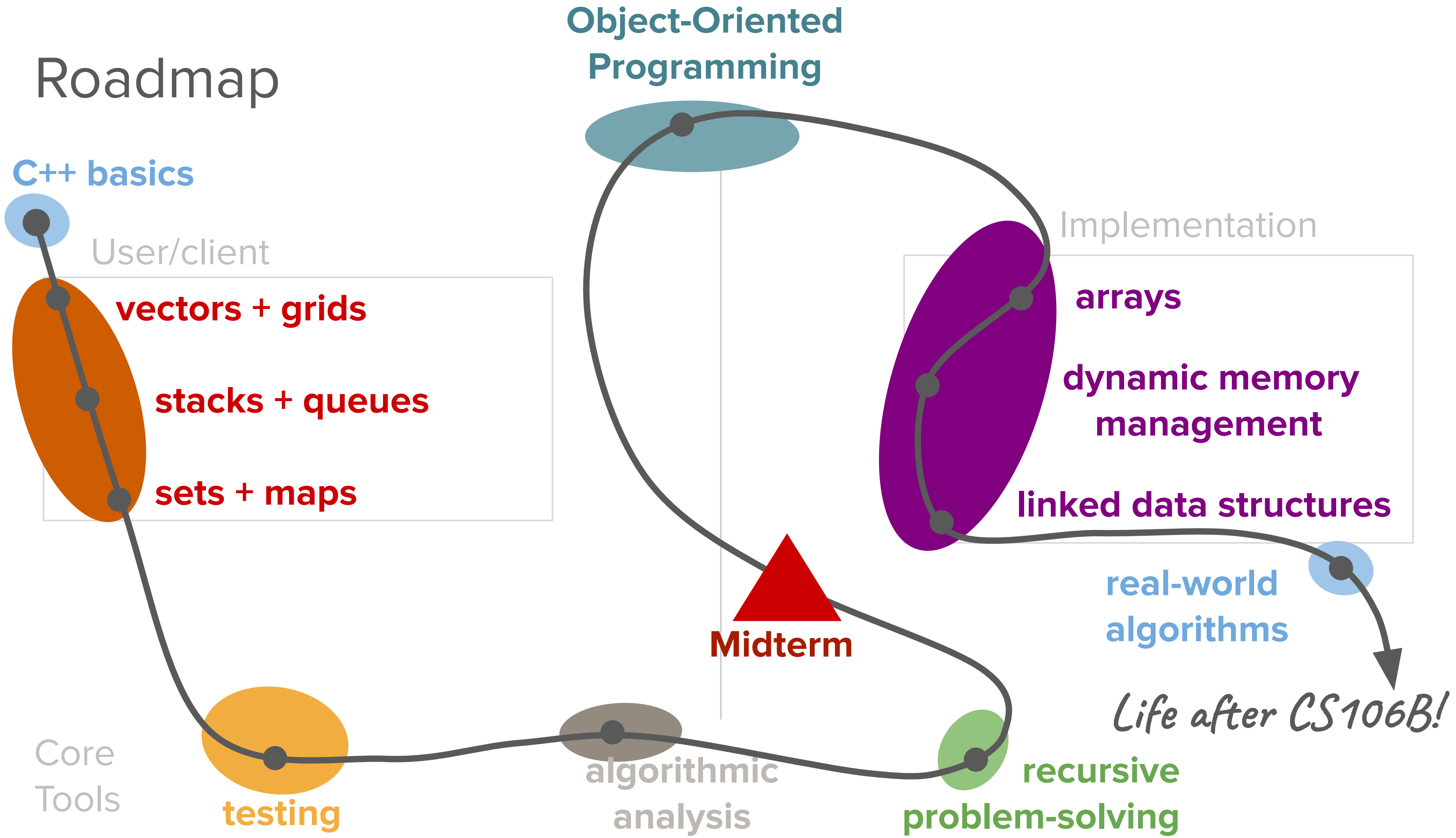
Object-Oriented Programming

What do you think makes a good, well-designed abstraction?

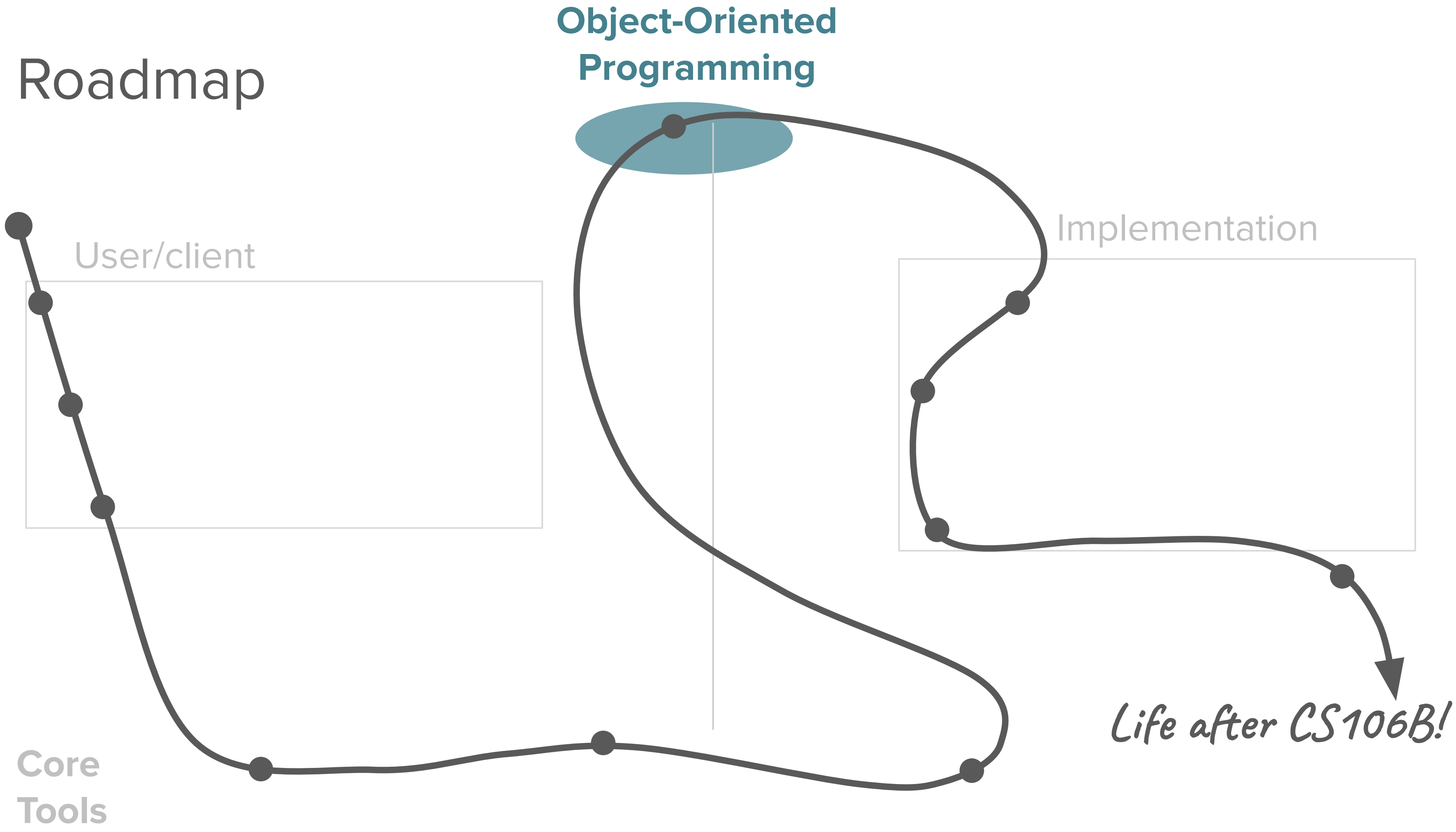
PollEv.com/cs106bpolls

What do you think makes a good, well-designed abstraction?

