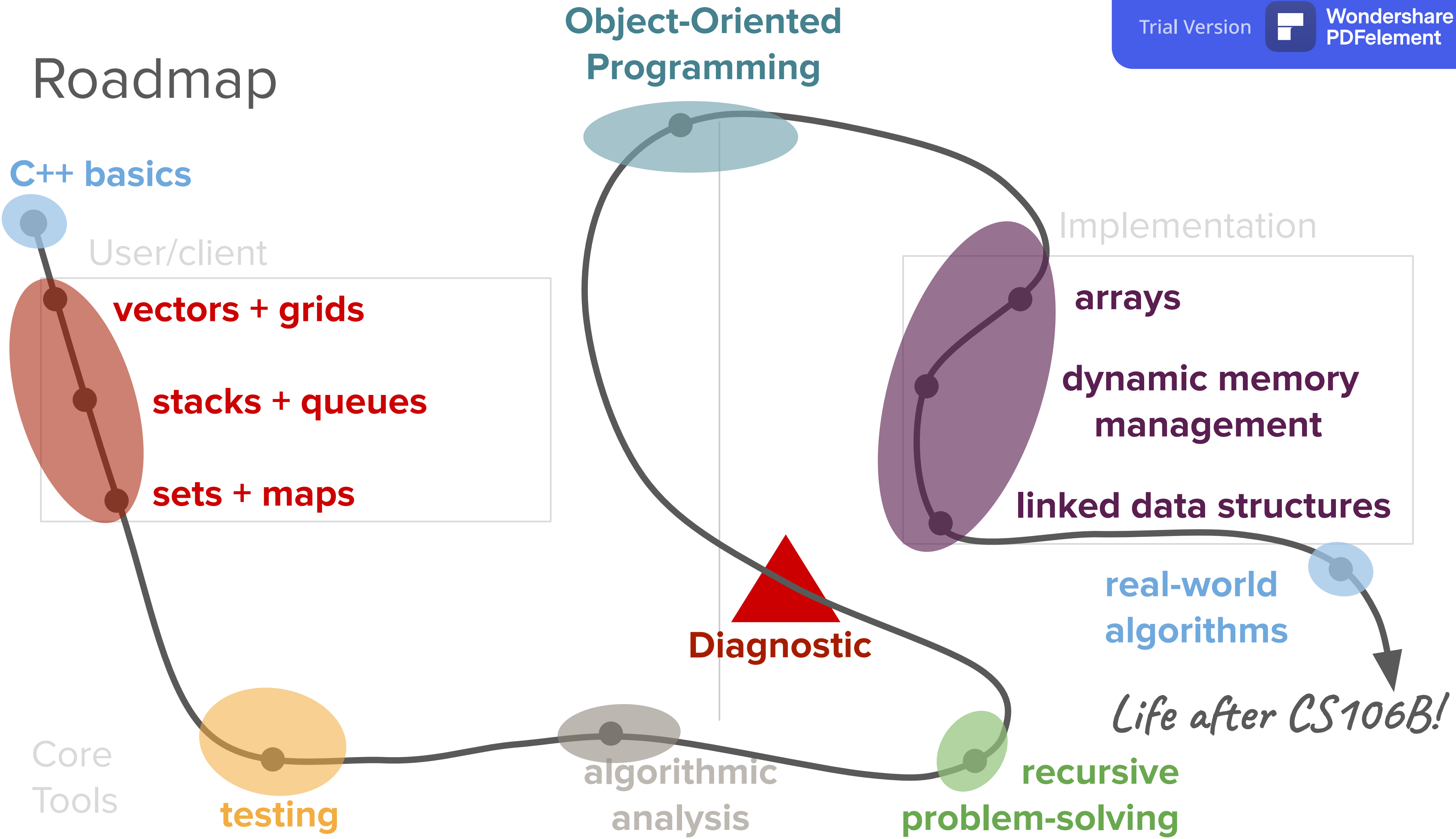
# Object-Oriented Programming

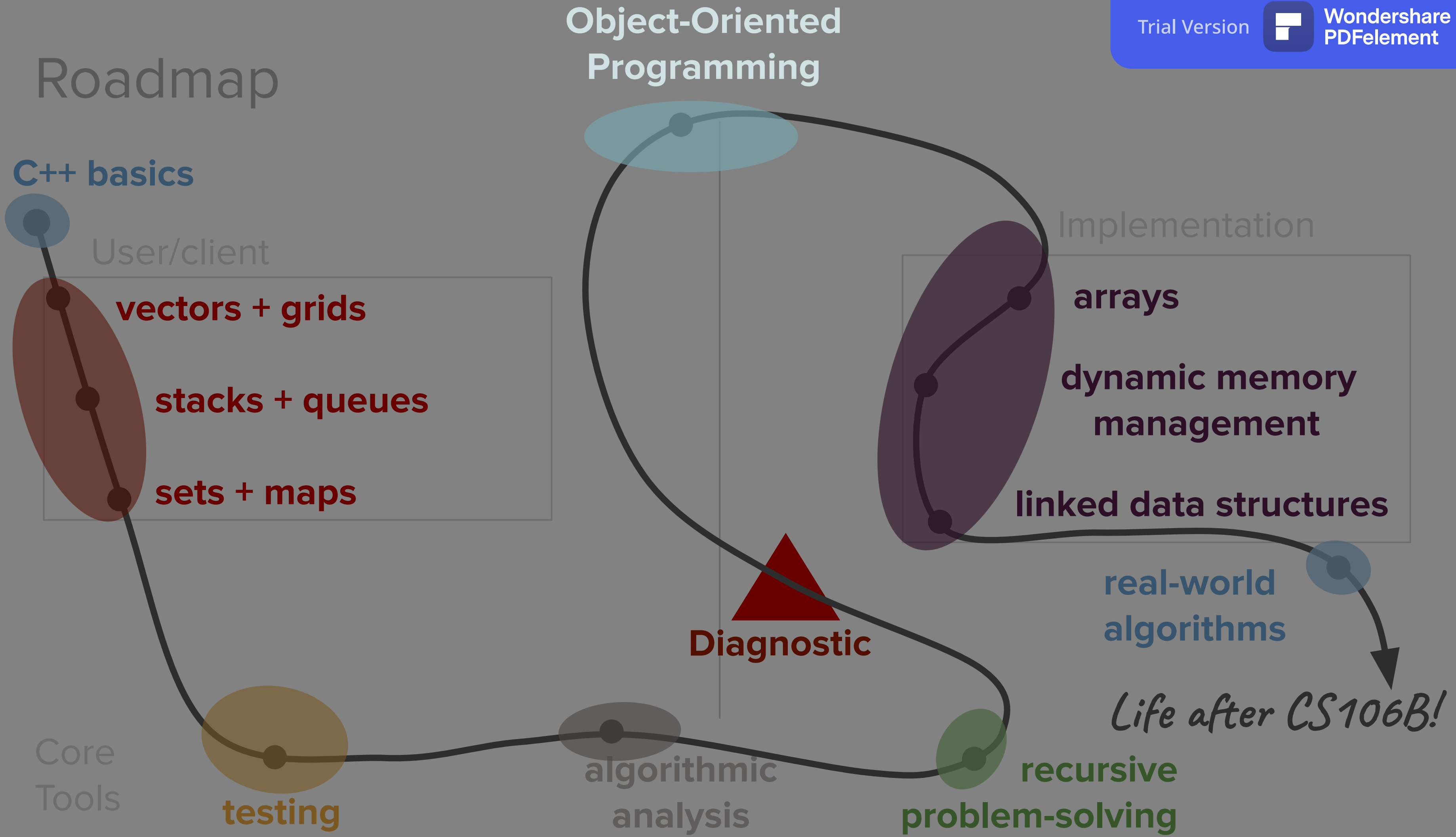
**What do you think makes a good, well-designed abstraction?**

(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

# 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

# Two types of recursion

**Basic recursion**

- One repeated task that builds up a solution as you come back up the call stack
- The final base case defines the initial seed of the solution and each call contributes a little bit to the solution
- Initial call to recursive function produces final solution

**Backtracking recursion**

- Build up many possible solutions through multiple recursive calls at each step
- Seed the initial recursive call with an “empty” solution
- At each base case, you have a potential solution

