

# Ordered Data Structures: Grids, Queues, and Stacks

What's an example of “ordered data” that you've encountered in your life?  
(put your answers the chat)



# Roadmap

## C++ basics

User/client

**vectors + grids**

**stacks + queues**

**sets + maps**

## Object-Oriented Programming

Implementation

**arrays**

**dynamic memory  
management**

**linked data structures**

**real-world  
algorithms**

**Diagnostic**

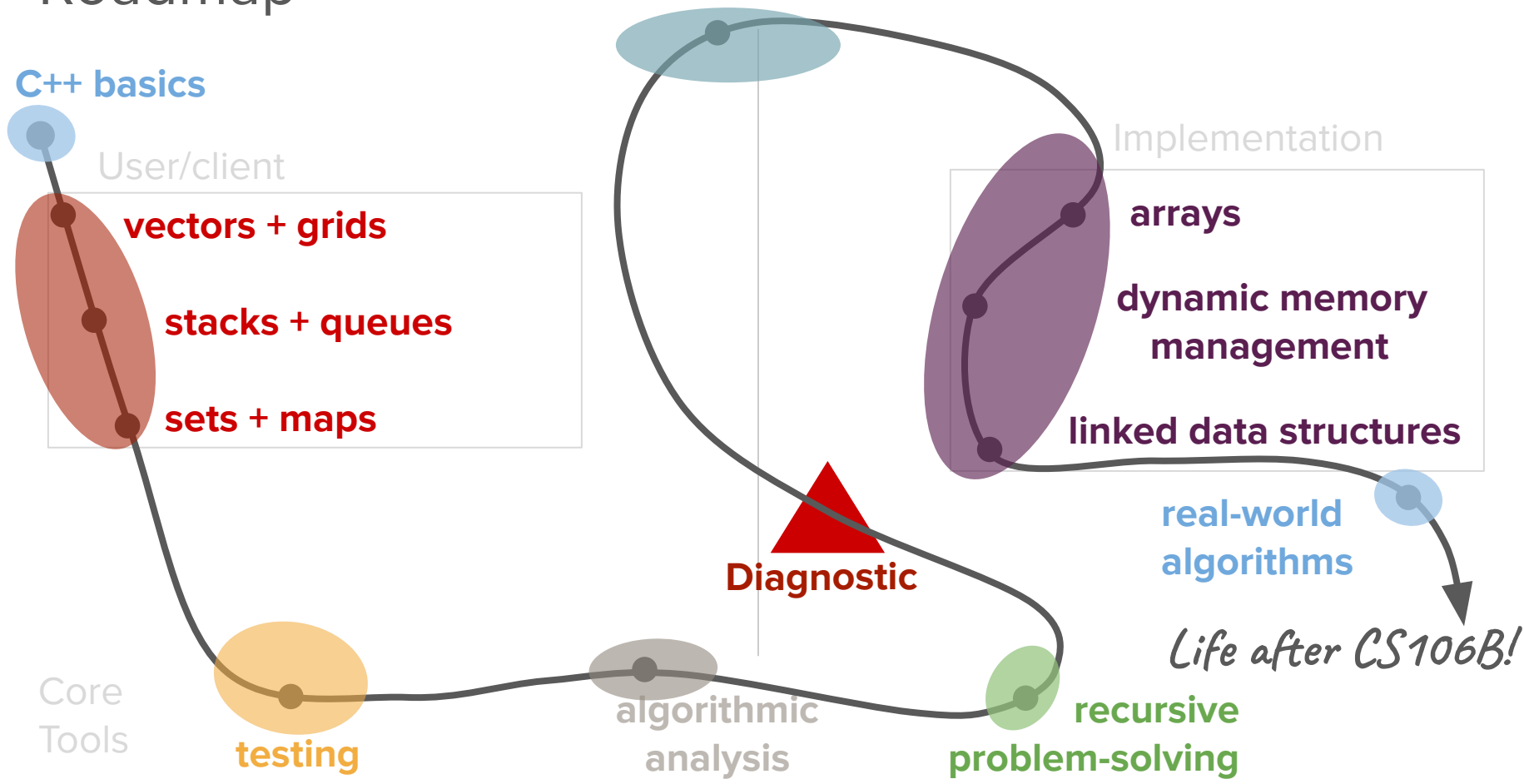