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
Life after CS106B!
Midterm
Core Tools
testing
algorithmic analysis
recursive problem-solving

Roadmap


Object-Oriented Programming
User/client
Implementation
Core Tools
Life after CS106B!

Today's question

How do we design and define our own abstractions?

Today's topics

1. Review
2. What is a class?
3. Designing C++ classes
4. Writing classes in C++

Review

Backtracking recursion: **Exploring many possible solutions**

Overall paradigm: choose/explore/unchoose

Two ways of doing it

- **Choose explore undo**
    - Uses pass by reference; usually with large data structures
    - Explicit unchoose step by "undoing" prior modifications to structure
    - E.g. Generating subsets (one set passed around by reference to track subsets)
- **Copy edit explore**
    - Pass by value; usually when memory constraints aren't an issue
    - Implicit unchoose step by virtue of making edits to copy
    - E.g. Building up a string over time

Three use cases for backtracking

1. Generate/count all solutions (enumeration)
2. Find one solution (or prove existence)
3. Pick one best solution

General examples of things you can do:

- Permutations
- Subsets
- Combinations
- etc.

Solving backtracking recursion problems

- Which of our three use cases does our problem fall into? (generate/count all solutions, find one solution/prove its existence, pick one best solution)
- What are we building up as our “many possibilities” in order to find our solution? (subsets, permutations, combinations, or something else)

- What’s the provided function prototype and requirements? Do we need a helper function?
  - What are we returning as our solution? (a boolean, a final value, a set of results, etc.)
  - Do we care about returning or keeping track of the path we took to get to our solution? If yes, what parameters are we already given and what others might be useful?

- What are our base and recursive cases?
  - What does my decision tree look like? (decisions, options, what to keep track of)
  - In addition to what we’re building up, are there any additional constraints on our solutions?
  - Does it make sense to use choose/explore/undo OR copy/edit/recurse for the recursion? (Note: In some very complex problems, it might be some combination of the two.)

Problems you’ve seen

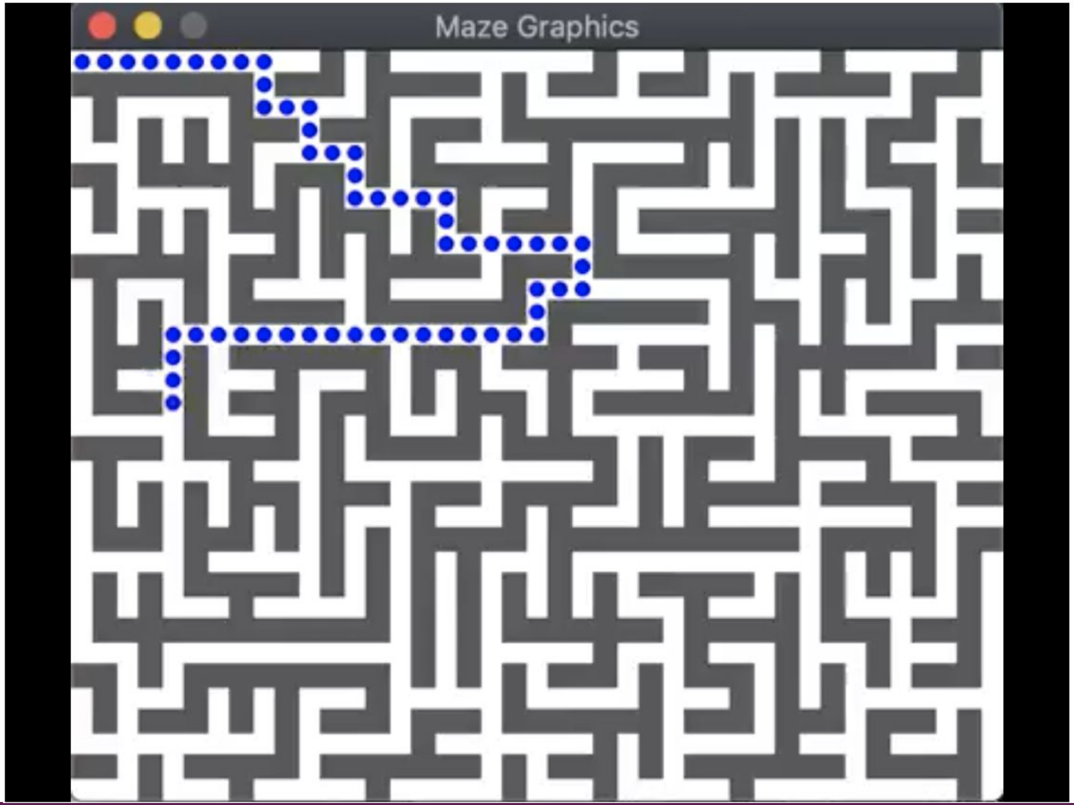
- Generating coin flip sequences
- Word scramble permutations
- Shrinkable words
- Subsets of graders
- Combinations of graders
- Solving a maze

Problems you've seen

- Generating coin flip sequences
- Word scramble permutations
- Shrinkable words
- Subsets of graders
- **Combinations of graders**
- Solving a maze

Check out a student's solution on Ed that combines recursion and iteration to solve this problem without a helper!

Solving mazes with depth-first search (DFS)



BFS vs. DFS comparison

What if we don’t unchoose?

BFS vs. DFS summary

- BFS is typically iterative while DFS is naturally expressed recursively.
- Although DFS is faster in this particular case, which search strategy to use depends on the problem you're solving.
- BFS looks at all paths of a particular length before moving on to longer paths, so it's guaranteed to find the shortest path (e.g. word ladder)!
- DFS doesn't need to store all partial paths along the way, so it has a smaller memory footprint than BFS does.

But recursion is memory intensive for the compiler

- Because a stack frame gets created for every recursive call, recursion can be memory intensive for your compiler. You'll get to experience this in A3!
    - The Big-O of DFS and BFS are the same when we consider the worst case (which is what we care about in this class), but it gets more complicated when we think about Big-O on average.
    - There are also different trade-offs: BFS can be expensive due to storing many copies of ADTs.

But recursion is memory intensive for the compiler

- Because a stack frame gets created for every recursive call, recursion can be memory intensive for your compiler. You'll get to experience this in A3!
    - The Big-O of DFS and BFS are the same when we consider the worst case (which is what we care about in this class), but it gets more complicated when we think about Big-O on average.
    - There are also different trade-offs: BFS can be expensive due to storing many copies of ADTs.

- Recursion is a powerful tool for understanding data structures and algorithms, especially in fields like artificial intelligence and systems design and programming languages.

But recursion is memory intensive for the compiler

- Because a stack frame gets created for every recursive call, recursion can be memory intensive for your compiler. You'll get to experience this in A3!
    - The Big-O of DFS and BFS are the same when we consider the worst case (which is what we care about in this class), but it gets more complicated when we think about Big-O on average.
    - There are also different trade-offs: BFS can be expensive due to storing many copies of ADTs.

