGrid(Grid<char>& grid) {
    for(int r = 0; r < grid.numRows(); r++) {
        for(int c = 0; c < grid.numCols(); c++) {
            cout << grid[r][c];
        }
        cout << endl;
    }
}
```

'y'	'e'
'e'	'h'
'a'	'w'

Output:
ye
e

How to traverse a Grid

```
void printGrid(Grid<char>& grid) {
    for(int r = 0; r < grid.numRows(); r++) {
        for(int c = 0; c < grid.numCols(); c++) {
            cout << grid[r][c];
        }
        cout << endl;
    }
}
```

Output:

ye
eh

Variables:
r = 1
c = 1

'y'	'e'
'e'	'h'
'a'	'w'

How to traverse a Grid

```
void printGrid(Grid<char>& grid) {
    for(int r = 0; r < grid.numRows(); r++) {
        for(int c = 0; c < grid.numCols(); c++) {
            cout << grid[r][c];
        }
        cout << endl;
    }
}
```

Output:

```
ye
eh
a
```

Variables:

```
r = 2
c = 0
```

'y'	'e'
'e'	'h'
'a'	'w'

How to traverse a Grid

```
void printGrid(Grid<char>& grid) {
    for(int r = 0; r < grid.numRows(); r++) {
        for(int c = 0; c < grid.numCols(); c++) {
            cout << grid[r][c];
        }
        cout << endl;
    }
}
```

Output:

```
ye
eh
aw
```

Variables:

```
r = 2
c = 1
```

'y'	'e'
'e'	'h'
'a'	'w'

Common pitfalls when using Grids

- Don't forget to specify what data type is stored in your grid

  NO: **`Grid board;`** YES: **`Grid<char> board;`**

- Like Vectors and other ADTs, Grids should be passed by reference when used as function parameters

- Watch your variable ordering with Grid indices! Rather than using **`i`** and **`j`** as indices to loop through a grid, it's better to use **`r`** for rows and **`c`** for columns.

- Unlike in other languages, you can only access cells (not individual rows).

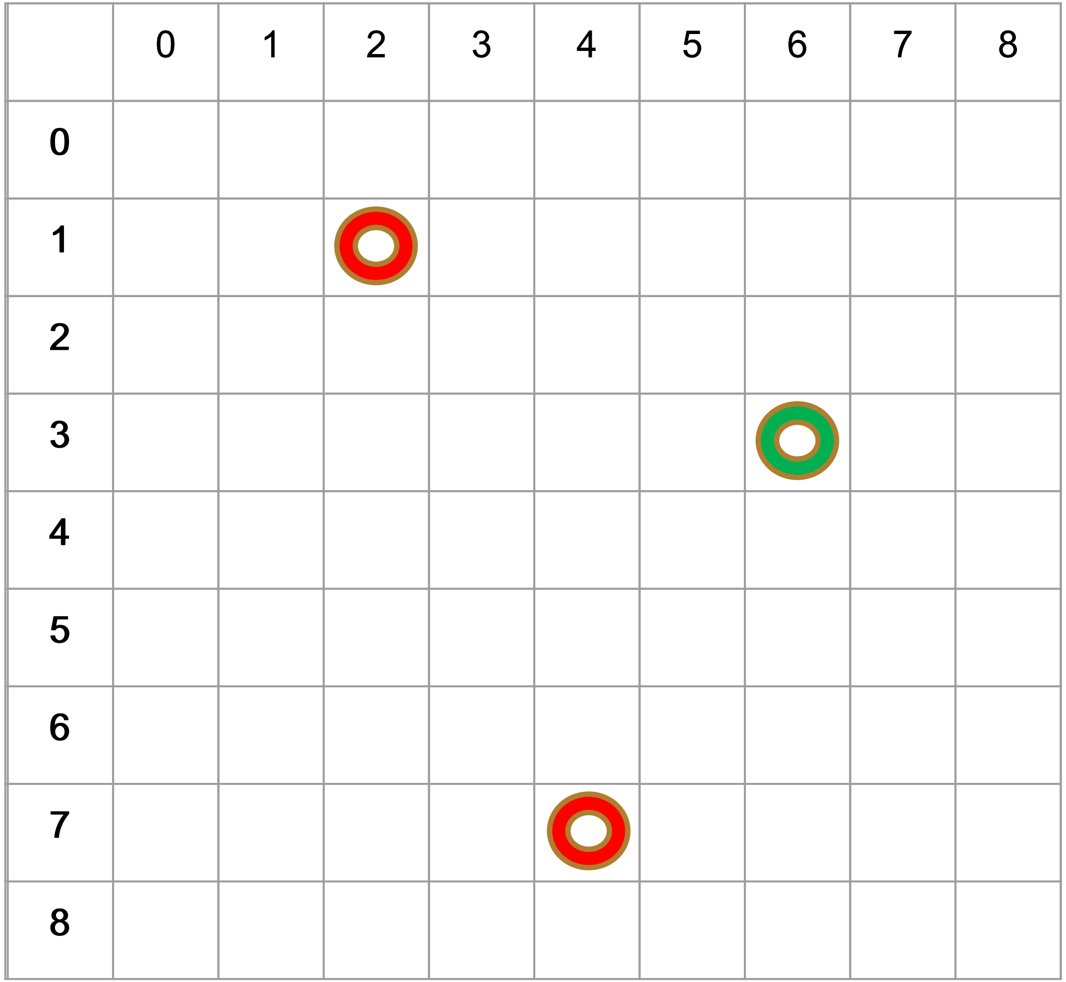
  `grid[0]` → doing this will cause an error!

Detecting, Destroying, and Removing Landmines:

Turning dangerous fields into playing fields

Source: https://www.cambodiadaily.com/news/thai-army-assists-with-removal-of-cambodian-border-mines-150588/

	0	1	2	3	4	5	6	7	8
0									
1									
2									
3									
4									
5									
6									
7									
8									



What if we want to keep track of all cells where a mine is present?

And we want to note whether it is no longer explosive.

Structs + `GridLocation`

Definition

struct

A way to bundle different types of info in C++.
It's like creating a custom data structure.

The **`GridLocation`** struct

- A pre-defined struct in the Stanford C++ libraries that makes it more convenient to store Grid locations:

```
struct GridLocation {
    int row;
    int col;
}
```

struct definition

The **GridLocation** struct

- A pre-defined struct in the Stanford C++ libraries that makes it more convenient to store Grid locations:

```
struct GridLocation {
    int row;
    int col;
}
```

struct members (these can be different types)

The **GridLocation** struct

- A pre-defined struct in the Stanford C++ libraries that makes it more convenient to store Grid locations

- To declare a struct, you can either assign each of its members separately or assign it when it's created:

```
GridLocation origin = {0, 0};
```

```
GridLocation origin;
origin.row = 0;
origin.col = 0;
```

```
struct GridLocation {
    int row;
    int col;
}
```

The **GridLocation** struct

- A pre-defined struct in the Stanford C++ libraries that makes it more convenient to store Grid locations

- To declare a struct, you can either assign each of its members separately or assign it when it's created:

```
GridLocation origin = {0, 0};
```

You can access members in a struct using the dot notation (no parentheses after the member name!)

```
GridLocation origin;
origin.row = 0;
origin.col = 0;
```

```
struct GridLocation {
    int row;
    int col;
}
```

The **GridLocation** struct

- A pre-defined struct in the Stanford C++ libraries that makes it more convenient to store Grid locations

- To declare a struct, you can either assign each of its members separately or assign it when it's created:

```
GridLocation origin = {0, 0};
```

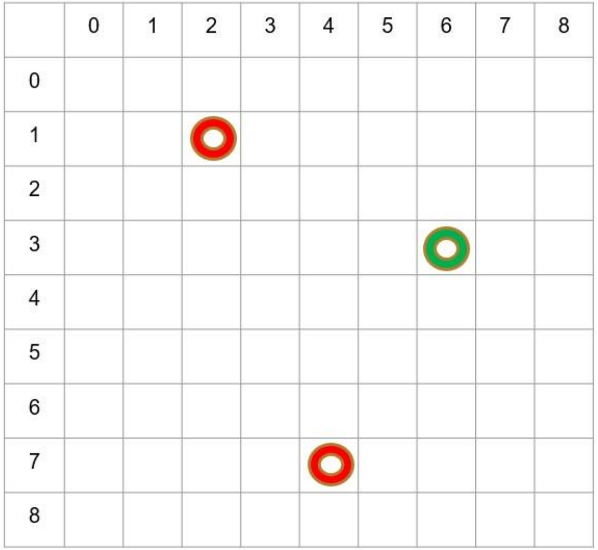
```
GridLocation origin;
origin.row = 0;
origin.col = 0;
```