Core  
Tools

**testing**

**algorithmic  
analysis**

**recursive  
problem-solving**

*Life after CS106B!*



# Roadmap

## C++ basics

User/client

vectors + grids

stacks + queues

sets + maps

## Object-Oriented Programming

Implementation

arrays

dynamic memory  
management

linked data structures

real-world  
algorithms

 Diagnostic

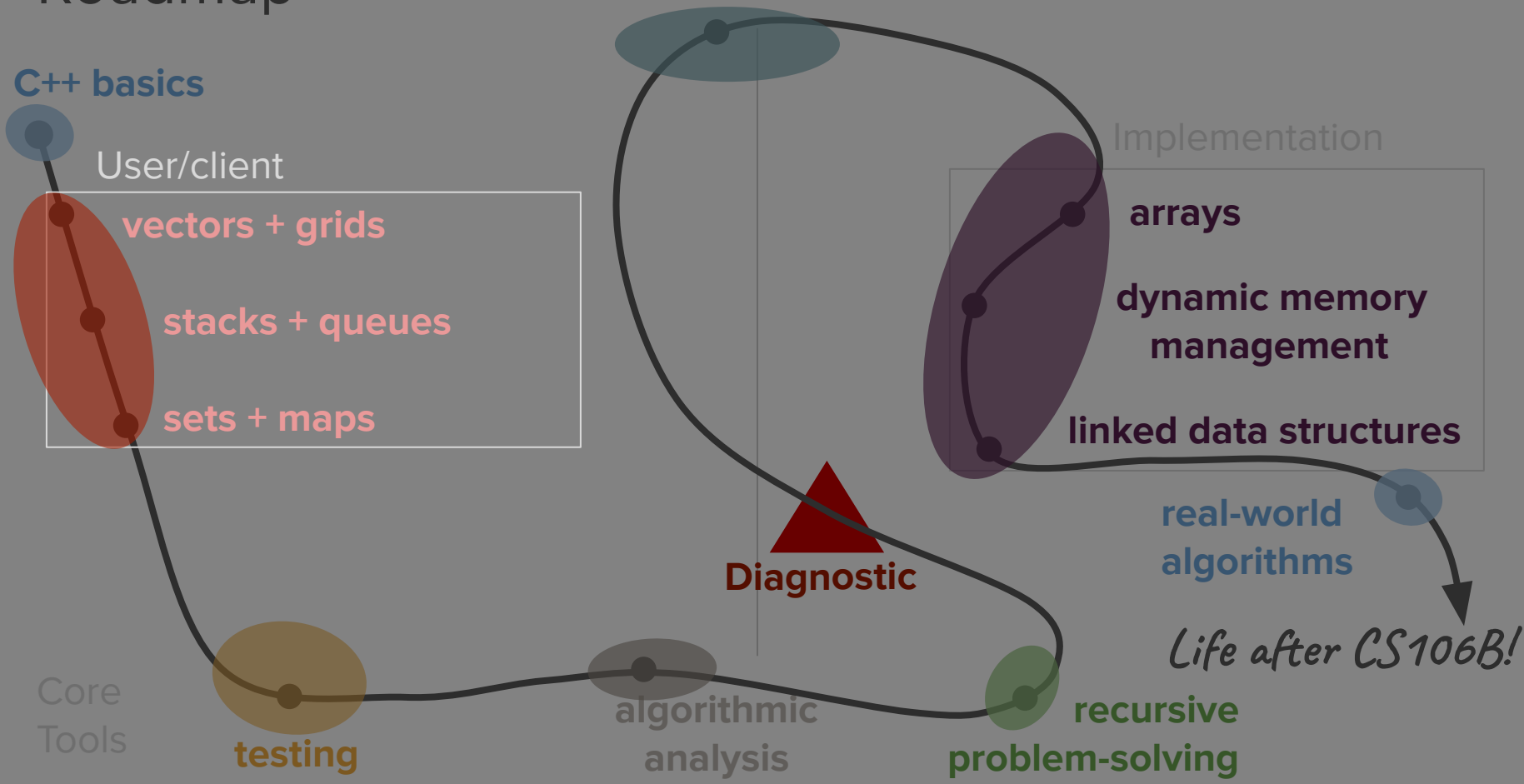
*Life after CS106B!*

Core  
Tools

testing

algorithmic  
analysis

recursive  