- Recursion is a powerful tool for understanding data structures and algorithms, especially in fields like artificial intelligence and systems design and programming languages.

- But it often can't be used in scenarios that require you to handle large amounts of data (without some sort of added optimizations).

Where are we now?

classes

object-oriented programming

abstract data structures
(vectors, maps, etc.)

arrays

dynamic memory
management

linked data structures

testing

algorithmic analysis

recursive problem-solving

classes

object-oriented programming

abstract data structures
(vectors, maps, etc.) ✓

arrays

dynamic memory
management

linked data structures

testing ✓ *algorithmic analysis* ✓ *recursive problem-solving* ✓

classes
object-oriented programming

This is our abstraction boundary!

abstract data structures
(vectors, maps, etc.)

arrays

dynamic memory management

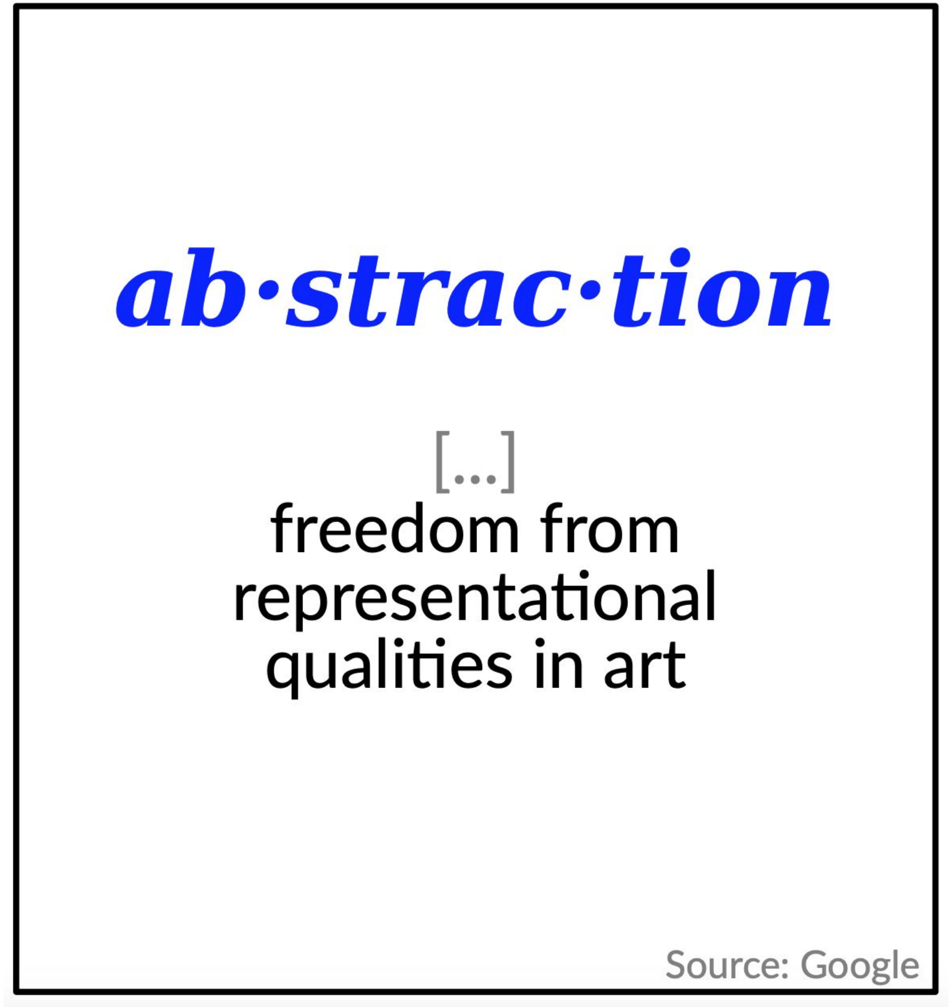
linked data structures

testing

algorithmic analysis

recursive problem-solving

Revisiting abstraction

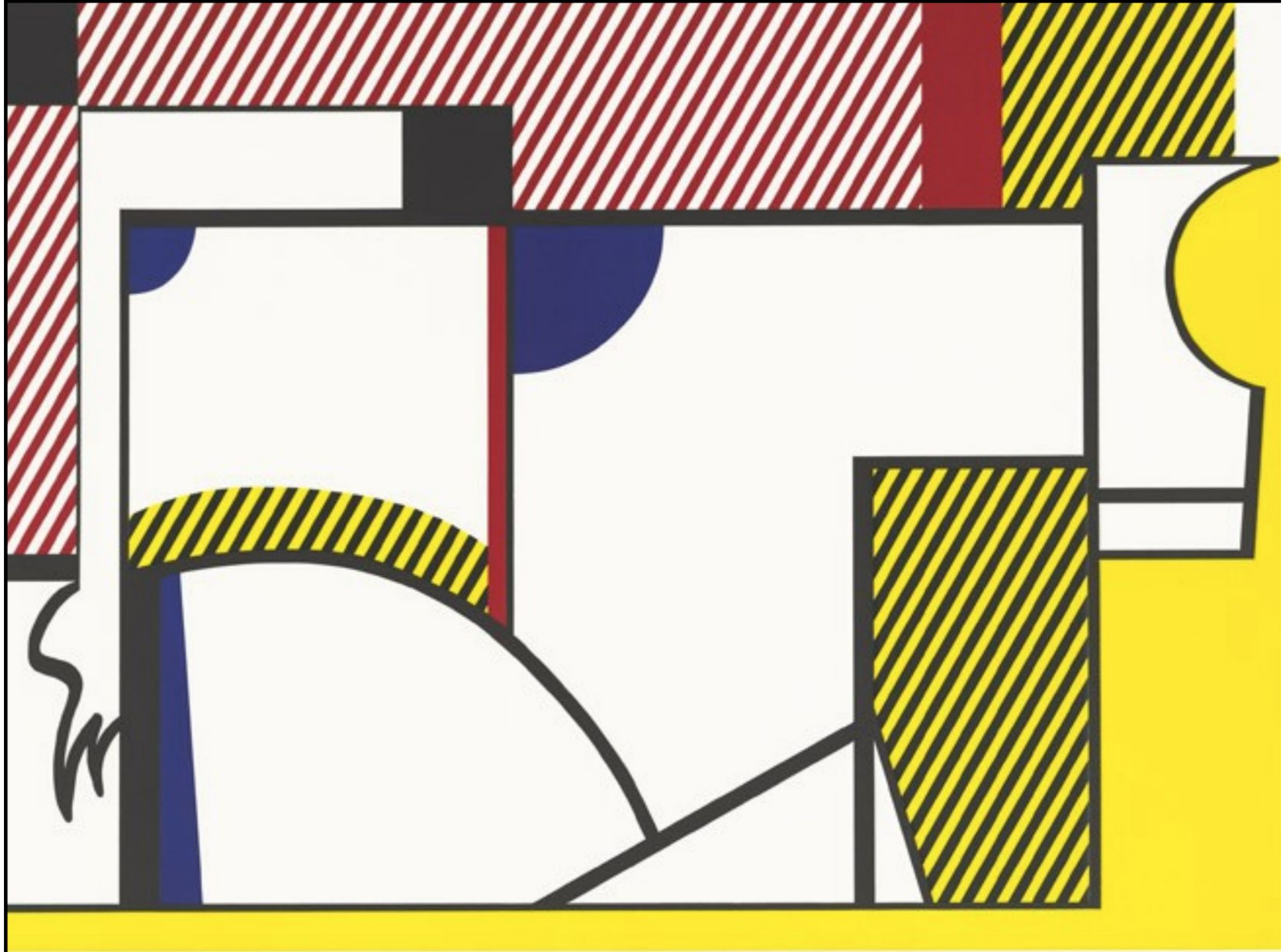


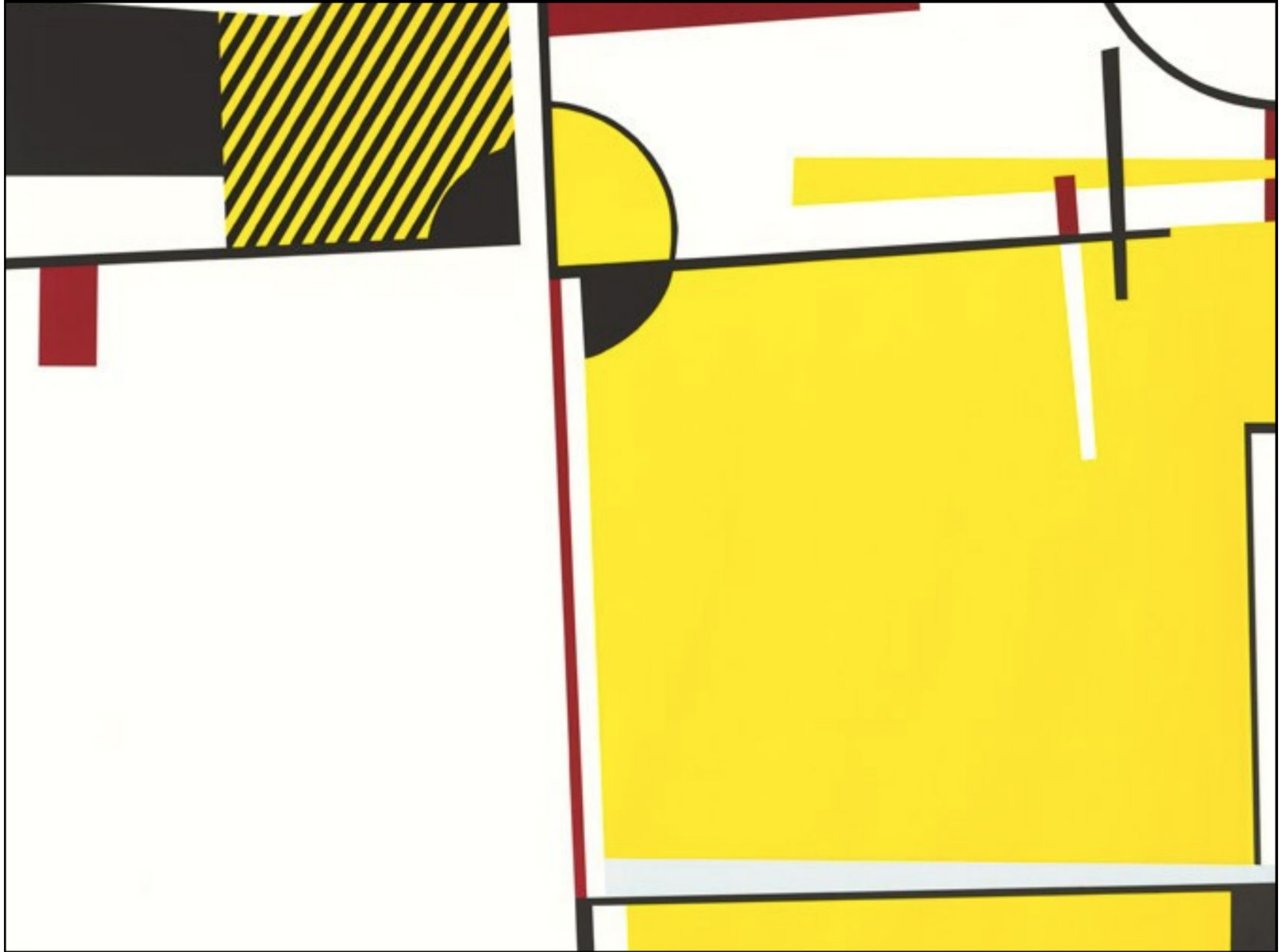
ab·strac·tion

[...]
freedom from representational qualities in art

Source: Google

Example demonstration borrowed from Keith Schwarz





Definition

abstraction

Design that hides the details of how something works while still allowing the user to access complex functionality

Definition

abstraction
Design that hides the details of how something works while still allowing the user to access complex functionality

What is a class?

Definition

class

A class defines a new data type for our programs to use.

Classes help us create types of objects - which is why we call this object-oriented programming!

Definition

class

A class defines a new data type for our programs to use.

Definition

class

A class defines a new data type for our programs to use.

This sounds familiar...

Remember structs?

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

struct Course {
    string name;
    string teacher;
    int numStudents;
};
```

Remember structs?

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

struct Course {
    string name;
    string teacher;
    int numStudents;
};
```

Definition

struct
A way to bundle different types of information in C++ – like creating a custom data structure.

Then what's the difference between a class and a struct?

Remember structs?

```
GridLocation chosen;
