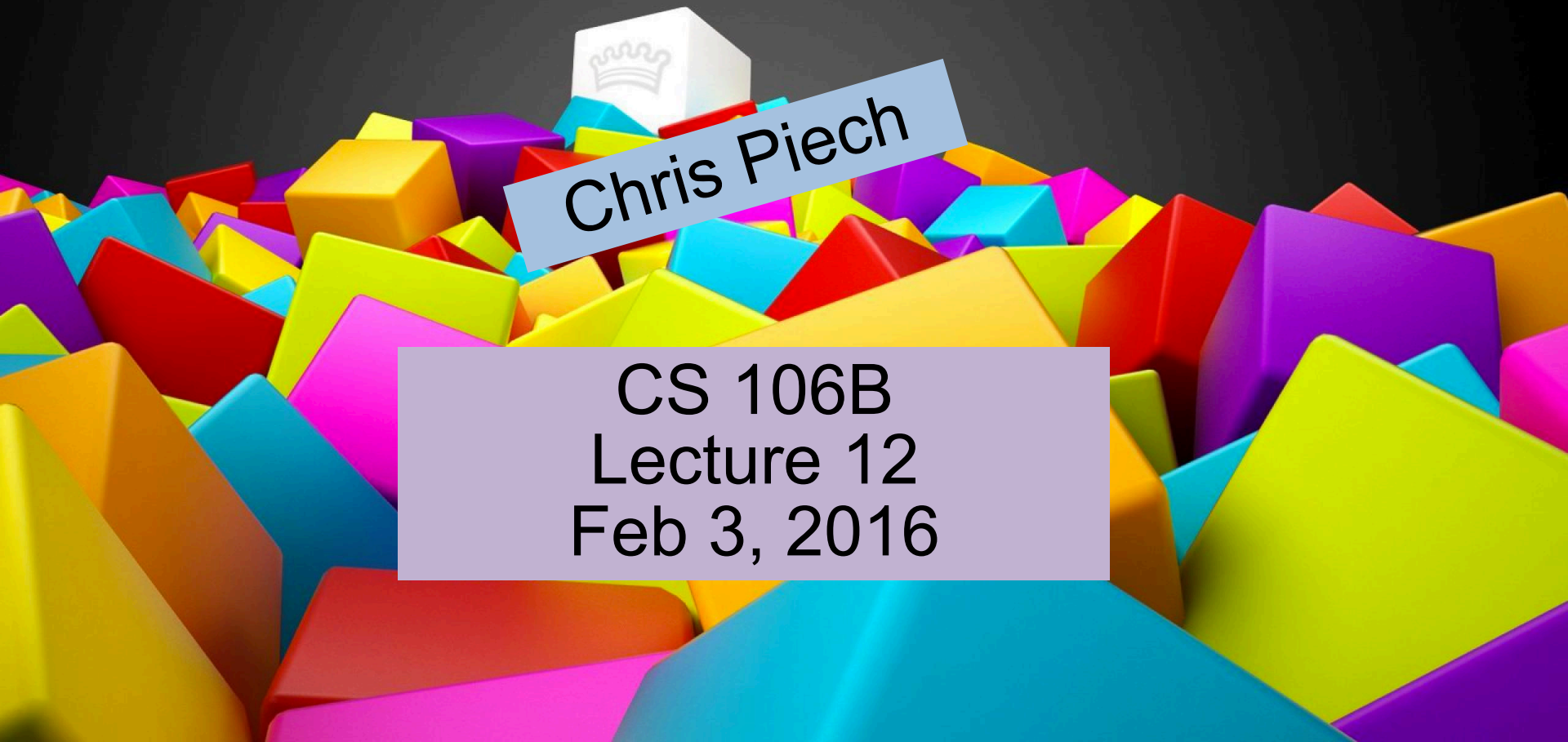
Classes



Chris Piech

CS 106B
Lecture 12
Feb 3, 2016

Socrative.com

Room: **106BWIN16**

STUDENT

Announcement: Midterm

Last name A-HAN: Hewlett 200
Last name HAP-MC: Hewlett 201
Last name ME-Z: Braun Auditorium

Concepts: Functions, Collections (Stacks, Queues, Vector, Grid, Map, Set), Recursion, Recursive Backtracking

Eg everything up to Monday and in the assignments you have done.

Midterm Review

A

Sunday morning review

B

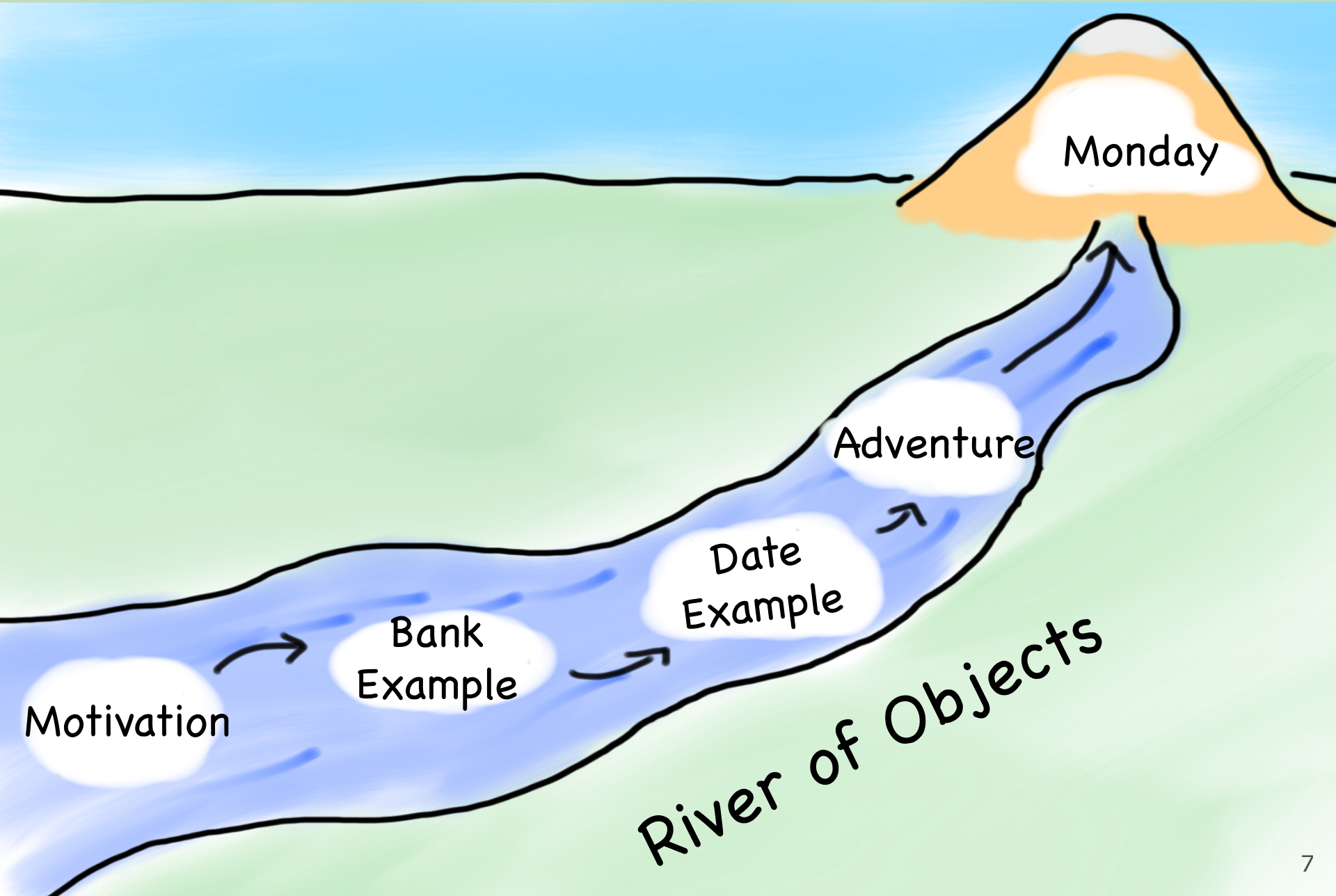
More weekend handouts describing what you should know + better practice exams.