# 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.

# We've seen lots of different backtracking strategies…

Questions to ask yourself when planning your strategy:

- What does my decision tree look like? (decisions, options, what to keep track of)
- What are our base and recursive cases?
- What's the provided function prototype and requirements? Do we need a helper function?
- Do we care about returning or keeping track of the path we took to get to our solution?
- 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 returning as our solution? (a boolean, a final value, a set of results, etc.)
- What are we building up as our "many possibilities" in order to find our solution? (subsets, permutations, combinations, or something else)

# 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

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

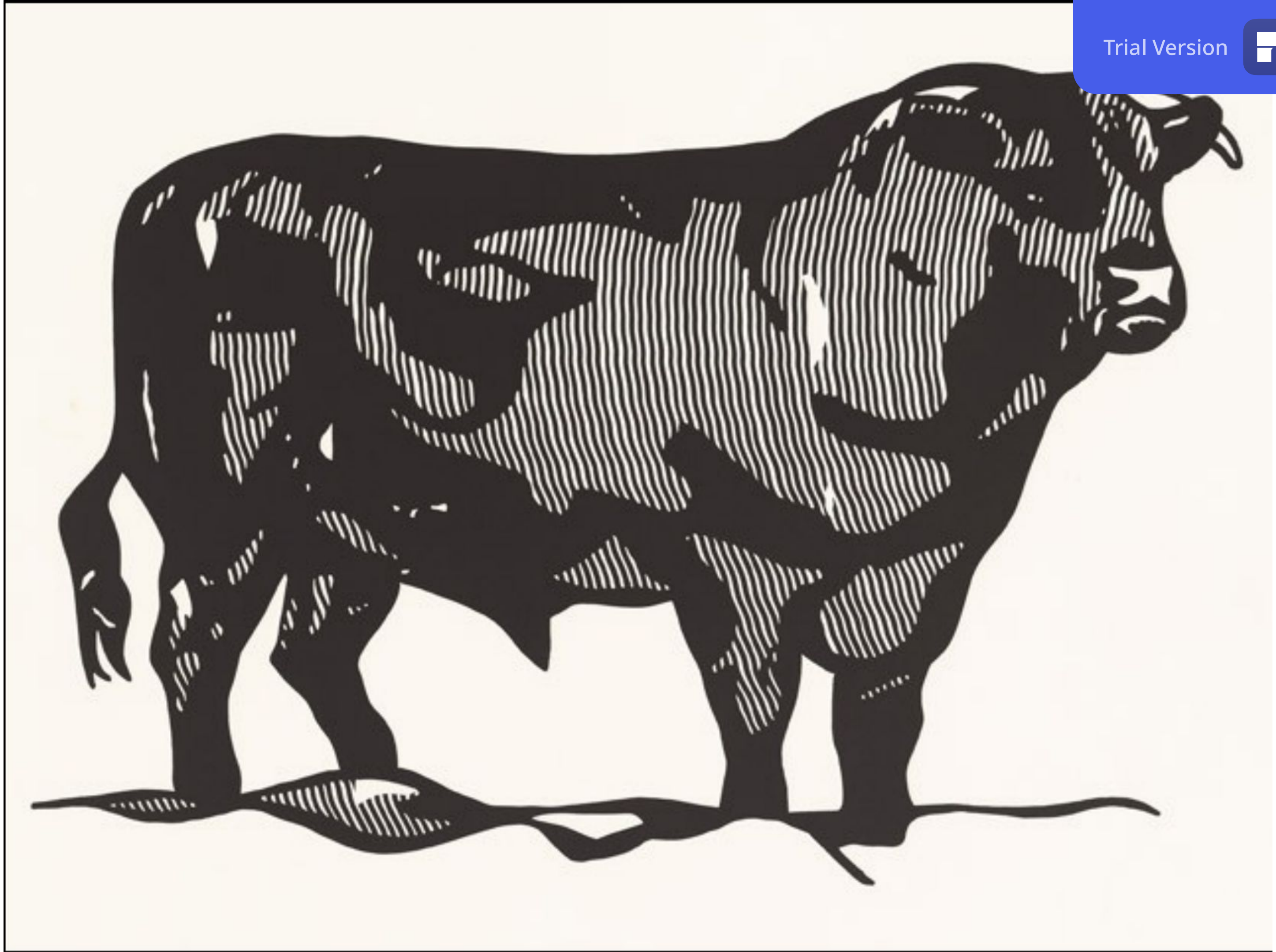
**ab·strac·tion**

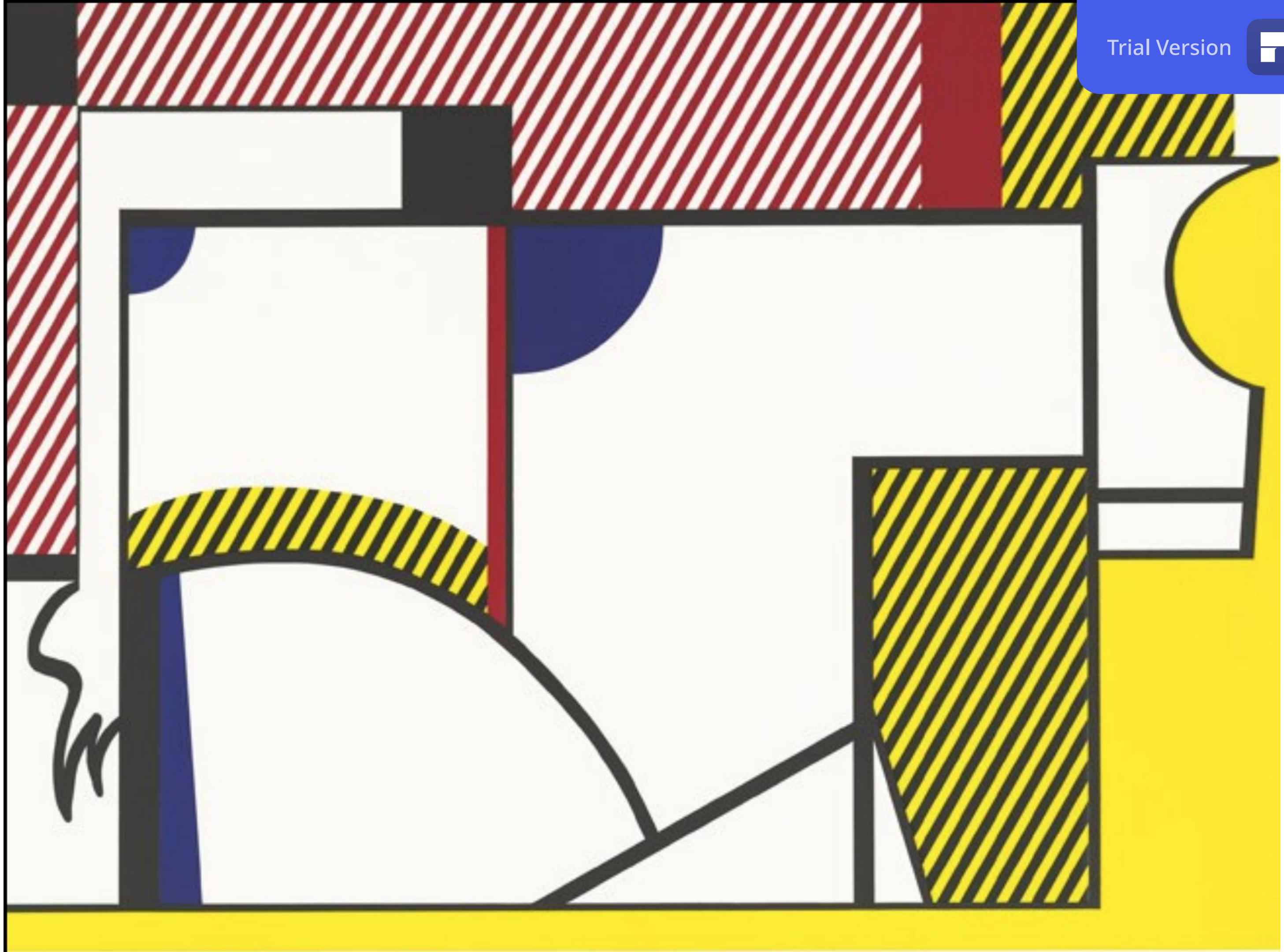
[...]