problem-solving



# Roadmap

## C++ basics

User/client

**vectors + grids**

**stacks + queues**

**sets + maps**

## Object-Oriented Programming

Implementation

**arrays**

**dynamic memory  
management**

**linked data structures**

**real-world  
algorithms**

*Life after CS106B!*

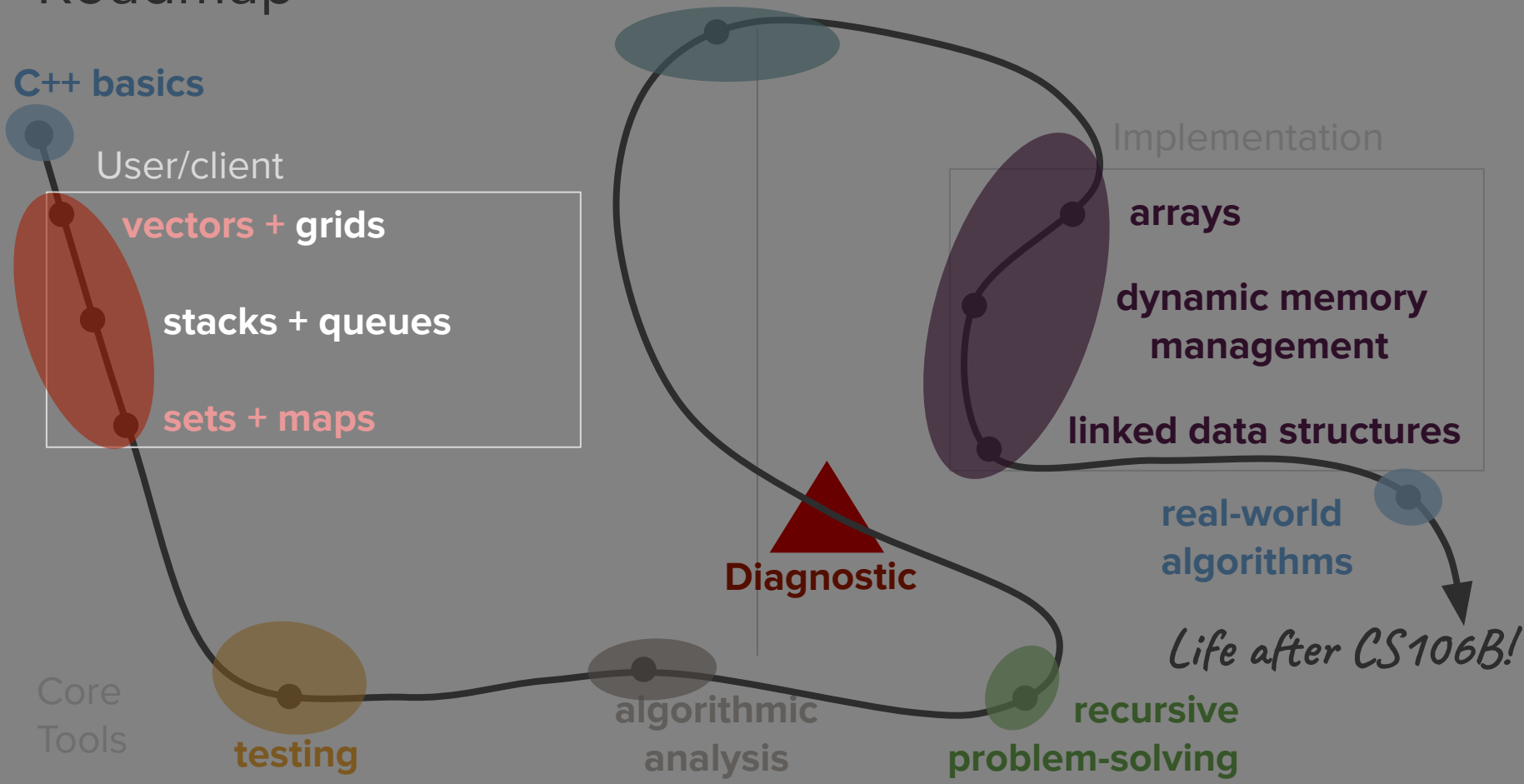
**Diagnostic**

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]

3. Queues

4. Stacks



# Review

(vectors and pass-by-reference)