```
struct GridLocation {
    int row;
    int col;
}
```

```
Vector<GridLocation> mineCells;
GridLocation eastField = {3,6};
GridLocation westField = {1,2};
mineCells.add(eastField);
mineCells.add(westField);
```

Vs.

```
Vector<int> rowIndices;
Vector<int> colIndices;
rowIndices.add(3);
rowIndices.add(1);
colIndices.add(6);
colIndices.add(2);
```



As an exercise on your own: Think about how you would answer the question "Is there a mine at (4, 3)?" for each of the different representations (with and without GridLocation structs).

Announcements

Announcements

- Assignment 1 is due tomorrow (Tuesday) at 11:59pm PDT!
  - Recall our 48 hour late policy, but please submit on-time if you want a small grade boost!
  - If you submit more than once after tomorrow's deadline, please notify your SL. Else, they may not grade your newest version.
  - Don't use EXPECT_ERROR on A1. We haven't covered error handling yet, so EXPECT_ERROR will not compile. S/o to Tomas for finding this!

- Assignment 2 will be released by the end of the day on Wednesday.
- If you'd like to submit an extension to a program for extra credit, please be sure to submit **two versions** of your assignment to paperless, one with and one without the extension!

Queues

What is a queue?

- Like a real queue/line!

- **F**irst person **I**n is the
  First person **O**ut (FIFO)
  - When you remove (**dequeue**) people from the queue, you remove them from the front of the line.

- Last person in is the last person served
  - When you insert (**enqueue**) people into a queue, you insert them at the back -- the end of the line

Queue methods

- A queue must implement at least the following functions:
  - **`enqueue(value)`** (or **`add(value)`**) - place an entity onto the back of the queue
  - **`dequeue()`** (or **`remove()`**) - remove an entity from the front of the queue and return it
  - **`peek()`** (or **`front()`**) - look at the entity at the front of the queue, but don't remove it
  - **`isEmpty()`** - a boolean value, true if the queue is empty, false if it has at least one element. (note: a runtime error occurs if a dequeue() or front() operation is attempted on an empty queue).
- For the exhaustive list, check out the Stanford Queue documentation.

Queue example

```
Queue<int> line;                          // {}       empty queue
line.enqueue(42);                         // {42}
line.enqueue(-3);                         // {42, -3}
line.enqueue(17);                         // {42, -3, 17}
cout << line.dequeue() << endl;           //  42      line is {-3, 17}
cout << line.peek() << endl;              // -3       line is {-3, 17}
cout << line.dequeue() << endl;  // -3      line is {17}
```

You can also create a queue using:

```
Queue<int> line = {42, -3, 17};
```

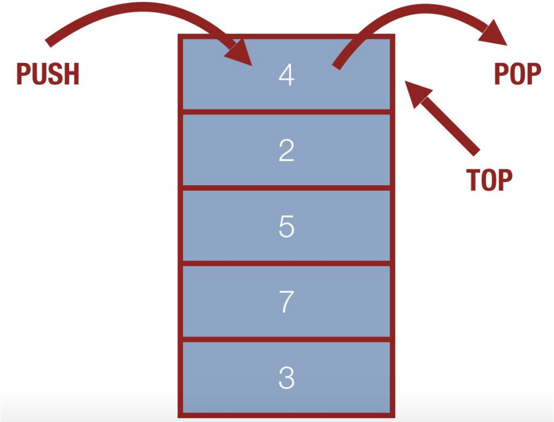
Stacks

What is a stack?

- Modeled like an actual stack (of pancakes)

- Only the top element in a stack is accessible.
  - The **L**ast item **I**n is the **F**irst one **O**ut. (LIFO)

- The push, pop, and top operations are the only operations allowed by the stack ADT.



Stack methods

- A stack is an abstract data type with the following behaviors/functions:
  - **`push(value)`** (or **`add(value)`**) - place an entity onto the top of the stack
  - **`pop()`** (or **`remove()`**) - remove an entity from the top of the stack and return it
  - **`peek()`** (or **`top()`**) - look at the entity at the top of the stack, but don't remove it
  - **`isEmpty()`** - a boolean value, true if the stack is empty, false if it has at least one element. (Note: a runtime error occurs if a **`pop()`** or **`top()`** operation is attempted on an empty stack.)
- For the exhaustive list, check out the Stanford Stack documentation.

Stack example