cout << chosen.row << endl;
cout << chosen.col << endl;
```

```
Grid<int> board(3, 3);
cout << board.numRows() << endl;
cout << board.numCols() << endl;
```

What's the difference in how you use a GridLocation vs. a Grid?

Remember structs?

```
GridLocation chosen;
cout << chosen.row << endl;
cout << chosen.col << endl;

chosen.row = 3;
chosen.col = 4;
```

```
Grid<int> board(3, 3);
cout << board.numRows() << endl;
cout << board.numCols() << endl;

board.numRows = 5;
board.numCols = 4;
```

What's the difference in how you use a GridLocation vs. a Grid?

Remember structs?

```
GridLocation chosen;
cout << chosen.row << endl;
cout << chosen.col << endl;

chosen.row = 3;
chosen.col = 4;
```

```
Grid<int> board(3, 3);
cout << board.numRows() << endl;
cout << board.numCols() << endl;

board.numRows = 5;
board.numCols = 4;
```

We don't have direct access to Grid's number of rows and number of columns!

Remember structs?

```
GridLocation chosen;
cout << chosen.row << endl;
cout << chosen.col << endl;

chosen.row = 3;
chosen.col = 4;
```

```
Grid<int> board(3, 3);
cout << board.numRows() << endl;
cout << board.numCols() << endl;

board.resize(5, 4);
```

We have to use a function that allows us to adjust those properties instead.

Remember structs?

```
GridLocation chosen;
cout << chosen.row << endl;
cout << chosen.col << endl;

chosen.row = 3;
chosen.col = 4;
```

```
Grid<int> board(3, 3);
cout << board.numRows() << endl;
cout << board.numCols() << endl;

board.resize(5, 4);
```

Why?

We have to use a function that allows us to adjust those properties instead.

Definition

encapsulation

The process of grouping related information and relevant functions into one unit and defining where that information is accessible

If we have time at the end of class, we'll look at an example of why this matters!

Definition

encapsulation
The process of grouping related information and relevant functions into one unit and defining where that information is accessible

What is a class?

- Examples of classes we've already seen: **`Vector`**s, **`Map`**s, **`Stack`**s, **`Queue`**s

What is a class?

- Examples of classes we've already seen: **`Vector`**s, **`Map`**s, **`Stack`**s, **`Queue`**s

- Every class has two parts:
  - an **interface** specifying what operations can be performed on instances of the class (this defines the abstraction boundary)
  - an **implementation** specifying how those operations are to be performed

What is a class?