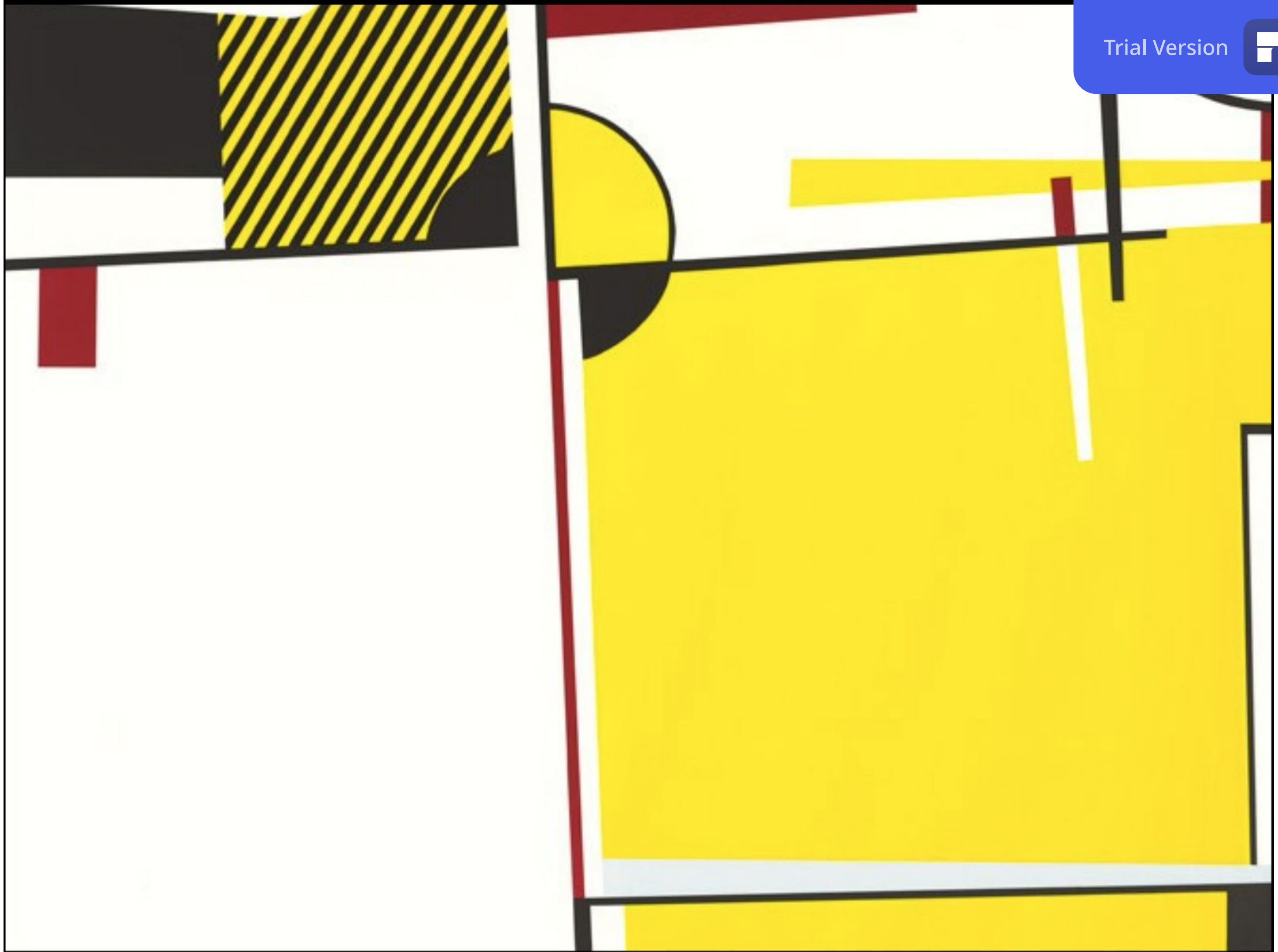
freedom from representational qualities in art

Source: Google

*Example demonstration borrowed from Keith Schwarz*







How do we accomplish this in C++? With classes!