Announcement: Boggle

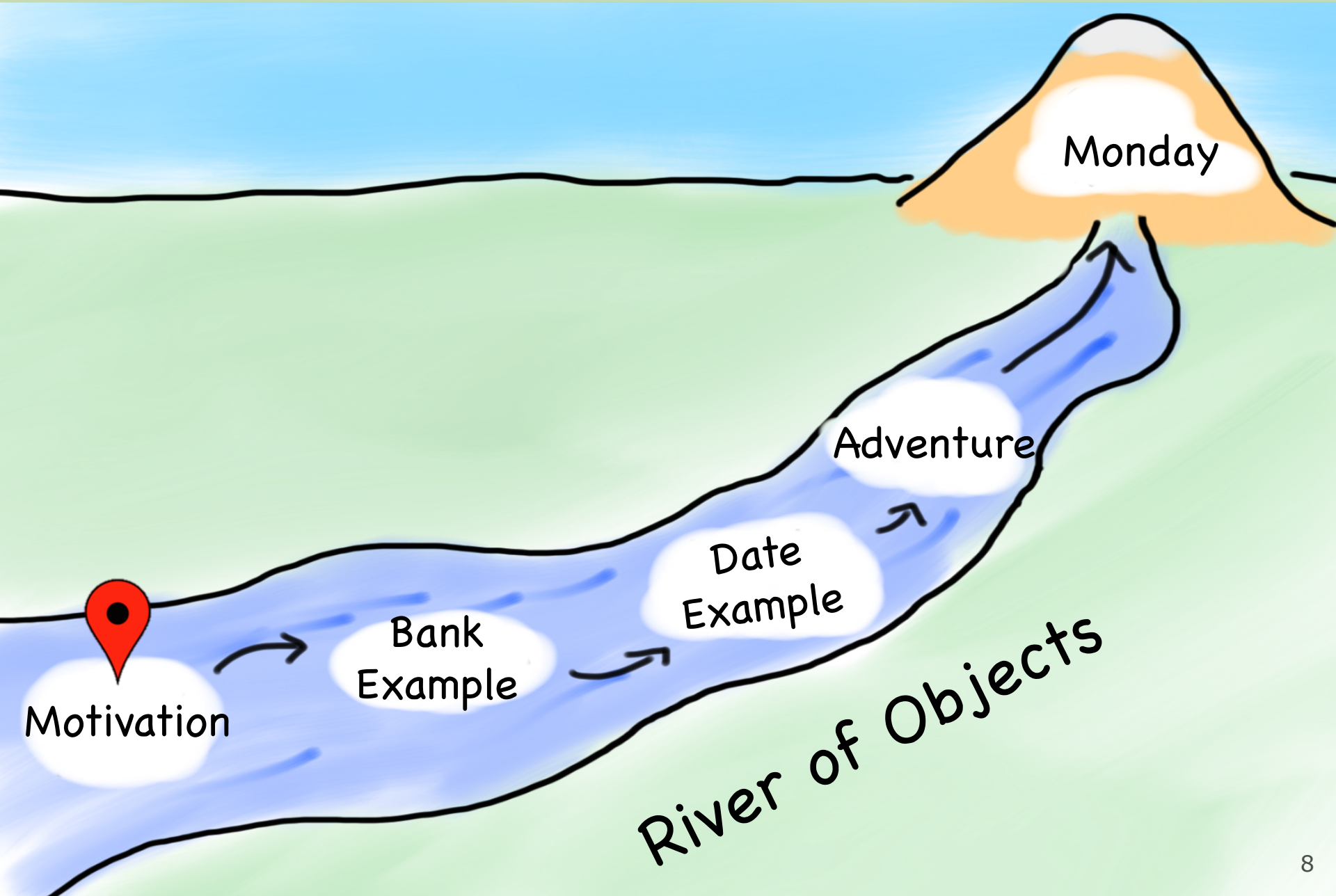
Today's Goals

1. Learn how to define a class in C++

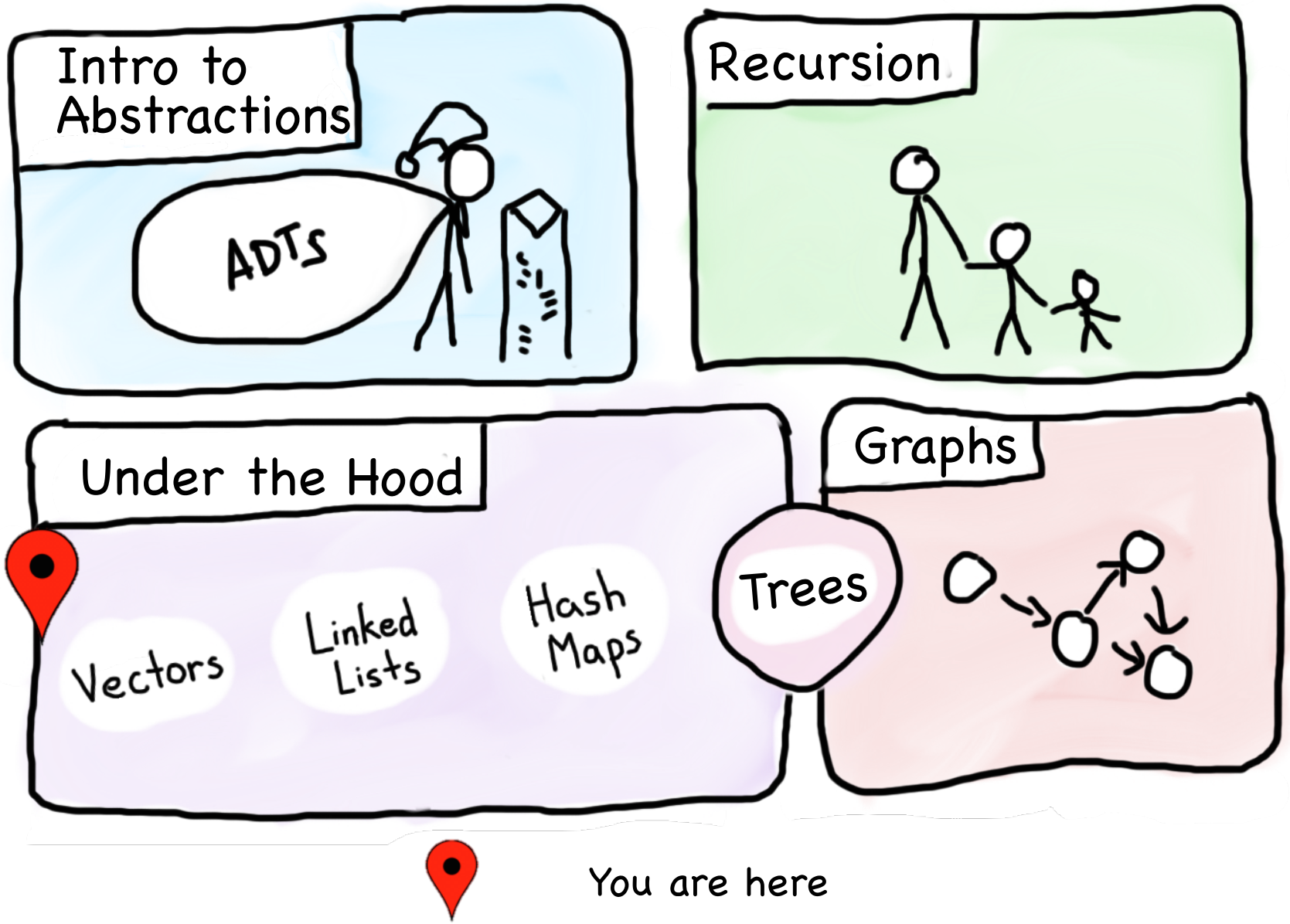
Today's Goals



Today's Goals



Course Syllabus



Some *large* programs are in C++


WORLD OF WARCRAFT

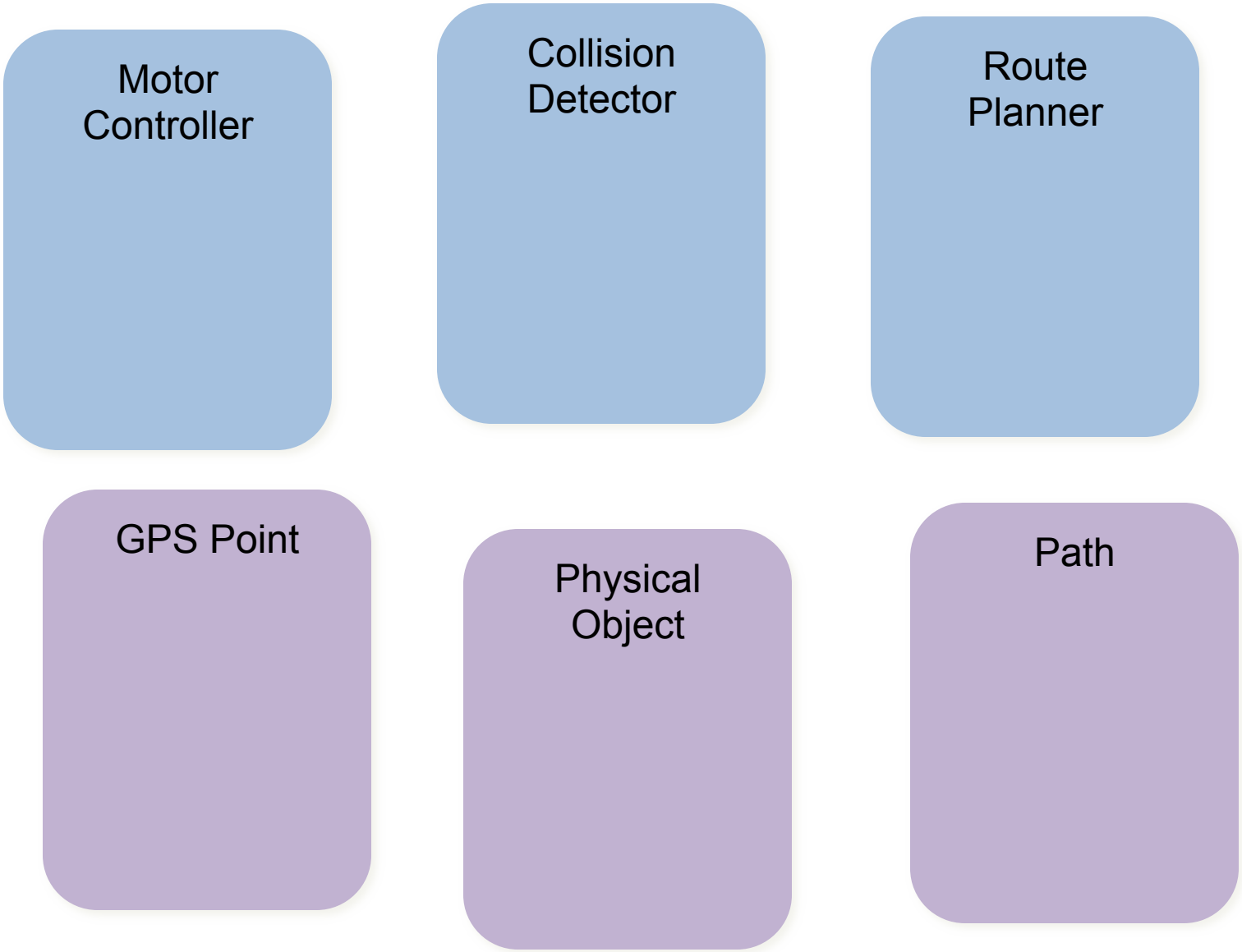

Google

Self Driving Car in C++

almost all the code is written in C++.
- Sebastian Thrun

How?

Decomposition Across Files



Class examples

- A calendar program might want to store information about dates, but C++ does not have a **`Date`** type.

- A student registration system needs to store info about students, but C++ has no **`Student`** type.

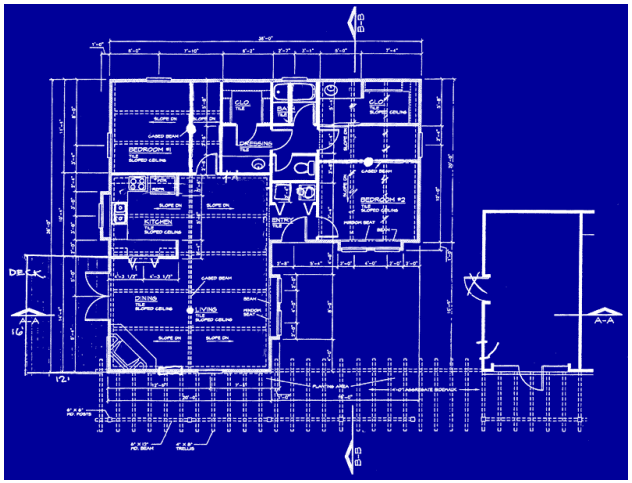
- A music synthesizer app might want to store information about users' accounts, but C++ has no **`Instrument`** type.

- However, C++ does provide a feature for us to add new data types to the language: **classes**.
  - Writing a class defines a new data type.

Classes

class: A template for a new type of variable.

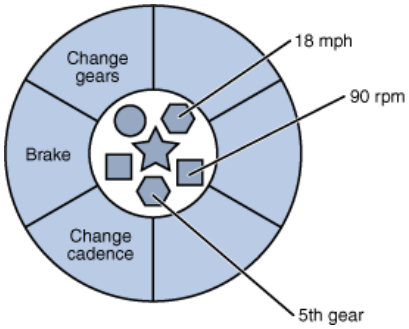
A blueprint is a helpful analogy



Elements of a class

member variables: State inside each object.

- Also called "instance variables" or "fields"
- Declared as `private`
- Each object created has a copy of each field.



member functions: Behavior that executes inside each object.

- Also called "methods"
- Each object created has a copy of each method.
- The method can interact with the data inside that object.