# 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

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





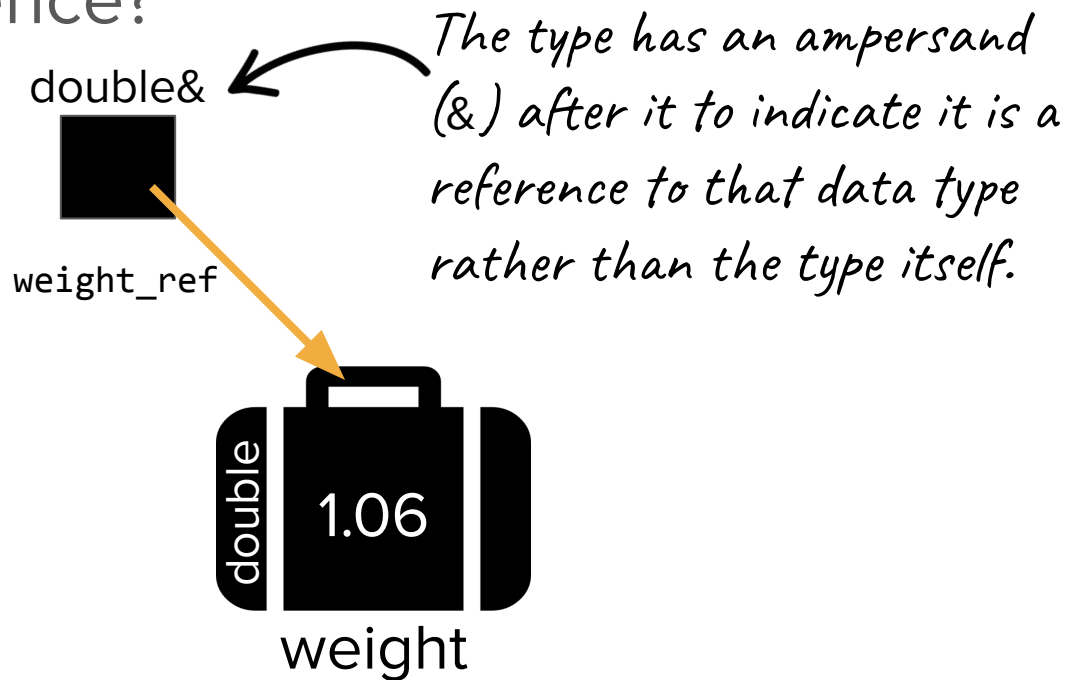
# Our first ADT: **Vectors**

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

# 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
  - 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!



# Trace problem

*[5-minute breakout rooms!]*



# Grids

# 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.*

a0	a1	a2
b0	b1	b2
c0	c1	c2

# What is a grid?

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

a0	a1	a2
b0	b1	b2
c0	c1	c2

# What is a grid?

- A 2D array, defined with a particular width and height
- Useful for spreadsheets, game boards, etc.
- Three ways to declare a grid
  - `Grid<type> gridName;`

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



*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.*

a0	a1	a2
b0	b1	b2
c0	c1	c2



# What is a grid?

- A 2D array, defined with a particular width and height
- Useful for spreadsheets, game boards, etc.
- Three ways to declare a grid
  - `Grid<type> gridName(numRows, numCols);`

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

a0	a1	a2
b0	b1	b2
c0	c1	c2

# What is a grid?

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

**`Grid<int> board = {{2,0}, {6,0}};`**

a0	a1	a2
b0	b1	b2
c0	c1	c2

# What is a grid?

- A 2D array, defined with a particular width and height
- Useful for spreadsheets, game boards, etc.
- Three ways to declare a grid
  - `Grid<type> gridName;`
  - `Grid<type> gridName(numRows, numCols);`
  - `Grid<type> gridName = {{r0c0, r0c1, r0c2}, {r1c0, r1c1, r1c2},...};`
- We could use a combination of Vectors to simulate a 2D matrix, but a Grid is easier!