## 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.

*This sounds familiar...*

# Remember structs?

```
struct BackpackItem {
    int survivalValue;
    int weight;
};

struct Juror {
    string name;
    int bias;
};
```

*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?

**struct**

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

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

**class**

```
GPoint origin(0, 0);
cout << origin.getX() << endl;
cout << origin.getY() << endl;

origin.x = 3;
origin.y = 4;
```

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

# Remember structs?

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

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

```
GPoint origin(0, 0);
cout << origin.getX() << endl;
cout << origin.getY() << endl;

origin.x = 3;
origin.y = 4;
```

*We don't have direct access to GPoint's x and y coordinates!*

# What is a class?

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

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

封装
- The only difference between structs + classes are the **encapsulation** defaults.
  可在类本身之外访问
  - A struct defaults to **public** members (accessible outside the class itself).
  - A class defaults to **private** members (accessible only inside the class implementation).

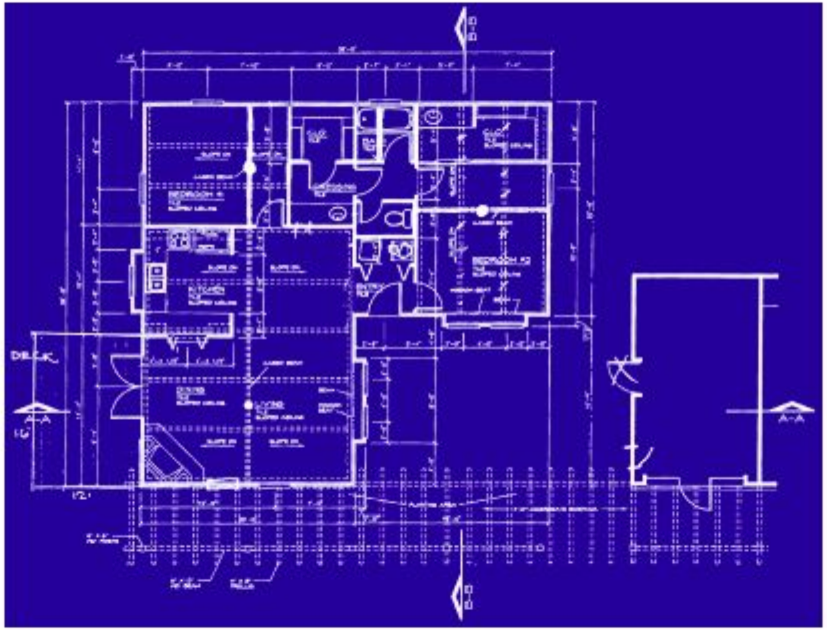
## Definition

**encapsulation**

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

# 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.

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.

```
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
  - 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?*

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

# Breakout 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?*

# Announcements

# Announcements

- The mid-quarter diagnostic is coming soon! Make sure to read through the information on the linked page if you haven't yet.
  - The link to access your personalized diagnostic access portal will be posted on the homepage of the website at 12:01am PDT Friday and will remain up until 11:59pm PDT Sunday.
- Assignment 3 is due tomorrow, **Thursday, July 16 at 11:59pm.**
- There will be a diagnostic review session hosted by Trip tomorrow night, from 7-8:30pm. The session will be recorded and made available on Canvas shortly afterwards.

# sourcegraph

## Thursday 5 PT on Twitch

Demo and Recursion Info Session: Come with all your recursion related questions! https://www.twitch.tv/sourcegraph

# 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 have a number of applications:
  - Simpler: Shuffling a deck of cards. **刷一副牌**
  - 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:
  - 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?

```
#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.

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;
};
```

Class definition and name

Methods

Member variable

# 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"
```

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"

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 解析 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."

```
#pragma once
#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!");
    }           random library in c++
    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;
}

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("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;
}
```

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;
};
```

```
#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;
}
```

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

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

This function is not goint to edit any of my private number variables.

```
#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!

```
#pragma once
#include "vector.h"
class RandomBag {
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 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;
}
```

```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;
};
```

# Using a custom class

[Qt Creator demo]

# 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 隐藏的 implicit parameter that allows them to know what object they're operating on
- When you don't have a constructor, there's a default 0 argument constructor that instantiates all private member variables
  - (We'll see an explicit constructor tomorrow!)