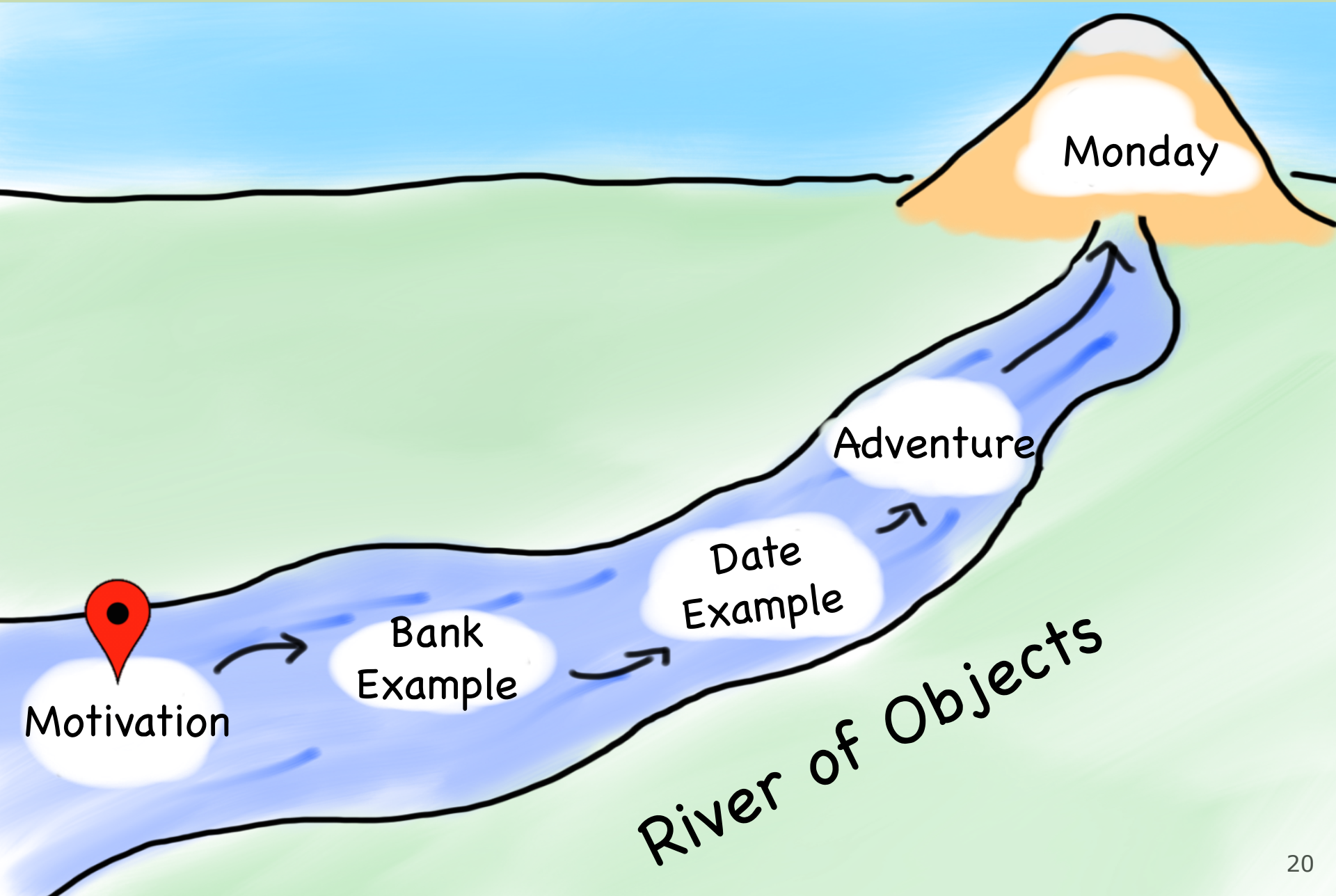
constructor: Initializes new objects as they are created.

- Sets the initial state of each `new` object.
- Often accepts parameters for the initial state of the fields.

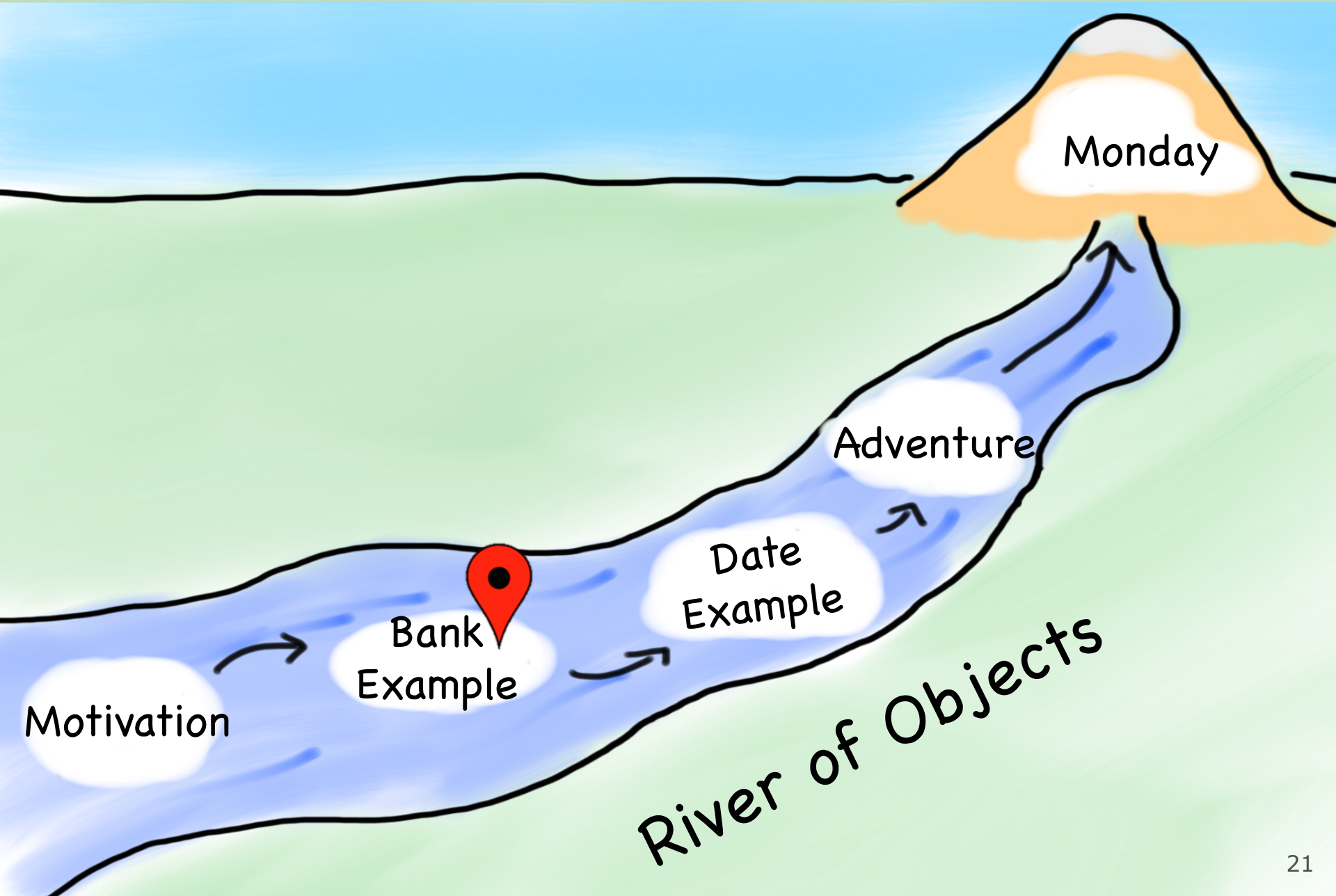
Source Interface Divide

Interface	Source
name.h	*name*.cpp
Client reads	Implementer writes
Shows methods and states instance variables	Implements methods

Today's Goals



Today's Goals



Structure of a .h file

```
// classname.h
#ifndef _classname_h
#define _classname_h

class declaration;

#endif
```

This is protection in case multiple .cpp files include this .h, so that its contents won't get declared twice