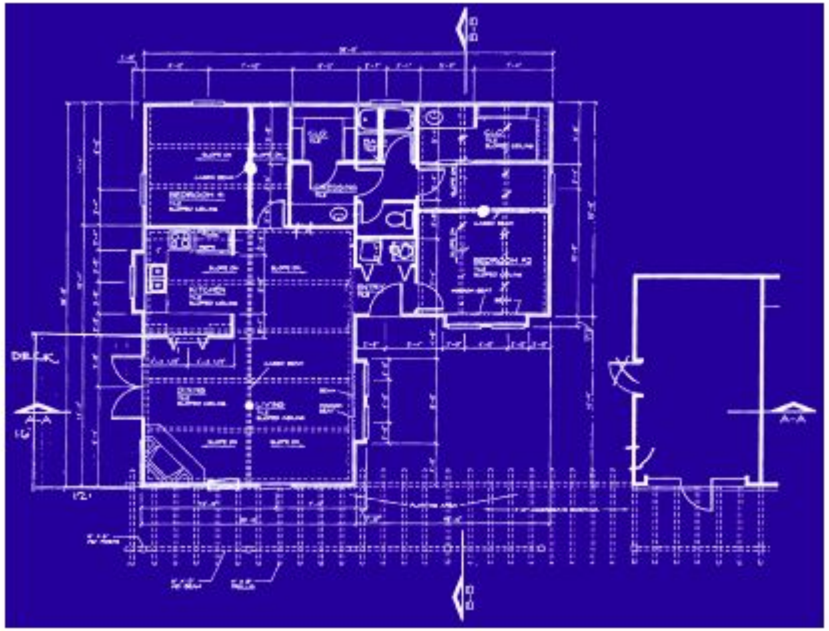
- Examples of classes we've already seen: **`Vector`**s, **`Map`**s, **`Stack`**s, **`Queue`**s

- Every class has two parts:
  - an **interface** specifying what operations can be performed on instances of the class (this defines the abstraction boundary)
  - an **implementation** specifying how those operations are to be performed

- The only difference between structs + classes are the **encapsulation** defaults.
  - A struct defaults to **public** members (accessible outside the class itself).
  - A class defaults to **private** members (accessible only inside the class implementation).

Another way to think about classes…

- A blueprint for a new type of C++ **object**!



Another way to think about classes...

- A blueprint for a new type of C++ **object**!
  - The blueprint describes a general structure, and we can create specific **instances** of our class using this structure.

Another way to think about classes...

- A blueprint for a new type of C++ **object**!
  - The blueprint describes a general structure, and we can create specific **instances** of our class using this structure.

Definition

instance

When we create an object that is our new type, we call this creating an instance of our class.

Another way to think about classes...

- A blueprint for a new type of C++ **object**!
  - The blueprint describes a general structure, and we can create specific **instances** of our class using this structure.

$$\underbrace{\texttt{Vector<int> vec;}}$$

Creates an **instance** of the Vector **class**
(i.e. an object of the type Vector)

How do we design C++ classes?

Three main parts

- Member variables
- Member functions (methods)
- Constructor

Three main parts

- Member variables
  - These are the variables stored within the class
  - Usually not accessible outside the class implementation

- Member functions (methods)

- Constructor

Three main parts

- Member variables

- Member functions (methods)
    - Functions you can call on the object
    - E.g. **`vec.add()`**, **`vec.size()`**, **`vec.remove()`**, etc.

- Constructor

Three main parts

- Member variables
- Member functions (methods)
- Constructor
  - Gets called when you create the object
  - E.g. **`Vector<int> vec;`**

Three main parts

- Member variables
    - These are the variables stored within the class
    - Usually not accessible outside the class implementation

- Member functions (methods)
    - Functions you can call on the object
    - E.g. **`vec.add()`**, **`vec.size()`**, **`vec.remove()`**, etc.

- Constructor
    - Gets called when you create the object
    - E.g. **`Vector<int> vec;`**

How do we design a class?

We must specify the 3 parts:

1. Member variables: *What subvariables make up this new variable type?*

2. Member functions: *What functions can you call on a variable of this type?*

3. Constructor: *What happens when you make a new instance of this type?*

How do we design a class?

We must specify the 3 parts:

1. Member variables: *What subvariables make up this new variable type?*

2. Member functions: *What functions can you call on a variable of this type?*

3. Constructor: *What happens when you make a new instance of this type?*

In general, classes are useful in helping us with complex programs where information can be grouped into objects.

Design activity

How would you design a class for...

- A bank account that enables transferring funds between accounts
- A Spotify (or other music platform) playlist

We must specify the 3 parts:

1. Member variables: *What subvariables make up this new variable type?*
2. Member functions: *What functions can you call on a variable of this type?*
3. Constructor: *What happens when you make a new instance of this type?*

Attendance ticket:

https://tinyurl.com/designClassesOOP

Please don't send this link to students who are not here. It's on your honor!

How would you design a class for...

- A bank account that enables transferring funds between accounts
- A Spotify (or other music platform) playlist

We must specify the 3 parts:

1. Member variables: *What subvariables make up this new variable type?*
2. Member functions: *What functions can you call on a variable of this type?*
3. Constructor: *What happens when you make a new instance of this type?*

Announcements

Announcements

- Assignment 3 is due next Tuesday at 11:59pm PDT. The grace period ends Wednesday at 11:59pm PDT.

- Assignment 2 revisions will be due Friday, July 22 at 11:59pm PDT.

- Midterm grades will be released early next week, and you'll be able to schedule a check-in with your SL either next week or the following.

- The final project proposals are due Sunday, July 24.
  - Come chat with us about what you're interested in at Office Hours!

How do we write classes in C++?

Random Bags

Random Bags

- A **random bag** is a data structure similar to a stack or queue. It supports two operations:
    - **add**, which puts an element into the random bag, and
    - **remove random**, which returns and removes a random element from the bag.

Random Bags

- A **random bag** is a data structure similar to a stack or queue. It supports two operations:
  - **add**, which puts an element into the random bag, and
  - **remove random**, which returns and removes a random element from the bag.

- Random bags have a number of applications:
  - Simpler: Shuffling a deck of cards.
  - More advanced: Generating artwork, designing mazes, and training self-driving cars to park and change lanes!

Random Bags

- A **random bag** is a data structure similar to a stack or queue. It supports two operations:
  - **add**, which puts an element into the random bag, and
  - **remove random**, which returns and removes a random element from the bag.

- Random bags have a number of applications:
  - Simpler: Shuffling a deck of cards.
  - More advanced: Generating artwork, designing mazes, and training self-driving cars to park and change lanes.

- Let’s go create our own custom **`RandomBag`** type!

Creating our own class

Classes in C++

- Defining a class in C++ (typically) requires two steps:

Classes in C++

- Defining a class in C++ (typically) requires two steps:
    - Create a **header file** (typically suffixed with `.h`) describing what operations the class can perform and what internal state it needs.

Classes in C++

- Defining a class in C++ (typically) requires two steps:
    - Create a **header file** (typically suffixed with `.h`) describing what operations the class can perform and what internal state it needs.
    - Create an **implementation file** (typically suffixed with `.cpp`) that contains the implementation of the class.

Classes in C++

- Defining a class in C++ (typically) requires two steps:
  - Create a **header file** (typically suffixed with **`.h`**) describing what operations the class can perform and what internal state it needs.
  - Create an **implementation file** (typically suffixed with **`.cpp`**) that contains the implementation of the class.