a0	a1	a2
b0	b1	b2
c0	c1	c2



# Grid methods

- The following methods are part of the grid collection and can be useful:
  - `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 **i**th row and **j**th 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 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;  
    }  
}
```

'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



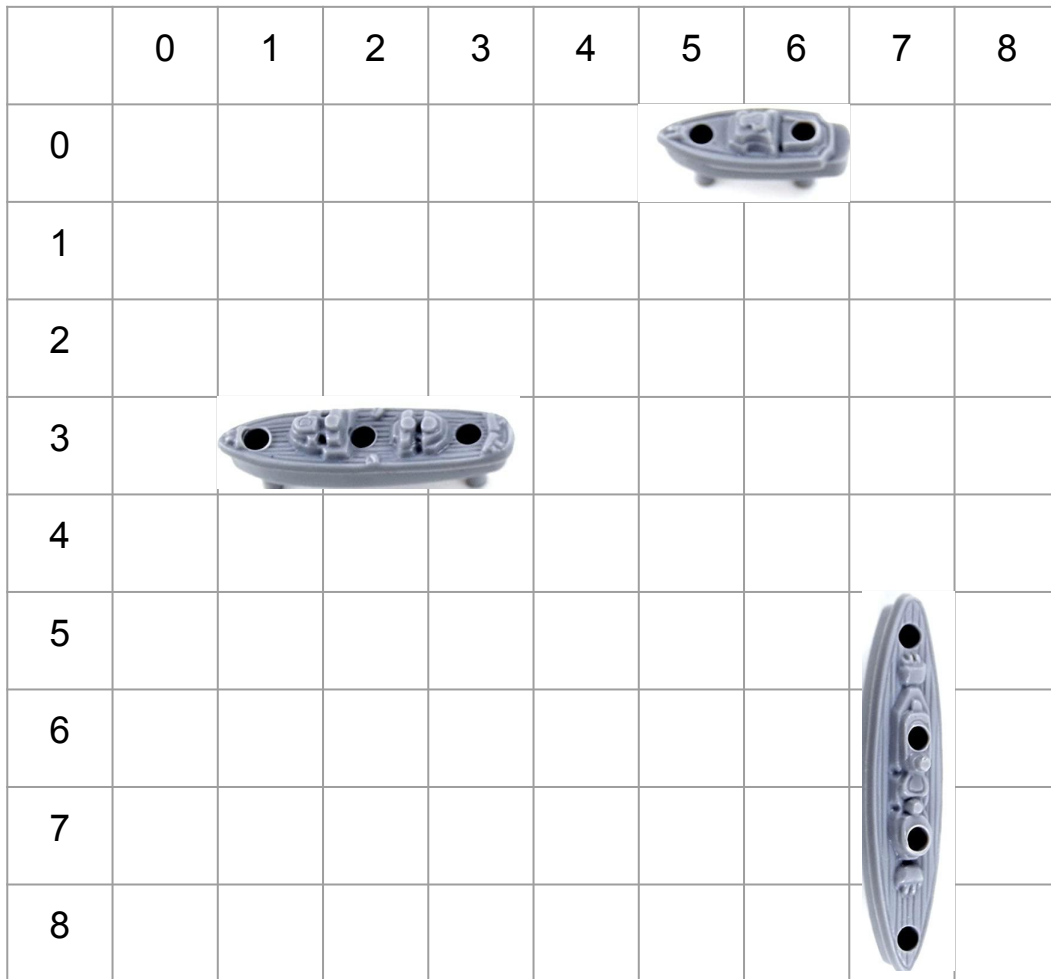
# Common pitfalls when using Grids

- 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!



Battleship uses  
a grid!

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



# Structs + **GridLocation**





## *Definition*

### **struct**

A way to bundle different types of information in C++ – 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};
```



*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;  
}
```

```
Vector<GridLocation> shipCells;  
GridLocation smallLeft = {0,5};  
GridLocation smallRight = {0,6};  
shipCells.add(smallLeft);  
shipCells.add(smallRight);
```

VS.