A class declaration

```
class ClassName {                          // in ClassName.h
public:
    ClassName(parameters);                 // constructor

    returnType name(parameters); // member functions
    returnType name(parameters); // (behavior inside
    returnType name(parameters); //  each object)

private:
    type name;      // member variables
    type name;      // (data inside each object)
};
```

IMPORTANT: *must* put a semicolon at end of class declaration (argh)

Class example (v1)

```
// Initial version of BankAccount.h.
// Uses public member variables and no functions.
// Not good style, but we will improve it.

#ifndef _bankaccount_h
#define _bankaccount_h

class BankAccount {
public:
    string name;       // each BankAccount object
    double balance;    // has a name and balance
};

#endif
```

Using objects

```
// v1 with public fields (bad)
BankAccount ba1;
ba1.name = ”Chris";
ba1.balance = 1.25;

BankAccount ba2;
ba2.name = "Mehran";
ba2.balance = 9999.00;
```

ba1

```
name    = ”Chris"
balance = 1.25
```

ba2

```
name    = "Mehran"
balance = 9999.00
```

- Think of an object as a way of grouping multiple variables.
  - Each object contains a name and balance field inside it.
  - We can get/set them individually.
  - Code that uses your objects is called *client* code.

What does that look like?

Member func. bodies

- In ***ClassName*.**cpp, we write bodies (definitions) for the member functions that were declared in the ***.h*** file:

```
// ClassName.cpp
#include "ClassName.h"

// member function
returnType ClassName::methodName(parameters) {
    statements;
}
```

  – Member functions/constructors can refer to the object's fields.

- *Exercise:* Write a `withdraw` member function to deduct money from a bank account's balance.

The implicit parameter

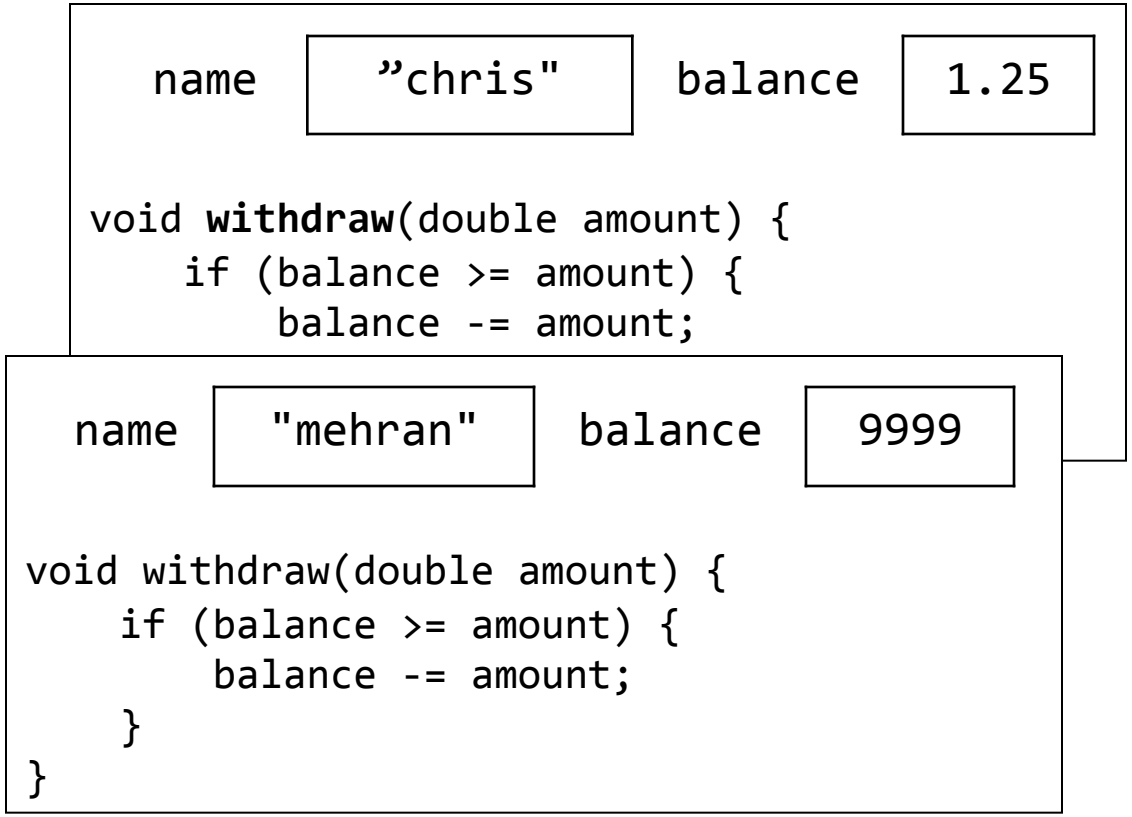
- **implicit parameter**:
  The object on which a member function is called.
  - During the call `chris.withdraw(...)`,
    the object named `chris` is the implicit parameter.
  - During the call `mehran.withdraw(...)`,
    the object named `mehran` is the implicit parameter.
  - The member function can refer to <u>that object's</u> member variables.
    - We say that it executes in the *context* of a particular object.
    - The function can refer to the data of the object it was called on.
    - It behaves as if each object has its own *copy* of the member functions.

Member func diagram

```
// BankAccount.cpp
void BankAccount::withdraw(double amount) {
    if (balance >= amount) {
        balance -= amount;
    }
}


// client program
BankAccount chris;
BankAccount mehran;
...
chris.withdraw(5.00);

mehran.withdraw(99.00);
```



Initializing objects

- It's bad to take 3 lines to create a `BankAccount` and initialize it:

```
BankAccount ba;
ba.name = ”Chris";
ba.balance = 1.25;                    // tedious
```

- We'd rather specify the fields' initial values at the start:

```
BankAccount ba(”Chris", 1.25);    // better
```

  – We are able to this with most types of objects in C++ and Java.
  – You can achieve this functionality using a **constructor**.

Constructors

```
ClassName::ClassName(parameters) {
    statements to initialize the object;
}
```

- **constructor**: Initializes state of new objects as they are created.
  - runs when the client declares a new object
  - no return type is specified;
    it implicitly "returns" the new object being created

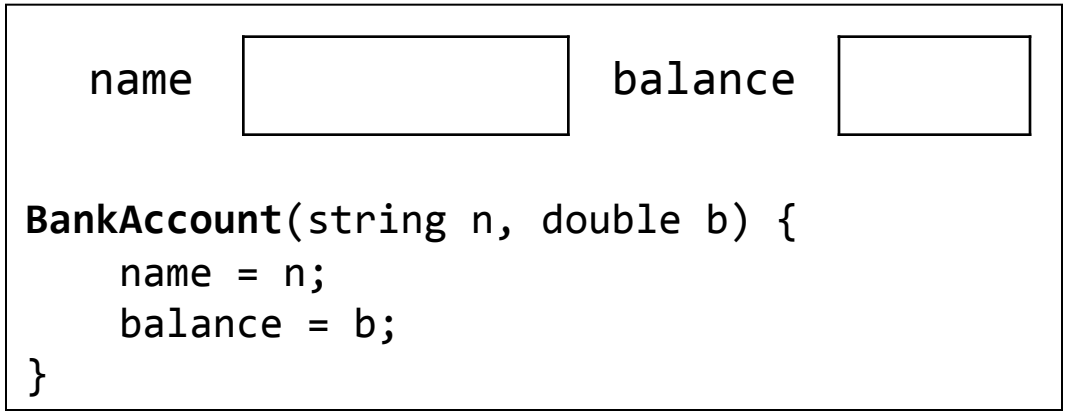
  - If a class has no constructor, C++ gives it a *default constructor* with no parameters that does nothing.