```
Stack<string> wordStack;                 // {}, empty stack
wordStack.push(“dog”);               // {“dog”}
wordStack.push(“cat”);               // {“dog”, “cat”}
wordStack.push(“mouse”);                 // {“dog”, “cat”, “mouse”}
cout << wordStack.pop() << endl;     // “mouse”
cout << wordStack.peek() << endl;    // “cat”
cout << wordStack.pop() << endl; // “cat” (stack is {“dog”})

// You can also create a stack using:
Stack<string> wordStack = {“dog”, “cat”, “mouse”};
// the “top” is the rightmost element
```

Queue + Stack patterns

Common patterns and pitfalls with stacks and queues

Idioms:

1. Emptying a stack/queue

Idiom 1: Emptying a queue/stack

```
Queue<int> queueIdiom1;

// produce: {1, 2, 3, 4, 5, 6}
for (int i = 1; i <= 6; i++) {
  queueIdiom1.enqueue(i);
}

while (!queueIdiom1.isEmpty()) {
  cout << queueIdiom1.dequeue() << " ";
}
cout << endl;

// prints: 1 2 3 4 5 6
```

Idiom 1: Emptying a queue/stack

```
Queue<int> queueIdiom1;

// produce: {1, 2, 3, 4, 5, 6}
for (int i = 1; i <= 6; i++) {
  queueIdiom1.enqueue(i);
}

while (!queueIdiom1.isEmpty()) {
  cout << queueIdiom1.dequeue() << " ";
}
cout << endl;

// prints: 1 2 3 4 5 6
```

```
Stack<int> stackIdiom1;

// produce: {1, 2, 3, 4, 5, 6}
for (int i = 1; i <= 6; i++) {
  stackIdiom1.push(i);
}

while (!stackIdiom1.isEmpty()) {
  cout << stackIdiom1.pop() << " ";
}
cout << endl;

// prints: 6 5 4 3 2 1
```

Common patterns and pitfalls with stacks and queues

Idioms:

1. Emptying a stack/queue
2. Iterating over and modifying a stack/queue ➔ only calculate the size once before looping

Idiom 2: Iterating over and modifying queue/stack

```
Queue<int> queueIdiom2 = {1,2,3,4,5,6};

int origQSize = queueIdiom2.size();
for (int i = 0; i < origQSize; i++) {
    int value = queueIdiom2.dequeue();
    // re-enqueue even values
    if (value % 2 == 0) {
        queueIdiom2.enqueue(value);
    }
}
cout << queueIdiom2 << endl;

// prints: {2, 4, 6}
```

Idiom 2: Iterating over and modifying queue/stack

```
Queue<int> queueIdiom2 = {1,2,3,4,5,6};

int origQSize = queueIdiom2.size();
for (int i = 0; i < origQSize; i++) {
   int value = queueIdiom2.dequeue();
   // re-enqueue even values
   if (value % 2 == 0) {
      queueIdiom2.enqueue(value);
   }
}
cout << queueIdiom2 << endl;

// prints: {2, 4, 6}
```

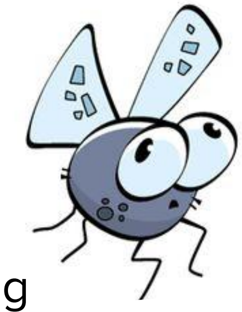
```
Stack<int> stackIdiom2 = {1,2,3,4,5,6};
Stack<int> result;

int origSSize = stackIdiom2.size();
for (int i = 0; i < origSSize; i++) {
   int value = stackIdiom2.pop();
   // add even values to result
   if (value % 2 == 0) {
       result.push(value);
   }
}
cout << result << endl;