```
Vector<int> rowIndices;  
Vector<int> colIndices;  
rowIndices.add(0);  
rowIndices.add(0);  
colIndices.add(5);  
colIndices.add(6);
```

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

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

# 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:

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

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

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

***NOTE:** Grids are not made up of GridLocation structs! GridLocations are just a convenient way to store a single cell or a path of cells within a Grid.*



# Announcements



# Announcements

- Assignment 1 is due tomorrow (Tuesday) at 11:59pm in your time zone!
- Check out our [Ed post](#) answering your questions from the C++ survey.
  - It includes review of commonly asked questions, language-specific differences, and a list of resources.
- Assignment 2 will be released by the end of the day on Wednesday.





# Queues

# What is a queue?

- Like a real queue/line!
- **F**irst person **I**n is the **F**irst 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:*

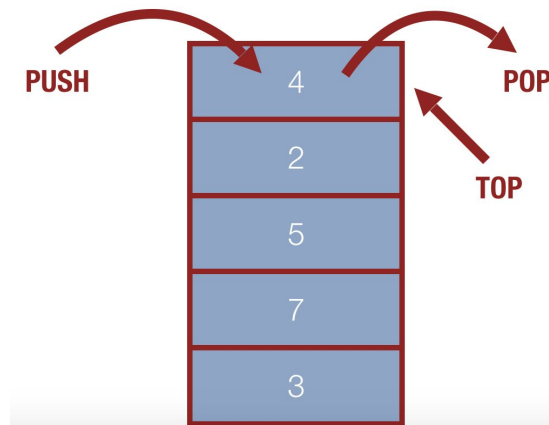
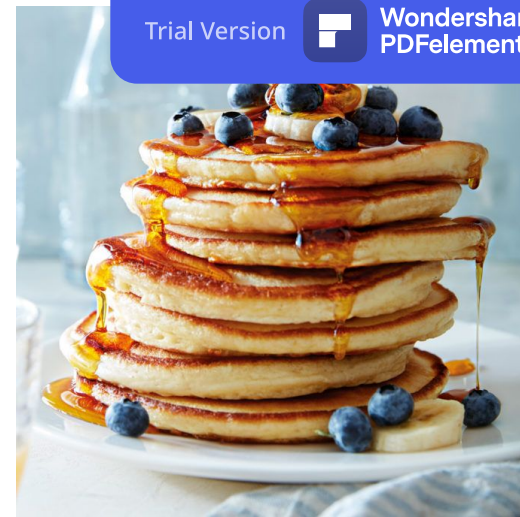
```
Queue<int> line = {42, -3, 17};
```



# 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("Kylie");           // {"Kylie"}
wordStack.push("Nick");            // {"Kylie", "Nick"}
wordStack.push("Trip");            // {"Kylie", "Nick", "Trip"}
cout << wordStack.pop() << endl;   // "Trip"
cout << wordStack.peek() << endl;  // "Nick"
cout << wordStack.pop() << endl;   // "Nick" (stack is {"Kylie"})
```

*// You can also create a stack using:*

```
Stack<string> wordStack = {"Kylie", "Nick", "Trip"};
// 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
```

```
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}
```

```
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}
```

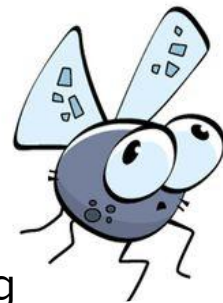
# Common patterns and pitfalls with stacks and queues

## 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.)
    - Where would you have the top of the stack if you build one using a Vector? Why would that be fast?



Activity: What ADT  
should we use?





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

文本编辑器中的撤销按钮 you only care about the last thing you did was

- **The undo button in a text editor**

**Vectors**

- Jobs submitted to a printer that can also be cancelled

**Grids**

- LRU requests

**Queues**

- Your browsing history

**Stacks**

- 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:

- The undo button in a text editor

**Vectors**

- Jobs submitted to a <sup>打印机</sup>printer that can also be cancelled

**Grids**

- layer requests

**(Queues)**

- Your browsing history

**Stacks**

- 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:

- The undo button in a text editor
- Jobs submitted to a printer that can also be cancelled

**Vectors**

**Grids**

- **LaIR requests**

**Queues**

- Your browsing history

**Stacks**

- 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:

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

**Vectors**

**Grids**

**Queues**

**Stacks**

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

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

**Vectors**

**Grids**

**Queues**

**Stacks**

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

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

**Vectors**

**Grids**

**Queues**

**Stacks**



# 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





# What's next?

# Roadmap

## C++ basics

User/client

vectors + grids

stacks + queues

sets + maps

## Object-Oriented Programming

Implementation

arrays

dynamic memory  
management

linked data structures

real-world  
algorithms

 Diagnostic

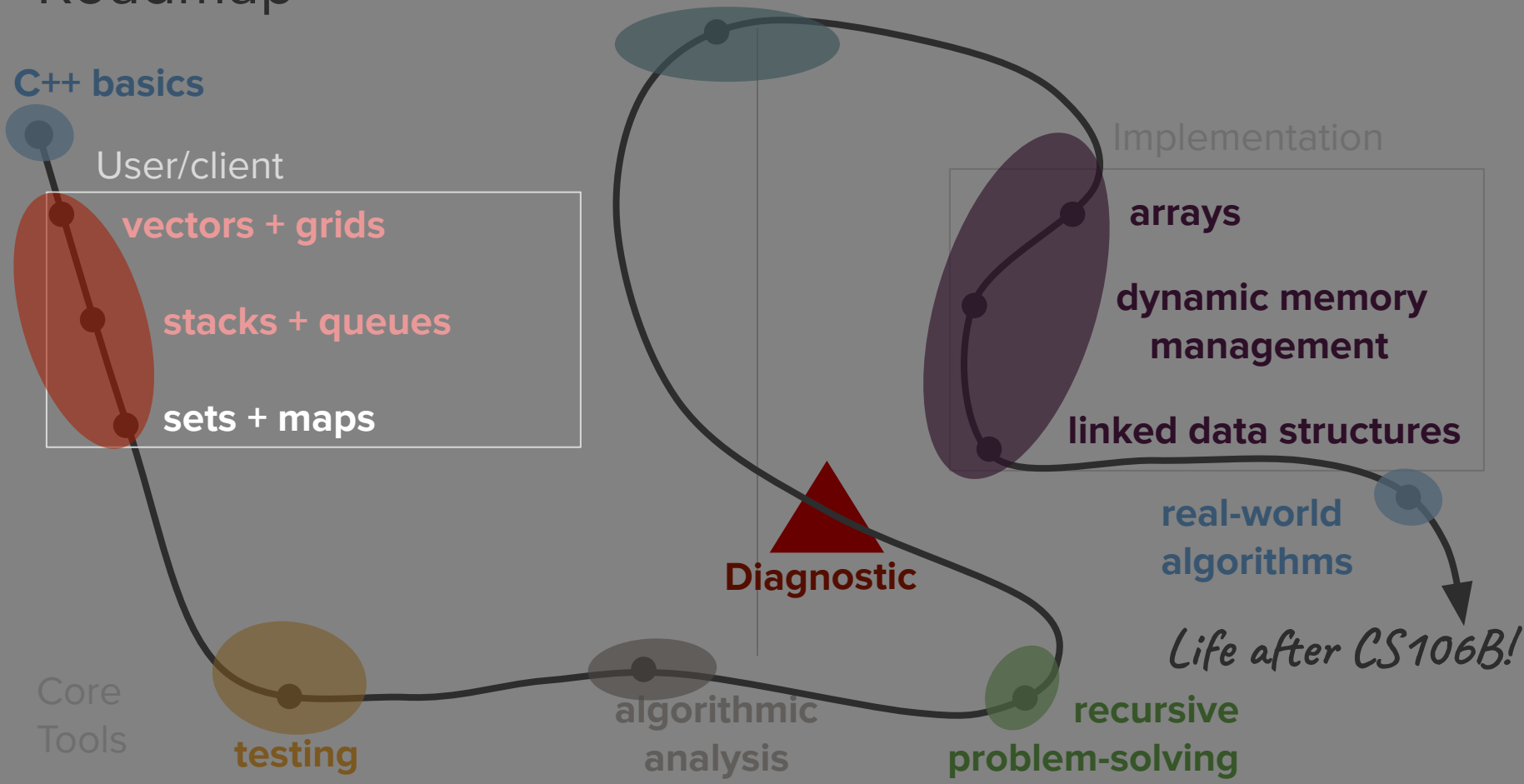
*Life after CS106B!*

Core  
Tools

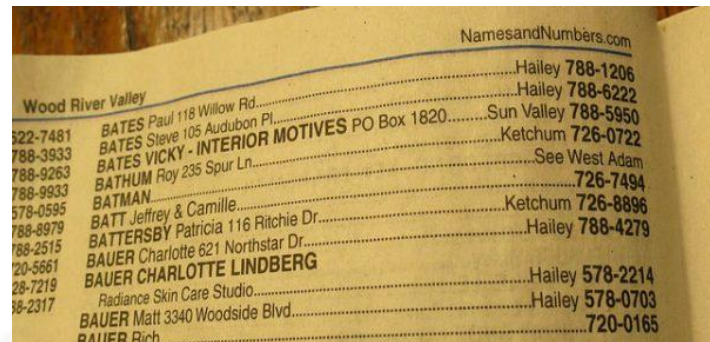
testing

algorithmic  
analysis

recursive  
problem-solving



# Unordered ADTs: Sets and Maps



NamesandNumbers.com

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