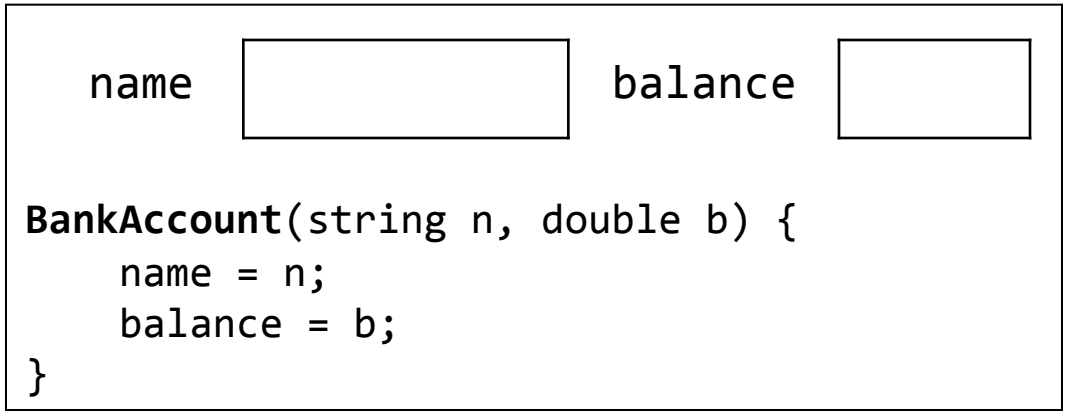
Constructor diagram

```
// BankAccount.cpp
BankAccount::BankAccount(string n, double b) {
    name = n;
    balance = b;
}
```

```
// client program
BankAccount b1(
    ”Chris", 1.25);

BankAccount b2(
    "Mehran", 9999);
```





The keyword this

- As in Java, C++ has a `this` keyword to refer to the current object.
  - Syntax: `this->`***`member`***
  - *Common usage:* In constructor, so parameter names can match the names of the object's member variables:

```
BankAccount::BankAccount(string name,
                         double balance) {
    this->name = name;
    this->balance = balance;
}
```

`this` uses `->` not `.` because it is a "pointer"; we'll discuss that later

Preconditions

- **precondition**: Something your code *assumes is true* at the start of its execution.
  - Often documented as a comment on the function's header.
  - If violated, the class often **throws an exception**.

```
// Initializes a BankAccount with the given state.
// Precondition: balance is non-negative
BankAccount::BankAccount(string name, double balance) {
    if (balance < 0) {
        throw balance;
    }
    this->name = name;
    this->balance = balance;
}
```

Private data

```
private:
    type name;
```

- **encapsulation**: Hiding implementation details of an object from its clients.
  - Encapsulation provides *abstraction*.
    - separates external view (behavior) from internal view (state)
  - Encapsulation protects the integrity of an object's data.

- A class's data members should be declared *private*.
  - No code outside the class can access or change it.

Accessor functions

- We can provide methods to get and/or set a data field's value:

```
// "read-only" access to the balance ("accessor")
double BankAccount::getBalance() {
    return balance;
}

// Allows clients to change the field ("mutator")
void BankAccount::setName(string newName) {
    name = newName;
}
```

  – Client code will look like this:

```
cout << ba.getName() << ":$" << ba.getBalance() << endl;
ba.setName("Cynthia");
```

Operator overloading (6.2)

- C++ allows you to *overload*, or redefine, the behavior of many common operators in the language:
  - unary: `+ - ++ -- * & ! ~ new delete`
  - binary: `+ - * / % += -= *= /= %= & | && || ^`
    `== != < > <= >= = [] -> () ,`

- Overuse of operator overloading can lead to confusing code.
  - *Rule of Thumb:* Don't abuse this feature. Don't define an overloaded operator unless its meaning and behavior are completely obvious.

Hey future Chris.

This is past Chris.
Tell them about Date!

Date Class

I am always calculating the number of days until a particular date….

Date Class

```
int main() {
    Date today(3,2,2016);
    Date springBreak(19,3,2016);

    cout << "spring break: " << springBreak << endl;

    cout << "days until spring break: ";
    cout << today.daysUntil(springBreak) << endl;

    today.incrementDay();

    cout << "days until spring break: ";
    cout << today.daysUntil(springBreak) << endl;

    return 0;
}
```

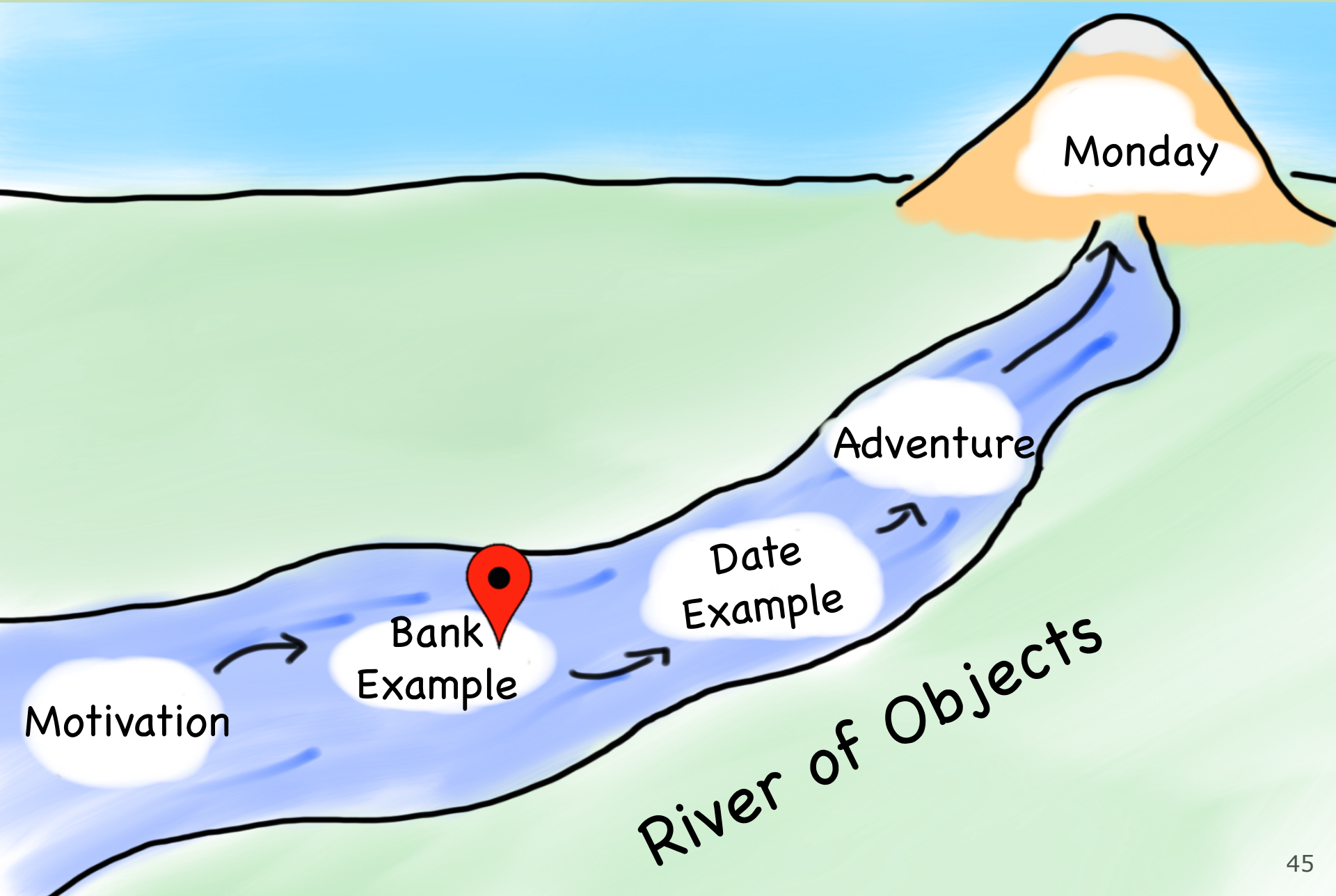
“summer's date hath all too short a lease”