# **An example:**
# Structs vs. classes

[time-permitting]

# Summary

# Object-Oriented Programming

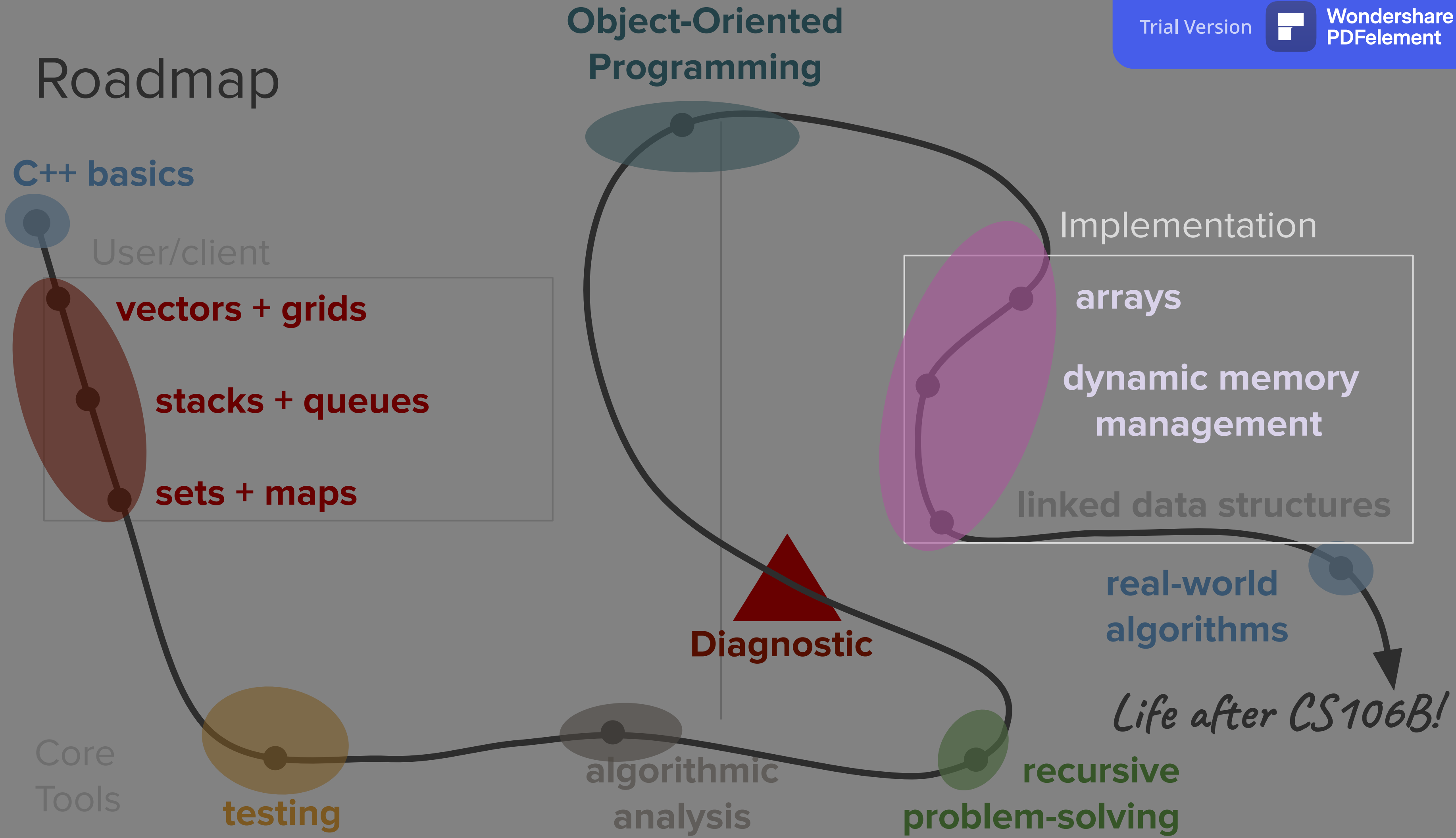
- We create our own abstractions for defining data types using classes. Classes allow us to encapsulate information in a structured way.