- Clients of the class can then include (using the **`#include`** directive) the header file to use the class.

Header files

What's in a header?

What's in a header?

```
#pragma once
```

This boilerplate code is called a preprocessor directive. It's used to make sure weird things don't happen if you include the same header twice.

Curious how it works? Come ask us after class!

What's in a header?

```
#pragma once

class RandomBag {



};
```

This is a *class definition*. We're creating a new class called **RandomBag**. Like a **struct**, this defines the name of a new type that we can use in our programs.

What's in a header?

```
#pragma once

class RandomBag {




};
```

Don't forget to add the semicolon!

You'll run into some scary compiler errors if you leave it out!

What's in a header?

```
#pragma once

class RandomBag {
public:




private:


};
```

What's in a header?

```
#pragma once

class RandomBag {
public:




private:

};
```

The *public interface* specifies what functions you can call on objects of this type. (i.e. its methods)

Think things like the `Vector` `.add()` function or the `string`'s `.find()`.

What's in a header?

```
#pragma once

class RandomBag {
public:




private:

};
```

The *public interface* specifies what functions you can call on objects of this type. (i.e. its methods)

Think things like the `Vector` `.add()` function or the `string`'s `.find()`.

The *private implementation* contains information that objects of this class type will need in order to do their job properly. This is invisible to people using the class.

What's in a header?

```
#pragma once

class RandomBag {
public:
   void add(int value);
   int removeRandom();


private:

};
```

These are member functions of the **RandomBag** *class. They're functions you can call on objects of type* **RandomBag**.

All member functions must be defined in the class definition. We'll implement these functions in the C++ file.

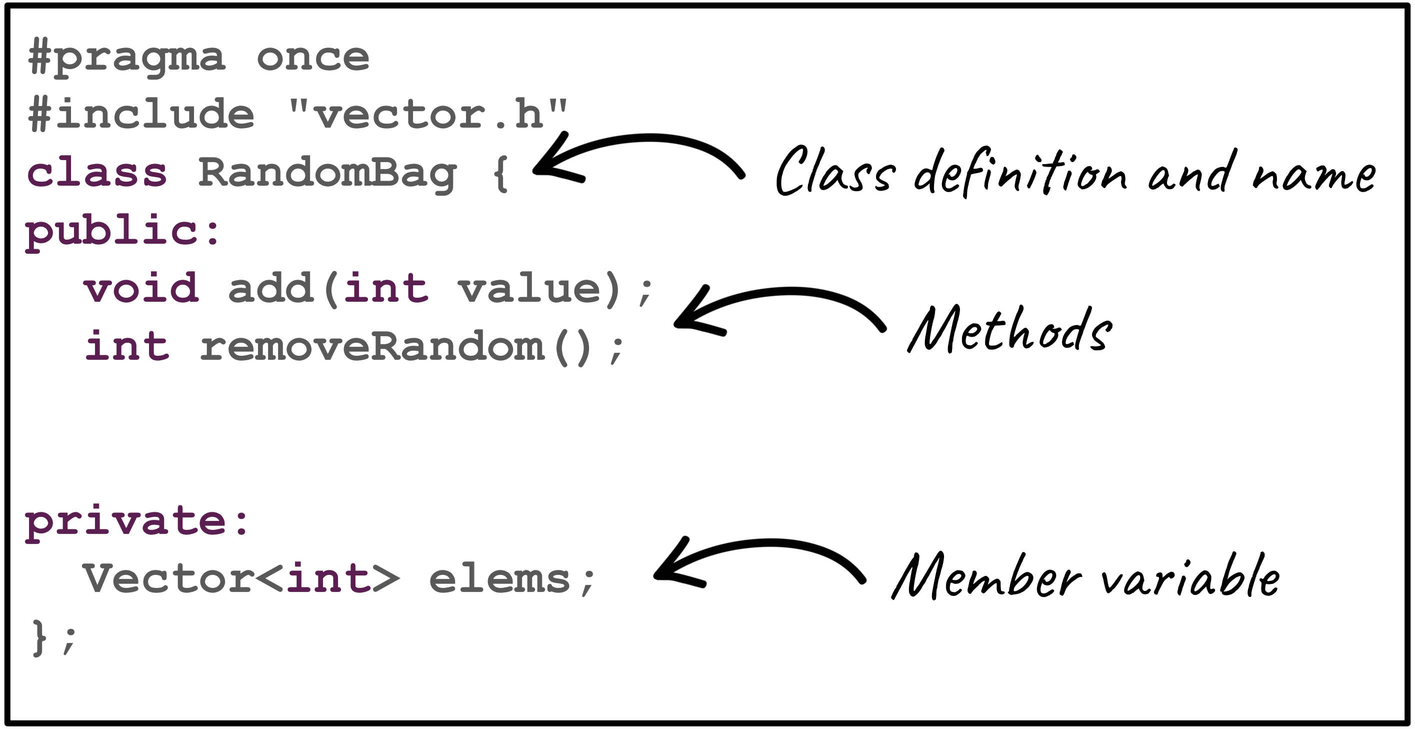
What's in a header?

```
#pragma once
#include "vector.h"
class RandomBag {
public:
   void add(int value);
   int removeRandom();


private:
   Vector<int> elems;
};
```

This is a data member of the class. This tells us how the class is implemented. Internally, we're going to store a `Vector<int>` holding all the elements. The only code that can access or touch this `Vector` is the `RandomBag` implementation.

Header summary



```
#pragma once
#include "vector.h"
class RandomBag {
public:
   void add(int value);
   int removeRandom();


private:
   Vector<int> elems;
};
```

Header summary

```
#pragma once
#include "vector.h"
class RandomBag {
public:
  void add(int value);
  int removeRandom();


private:
  Vector<int> elems;
};
```

Implementation files

`RandomBag.cpp`

```
#include "RandomBag.h"
```

```
#include "RandomBag.h"
```

If we're going to implement the `RandomBag` type, the `.cpp` file needs to have the class definition available. All implementation files need to include the relevant headers.

```
#include "RandomBag.h"
```

If we're going to implement the `RandomBag` type, the `.cpp` file needs to have the class definition available. All implementation files need to include the relevant headers.

```
#pragma once
#include "vector.h"
class RandomBag {
public:
  void add(int value);
  int removeRandom();


private:
  Vector<int> elems;
};
```

```
#include "RandomBag.h"
```

```
#pragma once
#include "vector.h"
class RandomBag {
public:
  void add(int value);
  int removeRandom();


private:
  Vector<int> elems;
};
```

```
#include "RandomBag.h"

void RandomBag::add(int value){
    elems.add(value);
}
```

```
#pragma once
#include "vector.h"
class RandomBag {
public:
  void add(int value);
  int removeRandom();


private:
  Vector<int> elems;
};
```

```
#include "RandomBag.h"

void RandomBag::add(int value){
    elems.add(value);
}
```

The syntax `RandomBag::add` means "the add function defined inside of `RandomBag`." The :: operator is called the scope resolution operator in C++ and is used to say where to look for things.

```
#pragma once
#include "vector.h"
class RandomBag {
public:
  void add(int value);
  int removeRandom();


private:
  Vector<int> elems;
};
```

```
#include "RandomBag.h"

void ~~RandomBag::~~add(int value){
     elems.add(value);
}
```