-Bill Shakespeare, Sonnet 18

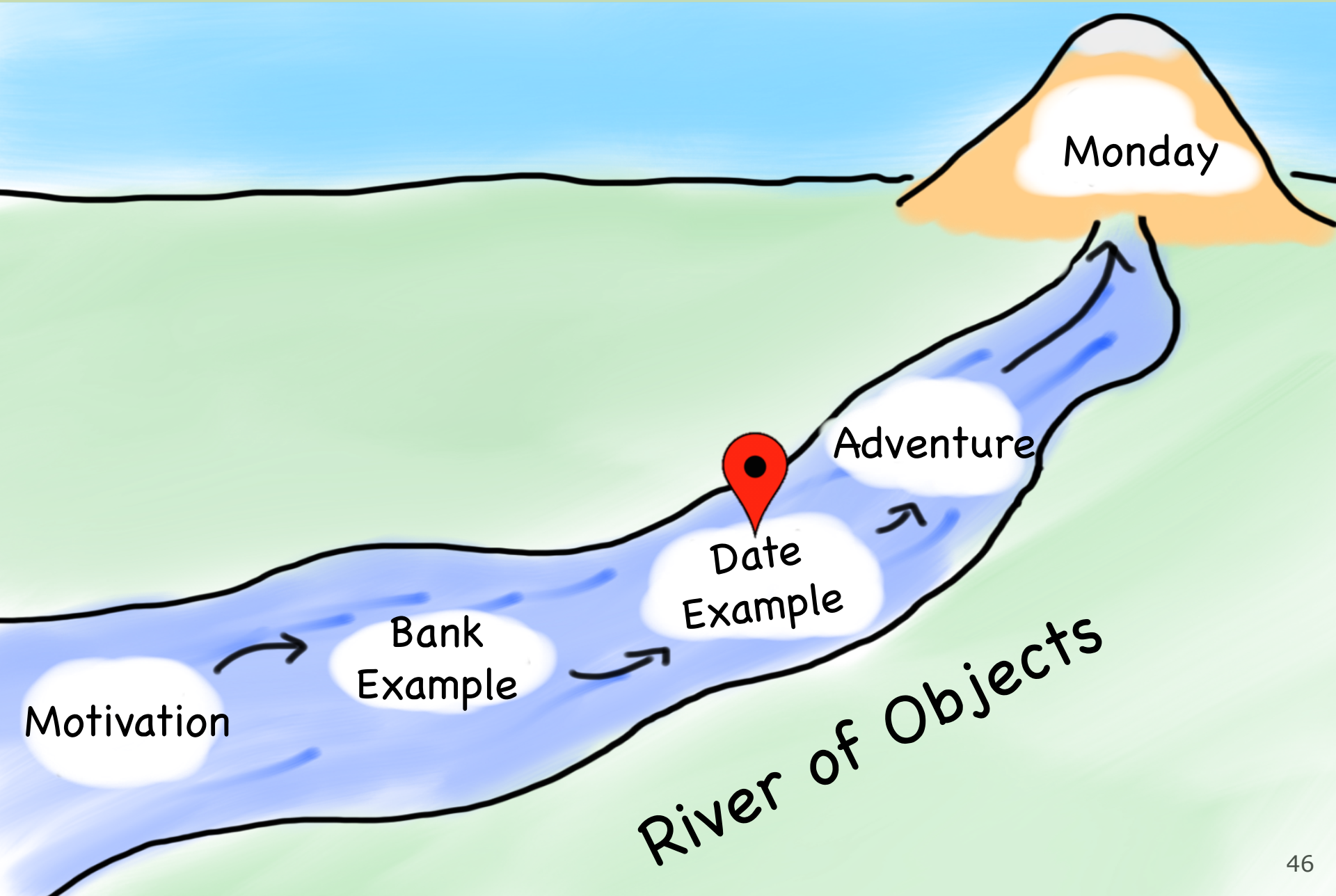
But…

C++ has no Dates ☹

Today's Goals



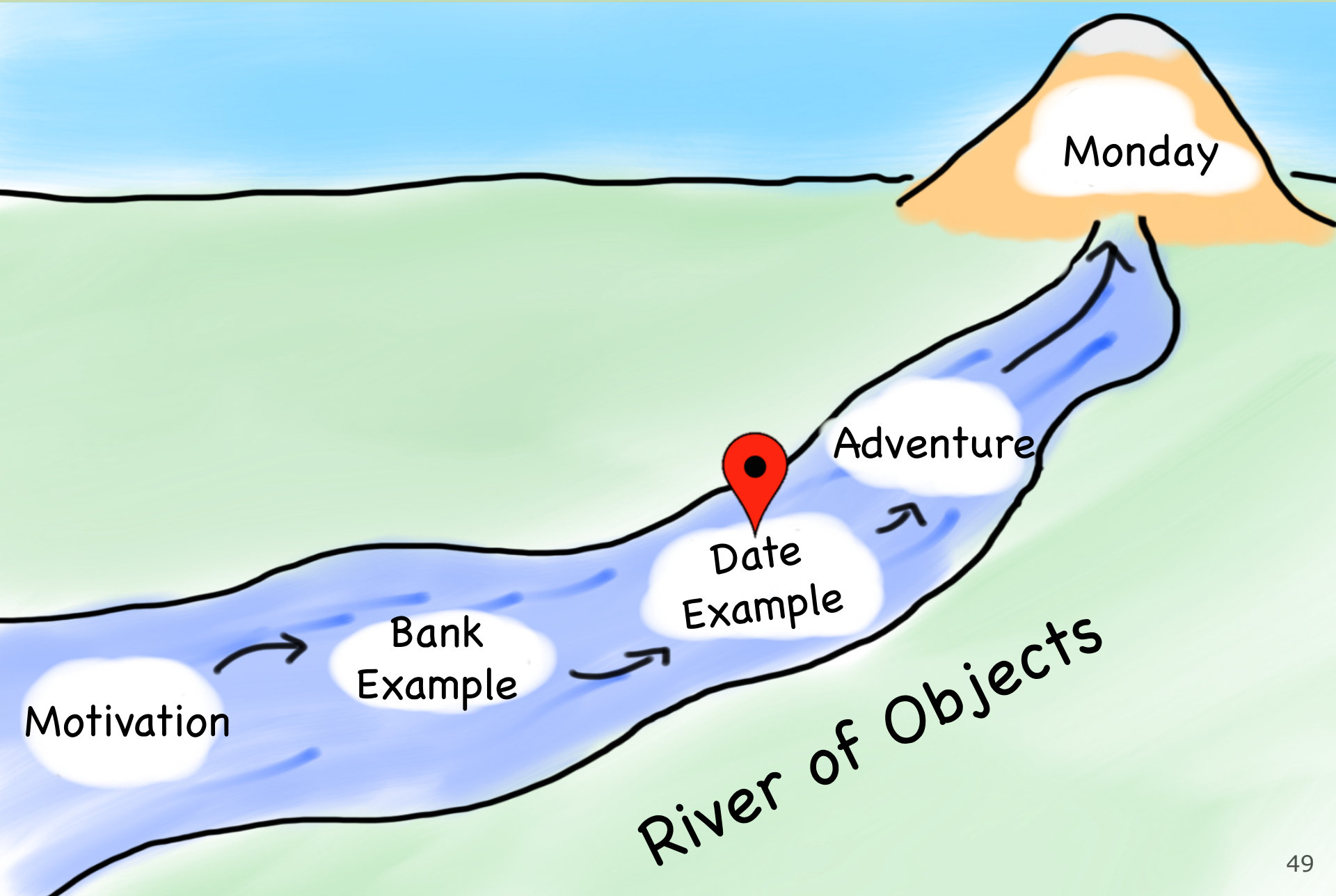
Today's Goals



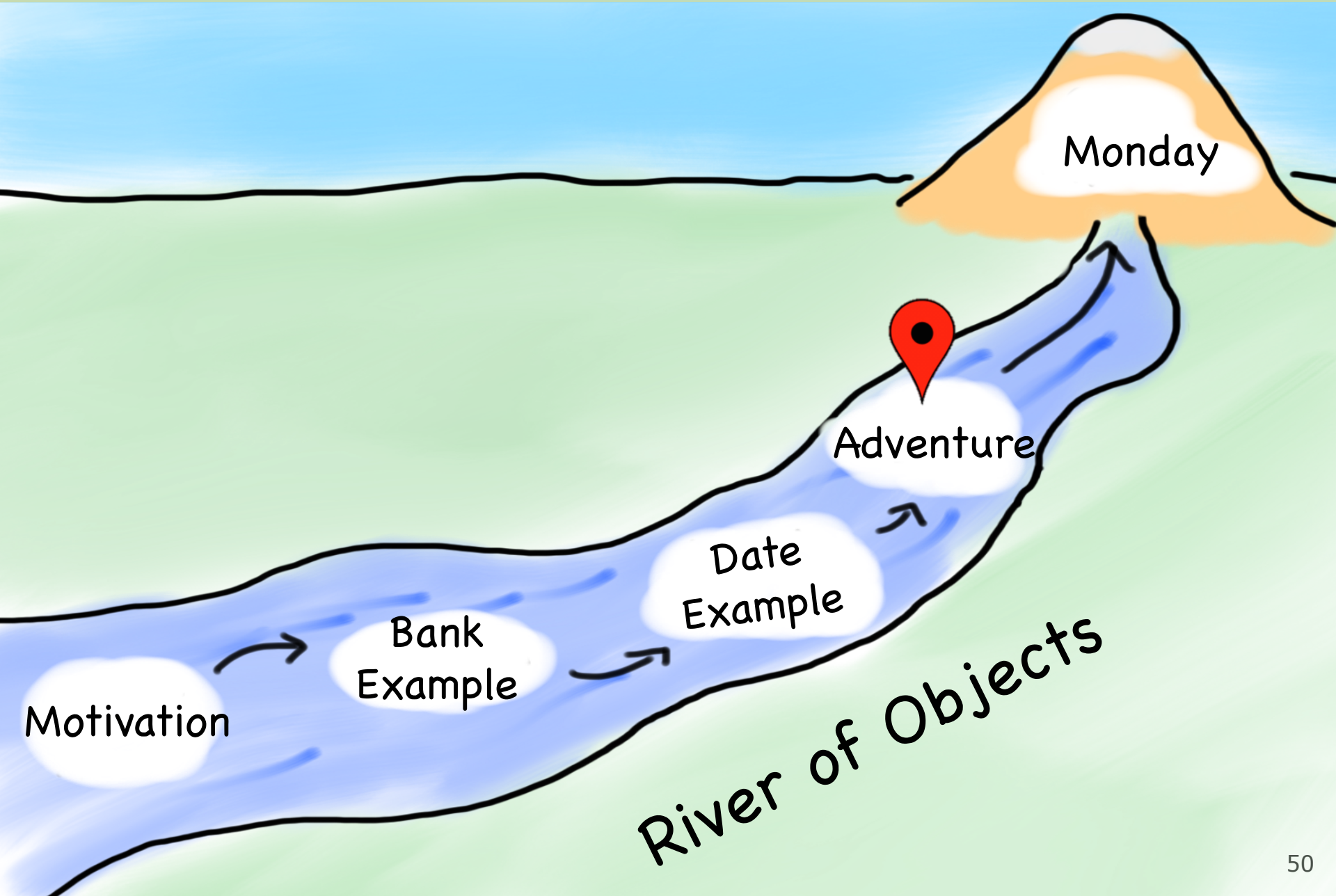