// prints: {6, 4, 2}
```

Common patterns and pitfalls with stacks and queues

Idioms:

1. Emptying a stack/queue
2. Iterating over and modifying a stack/queue ➔ only calculate the size once before looping

Common bugs:



- If you edit the ADT within a loop, don't use `.size()` in the loop's conditions! The size changes while the loop runs.
- Unlike with queues, you can't iterate over a stack without destroying it ➔ think about when it might be beneficial to make a copy instead.

Tradeoffs with queues and stacks (vs. other ADTs)

- What are some downsides to using a queue/stack?
    - No random access. You get the front/top, or nothing.
    - No side-effect-free traversal — you can only iterate over all elements in the structure by removing previous elements first.
    - No easy way to search through a queue/stack.

- What are some benefits?
    - Useful for lots of problems – many real-world problems can be solved with either a LIFO or FIFO model
    - Very easy to build one from an array such that access is guaranteed to be fast. (We’ll talk more about arrays later in the quarter, and we'll talk about what "fast" access means later this week.)

Activity: What ADT should we use?

For each of the tasks, pick which ADT is best suited for the task:

Vectors

Grids

Queues

Stacks

- The undo button in a text editor
- Jobs submitted to a printer that can also be cancelled
- LaIR requests
- Your browsing history
- Google spreadsheets
- Call centers (“your call will be handled by the next available agent”)

For each of the tasks, pick which ADT is best suited for the task:

Vectors

Grids

Queues

Stacks

- **The undo button in a text editor**
- Jobs submitted to a printer that can also be cancelled
- LaIR requests
- Your browsing history
- Google spreadsheets
- Call centers (“your call will be handled by the next available agent”)