If we had written something like this instead, then the compiler would think we were just making a free function named **add** *that has nothing to do with* **RandomBag's** *version of* **add**. *That's an easy mistake to make!*

```
#pragma once
#include "vector.h"
class RandomBag {
public:
  void add(int value);
  int removeRandom();


private:
  Vector<int> elems;
};
```

```
#include "RandomBag.h"

void RandomBag::add(int value){
     elems.add(value);
}
```

We don't need to specify where **elems** *is. The compiler knows that we're inside* **RandomBag***, and so it knows that this means "the current* **RandomBag***'s collection of elements." Using the scope resolution operator is like passing in an invisible parameter to the function to indicate what the current instance is.*

```
#include "vector.h"
class RandomBag {
public:
  void add(int value);
  int removeRandom();


private:
  Vector<int> elems;
};
```

```cpp
#include "RandomBag.h"

void RandomBag::add(int value){
    elems.add(value);
}

int RandomBag::removeRandom() {
    if (elems.isEmpty()) {
        error("Aaaaahhh!");
    }
    int index = randomInteger(0, elems.size() - 1);
    int result = elems[index];
    elems.remove(index);
    return result;
}
```

```cpp
#pragma once
#include "vector.h"
class RandomBag {
public:
  void add(int value);
  int removeRandom();


private:
  Vector<int> elems;
};
```

```
#include "RandomBag.h"

void RandomBag::add(int value){
     elems.add(value);
}

int RandomBag::removeRandom() {
     if (elems.isEmpty()) {
          error("Aaaaahhh!");
     }
     int index = randomInteger(0, elems.size() - 1);
     int result = elems[index];
     elems.remove(index);
     return result;
}
```

```
#pragma once
#include "vector.h"
class RandomBag {
public:
  void add(int value);
  int removeRandom();
  int size();
  bool isEmpty();
private:
  Vector<int> elems;
};
```

```
#include "RandomBag.h"

void RandomBag::add(int value){
    elems.add(value);
}

int RandomBag::removeRandom() {
    if (elems.isEmpty()) {
        error("Aaaaahhh!");
    }
    int index = randomInteger(0, elems.size() - 1);
    int result = elems[index];
    elems.remove(index);
    return result;
}

int RandomBag::size() {
    return elems.size();
}
```

```
#pragma once
#include "vector.h"
class RandomBag {
public:
  void add(int value);
  int removeRandom();
  int size();
  bool isEmpty();
private:
  Vector<int> elems;
};
```

```cpp
#include "RandomBag.h"

void RandomBag::add(int value){
    elems.add(value);
}

int RandomBag::removeRandom() {
    if (elems.isEmpty()) {
        error("Aaaaahhh!");
    }
    int index = randomInteger(0, elems.size() - 1);
    int result = elems[index];
    elems.remove(index);
    return result;
}

int RandomBag::size() {
    return elems.size();
}

bool RandomBag::isEmpty() {
    return size() == 0;
}
```

```cpp
#pragma once
#include "vector.h"
class RandomBag {
public:
  void add(int value);
  int removeRandom();
  int size();
  bool isEmpty();
private:
  Vector<int> elems;
};
```

```
#include "RandomBag.h"

void RandomBag::add(int value){
     elems.add(value);
}

int RandomBag::removeRandom() {
     if (elems.isEmpty()) {
          error("Aaaaahhh!");
     }
     int index = randomInteger(0, elems.size() - 1);
     int result = elems[index];
     elems.remove(index);
     return result;
}

int RandomBag::size() {
     return elems.size();
}

bool RandomBag::isEmpty() {
     return size() == 0;
}
```

This code calls our own `size()` *function. The class implementation can use the public interface.*

```
#pragma once
#include "vector.h"
class RandomBag {
public:
   void add(int value);
   int removeRandom();
   int size();
   bool isEmpty();
private:
   Vector<int> elems;
};
```

```
#include "RandomBag.h"

void RandomBag::add(int value){
     elems.add(value);
}

int RandomBag::removeRandom() {
     if (elems.isEmpty()) {
          error("Aaaaahh!");
     }
     int index = randomInteger(0, size() - 1);
     int result = elems[index];
     elems.remove(index);
     return result;
}

int RandomBag::size() {
     return elems.size();
}

bool RandomBag::isEmpty() {
     return size() == 0;
}
```

What a good idea! Let's use it up here as well.

```
#pragma once
#include "vector.h"
class RandomBag {
public:
   void add(int value);
   int removeRandom();
   int size();
   bool isEmpty();
private:
   Vector<int> elems;
};
```

```cpp
#include "RandomBag.h"

void RandomBag::add(int value){
    elems.add(value);
}

int RandomBag::removeRandom() {
    if (elems.isEmpty()) {
        error("Aaaaahhh!");
    }
    int index = randomInteger(0, size() - 1);
    int result = elems[index];
    elems.remove(index);
    return result;
}

int RandomBag::size() {
    return elems.size();
}

bool RandomBag::isEmpty() {
    return size() == 0;
}
```

This use of the `const` *keyword means "I promise that this function doesn't change the state of the object."*

```cpp
public:
   void add(int value);
   int removeRandom();
   int size() const;
   bool isEmpty() const;
private:
   Vector<int> elems;
};
```

```cpp
#include "RandomBag.h"

void RandomBag::add(int value){
     elems.add(value);
}

int RandomBag::removeRandom() {
     if (elems.isEmpty()) {
          error("Aaaaahhh!");
     }
     int ind                                 size() - 1);
     int res
     elems.r
     return
}

int RandomBag::size() const {
     return elems.size();
}

bool RandomBag::isEmpty() const {
     return size() == 0;
}
```

We have to remember to add it into the implementation as well!

```cpp
#pragma once
#include "vector.h"
class RandomBag {
public:
   void add(int value);
   int removeRandom();
   int size() const;
   bool isEmpty() const;
private:
   Vector<int> elems;
};
```

```
#include "RandomBag.h"

void RandomBag::add(int value){
    elems.add(value);
}

int RandomBag::removeRandom() {
    if (elems.isEmpty()) {
        error("Aaaaahhh!");
    }
    int index = randomInteger(0, size() - 1);
    int result = elems[index];
    elems.remove(index);
    return result;
}

int RandomBag::size() const {
    return elems.size();
}

bool RandomBag::isEmpty() const {
    return size() == 0;
}
```

Note: *There are some additional* `#include`*s that we'll need. (We'll see them in the actual .cpp file.)*

```
#pragma once
#include "vector.h"
class RandomBag {
public:
   void add(int value);
   int removeRandom();
   int size() const;
   bool isEmpty() const;
private:
   Vector<int> elems;
};
```

Using a custom class

[Qt Creator demo]

Takeaways

- Public member variables declared in the header file are automatically accessible in the **`.cpp`** file

Takeaways

- Public member variables declared in the header file are automatically accessible in the `.cpp` file
- As a best practice, member variables should be private, and you can create public member functions to allow users to edit them

Takeaways

- Public member variables declared in the header file are automatically accessible in the **`.cpp`** file

- As a best practice, member variables should be private, and you can create public member functions to allow users to edit them

- Member functions have an implicit parameter that allows them to know what instance of the class (i.e. which object) they’re operating on

Takeaways

- Public member variables declared in the header file are automatically accessible in the `.cpp` file
- As a best practice, member variables should be private, and you can create public member functions to allow users to edit them
- Member functions have an implicit parameter that allows them to know what instance of the class (i.e. which object) they're operating on
- When you don't have a constructor, there's a default, zero-argument constructor that instantiates all private member variables
    - (We'll see an explicit constructor next week!)