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

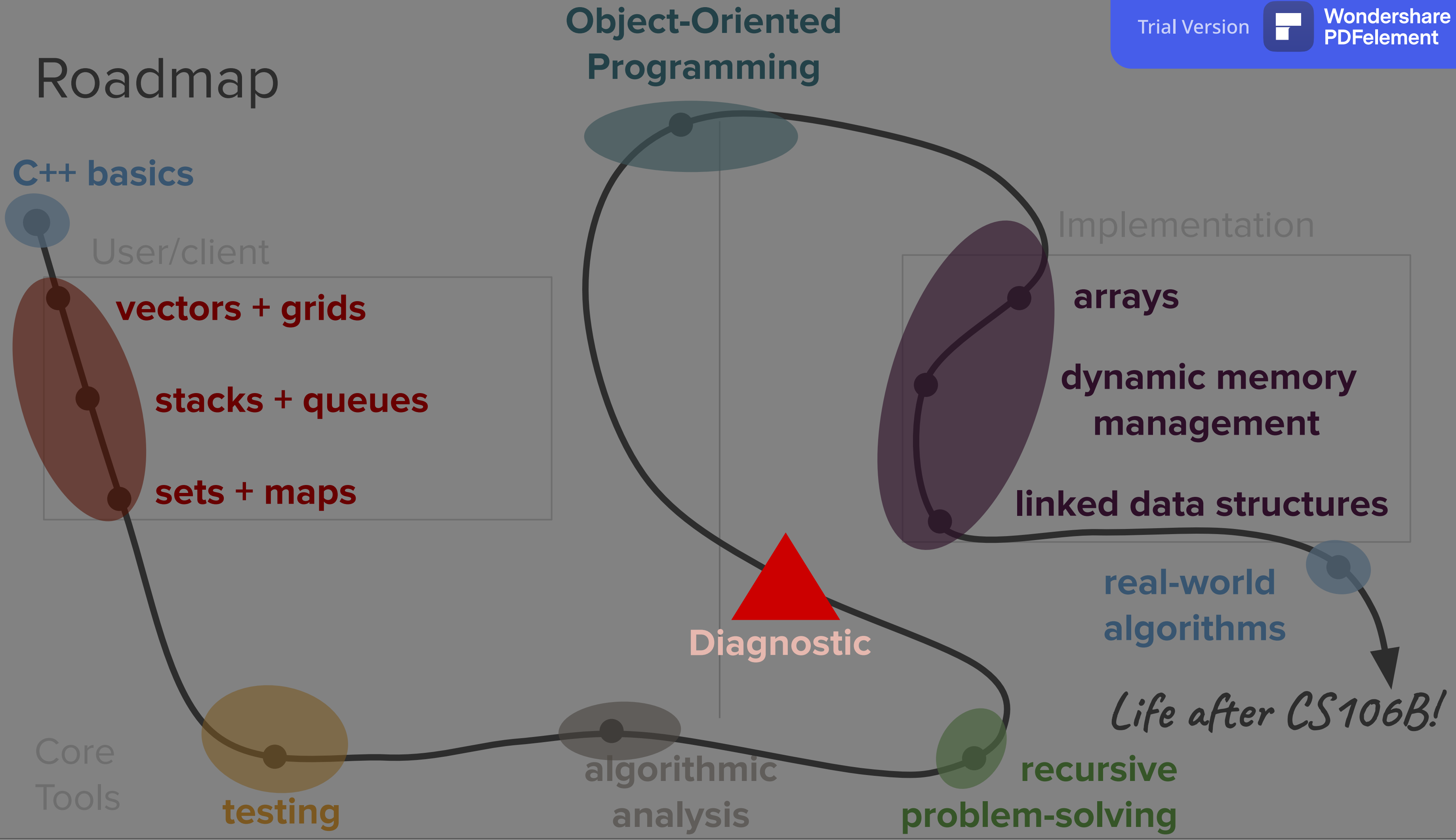
- Writing classes requires the creation of a header (`.h`) file for the interface and an implementation (`.cpp`) file.

# 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

# 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!
Diagnostic
Core Tools
testing
algorithmic analysis
recursive problem-solving

# Dynamic memory and arrays