For each of the tasks, pick which ADT is best suited for the task:

Vectors

Grids

(Queues)

Stacks

- The undo button in a text editor
- Jobs submitted to a printer that can also be cancelled
- LaIR requests
- Your browsing history
- Google spreadsheets
- Call centers ("your call will be handled by the next available agent")

For each of the tasks, pick which ADT is best suited for the task:

Vectors

Grids

Queues

Stacks

- The undo button in a text editor
- Jobs submitted to a printer that can also be cancelled
- **LaIR requests**
- Your browsing history
- Google spreadsheets
- Call centers ("your call will be handled by the next available agent")

For each of the tasks, pick which ADT is best suited for the task:

Vectors

Grids

Queues

Stacks

- The undo button in a text editor
- Jobs submitted to a printer that can also be cancelled
- LaIR requests
- **Your browsing history**
- Google spreadsheets
- Call centers (“your call will be handled by the next available agent”)

For each of the tasks, pick which ADT is best suited for the task:

Vectors

Grids

Queues

Stacks

- The undo button in a text editor
- Jobs submitted to a printer that can also be cancelled
- LaIR requests
- Your browsing history
- **Google spreadsheets**
- Call centers ("your call will be handled by the next available agent")

For each of the tasks, pick which ADT is best suited for the task:

Vectors

Grids

Queues

Stacks

- The undo button in a text editor
- Jobs submitted to a printer that can also be cancelled
- LaIR requests
- Your browsing history
- Google spreadsheets
- **Call centers (“your call will be handled by the next available agent”)**

ADTs summary (so far)

Ordered ADTs with accessible indices

Types:

- Vectors (1D)
- Grids (2D)

Traits:

- Easily able to search through all elements