An example:
Structs vs. classes

[time-permitting]

Summary

Object-Oriented Programming

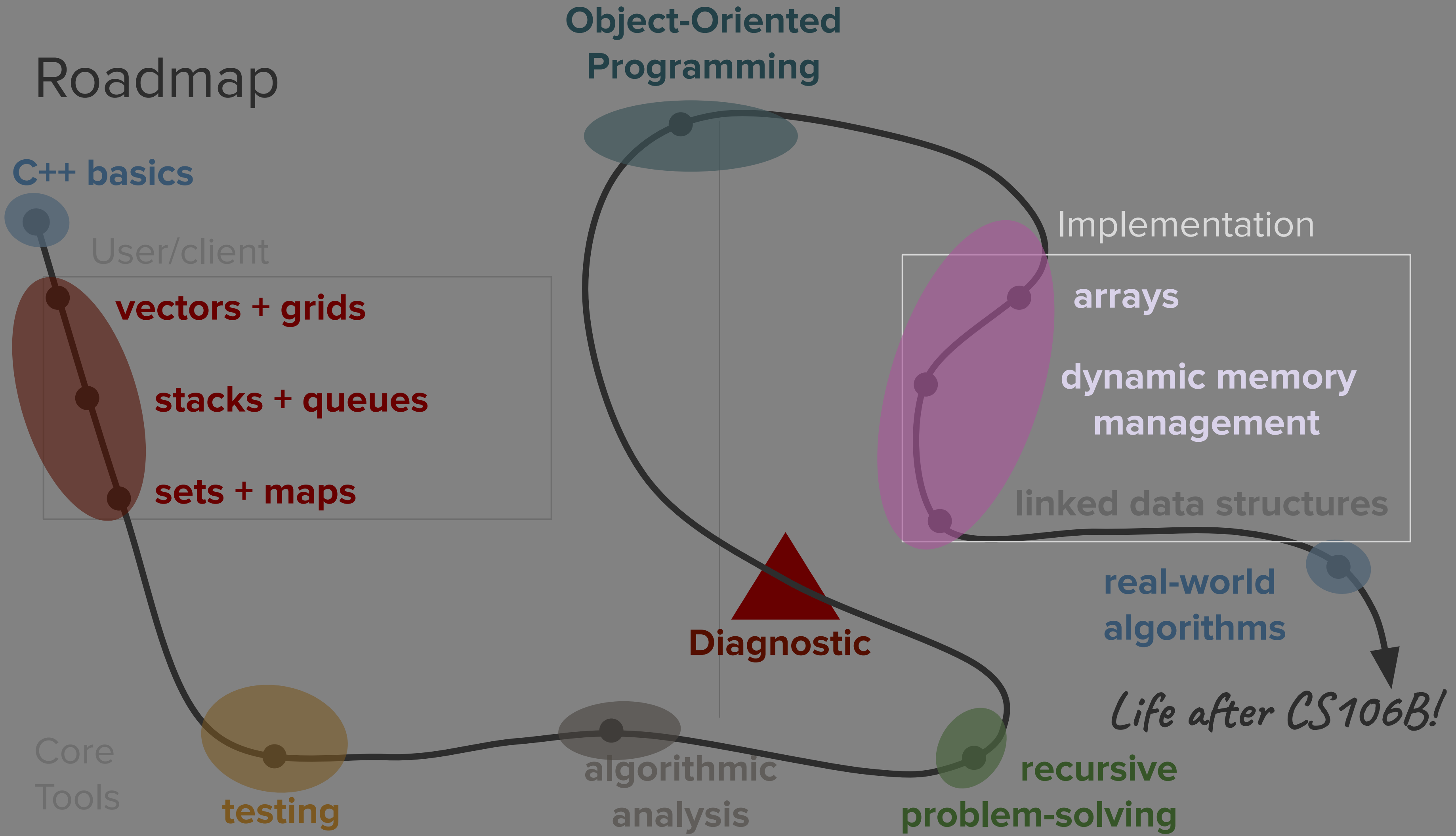
- We create our own abstractions for defining data types using classes. Classes allow us to encapsulate information in a structured way.

- Classes have three main parts to keep in mind when designing them:
  - Member variables ➔ these are always private
  - Member functions (methods) ➔ these can be private or public
  - Constructor ➔ this is created by default if you don't define one

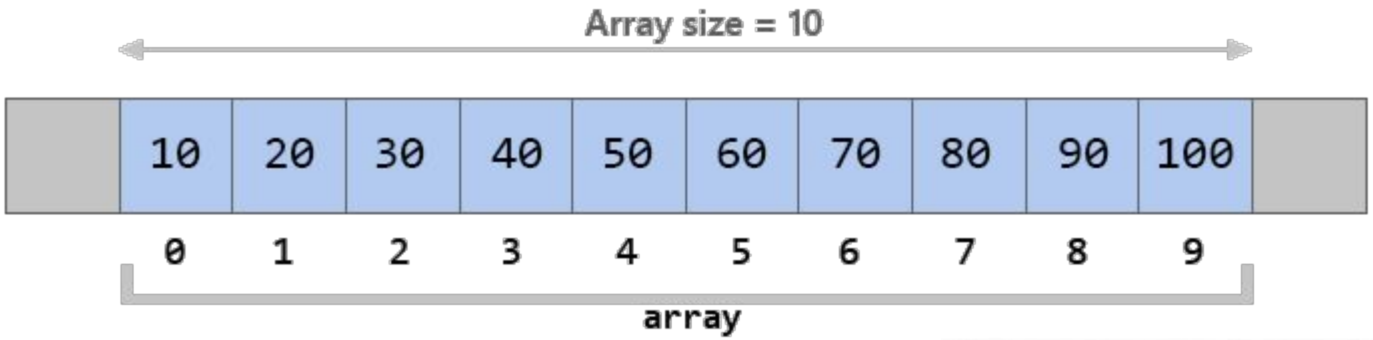
- Writing classes requires the creation of a header (`.h`) file for the interface and an implementation (`.cpp`) file.

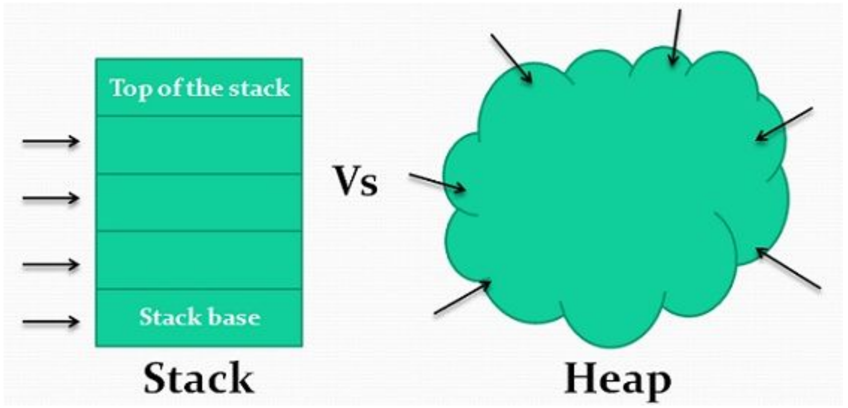
What’s next?

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

Dynamic memory and arrays


Array size = 10
10
20
30
40
50
60
70
80
90
100
0
1
2
3
4
5
6
7
8
9
array


Top of the stack
Stack base
Stack
Vs
Heap