- Can use the indices as a way of structuring the data

Ordered ADTs where you can't access elements by index

Types:

- Queues (FIFO)
- Stacks (LIFO)

Traits:

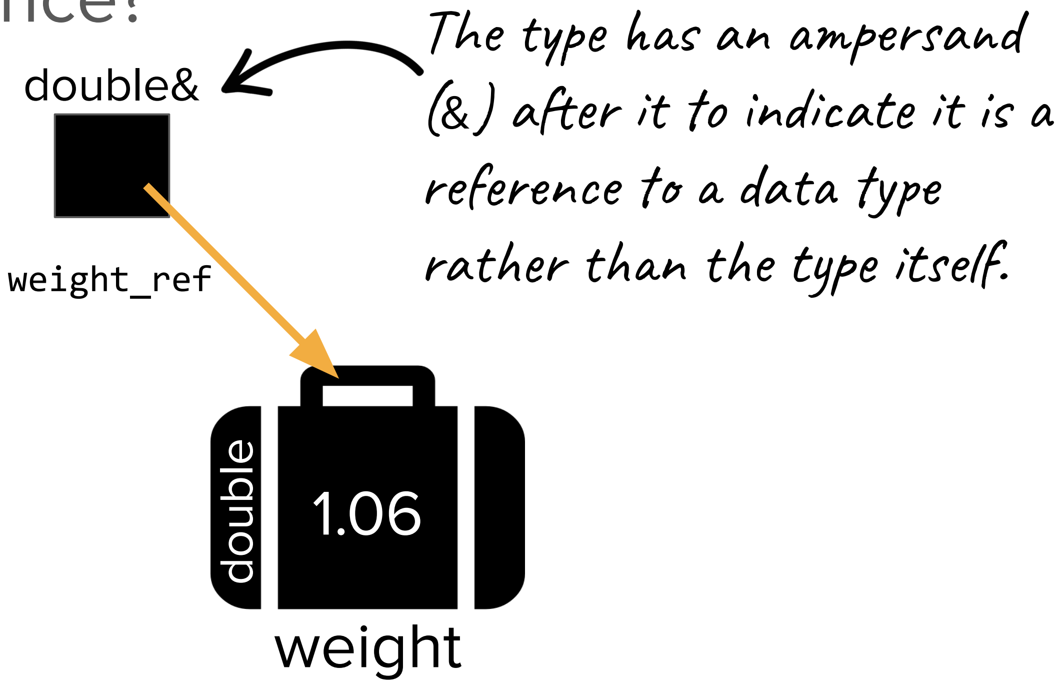
- Constrains the way you can insert and access data
- More efficient for solving specific LIFO/FIFO problems

Review

(pass-by-reference)

What exactly is a reference?

- References look like this:



References have names and types, just like regular variables.

When we use references

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

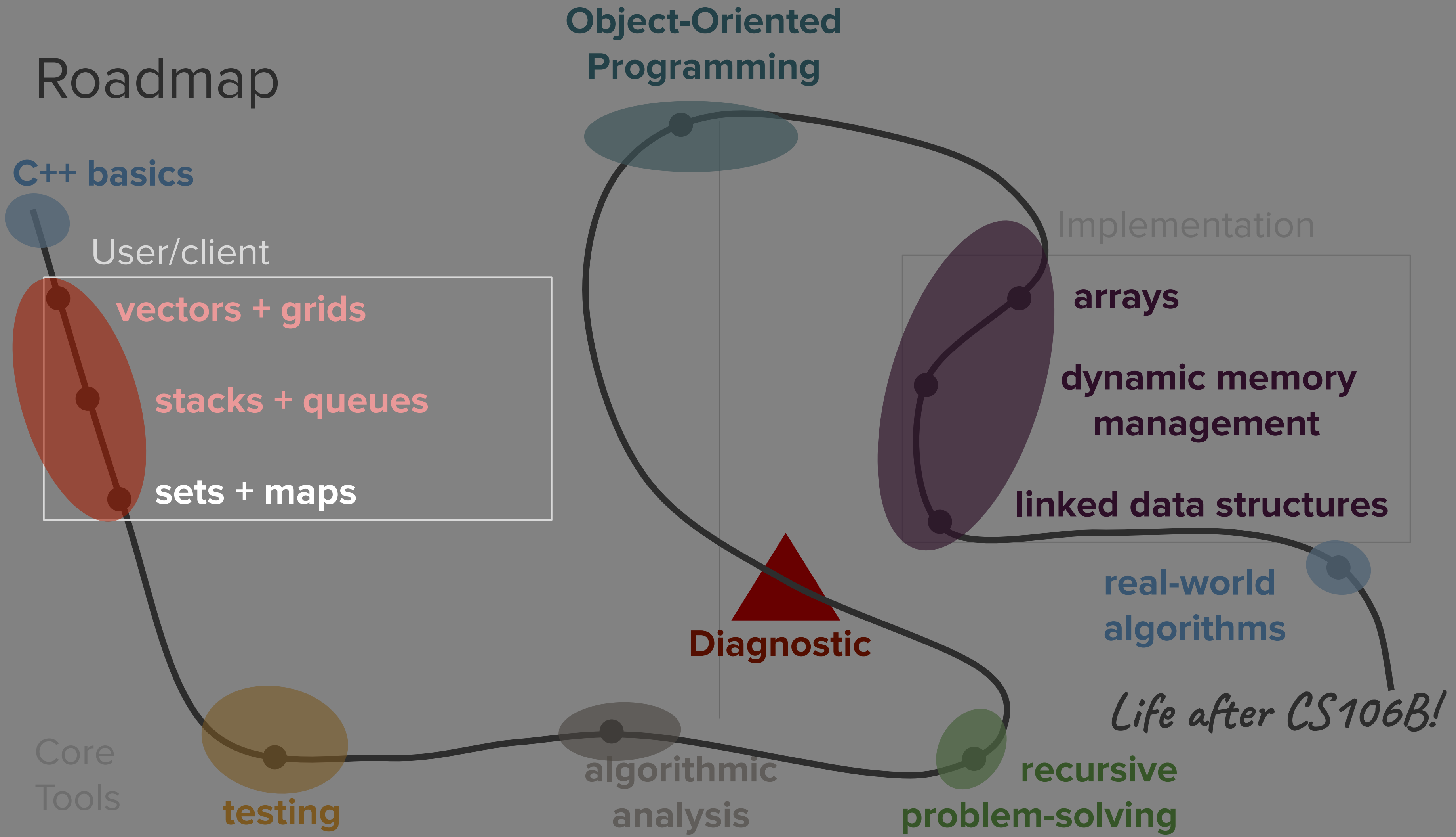
- To avoid making new copies of large data structures in memory

  Passing data structures by reference makes your code more efficient!

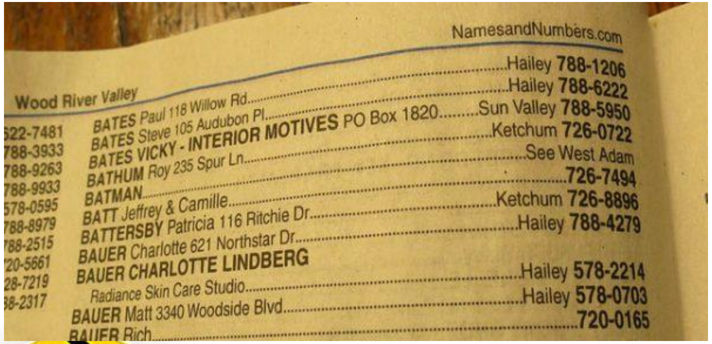
- References also provide a workaround for **multiple return values**
  - Your function can take in multiple pieces of information by reference and modify them all.
  - In this way, you can "return" both a modified Vector and some auxiliary piece of information about how the structure was modified.
  - It’s as if your function is returning 2 updated pieces of information to the function that called it!

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

Unordered ADTs: Sets and Maps


NamesandNumbers.com
Wood River Valley
BATES Paul 118 Willow Rd. .... Hailey 788-1206
BATES Steve 105 Audubon Pl. .... Hailey 788-6222
BATES VICKY - INTERIOR MOTIVES PO Box 1820 .... Sun Valley 788-5950
BATHUM Roy 235 Spur Ln. .... Ketchum 726-0722
BATMAN .... See West Adam
BATT Jeffrey & Camille 726-7494
BATTERSBY Patricia 116 Ritchie Dr. .... Ketchum 726-8896
BAUER Charlotte 621 Northstar Dr. .... Hailey 788-4279
BAUER CHARLOTTE LINDBERG
Radiance Skin Care Studio .... Hailey 578-2214
BAUER Matt 3340 Woodside Blvd. .... Hailey 578-0703
BAUER Rich 720-0165