You know what to do


Calendar
SUNDAY MONDAY TUESDAY WEDNESDAY THURSDAY
4 5 6 7 8 9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31

Today's Goals

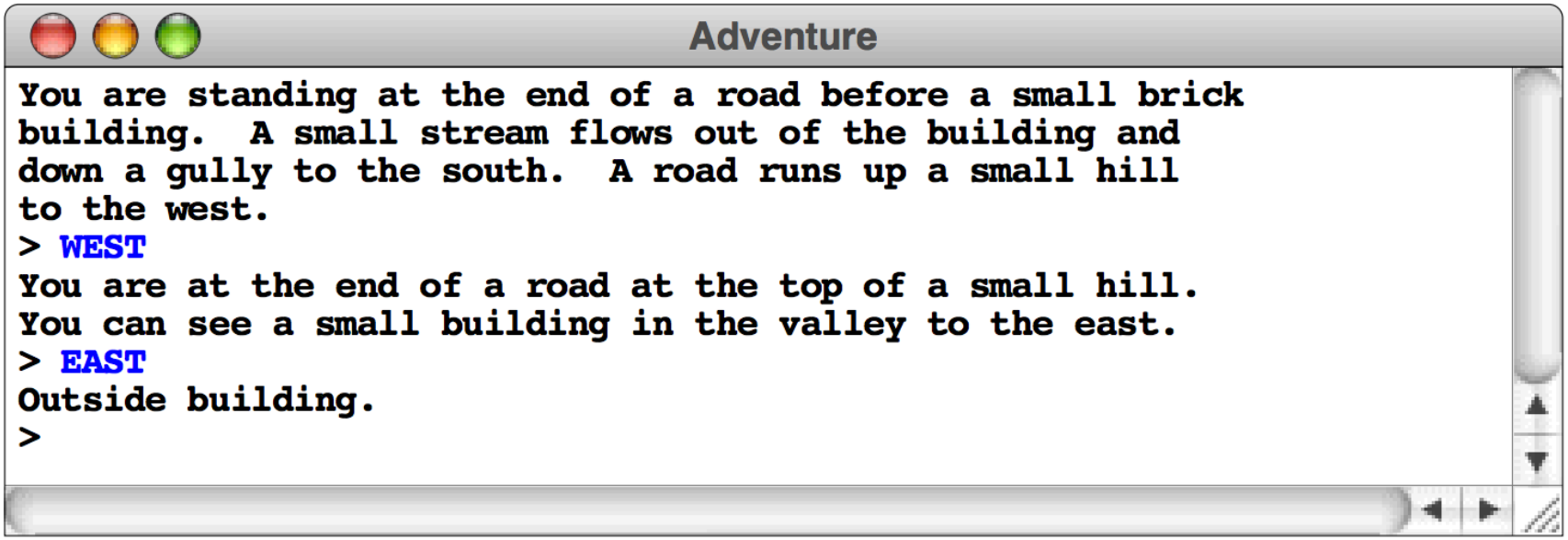


Today's Goals

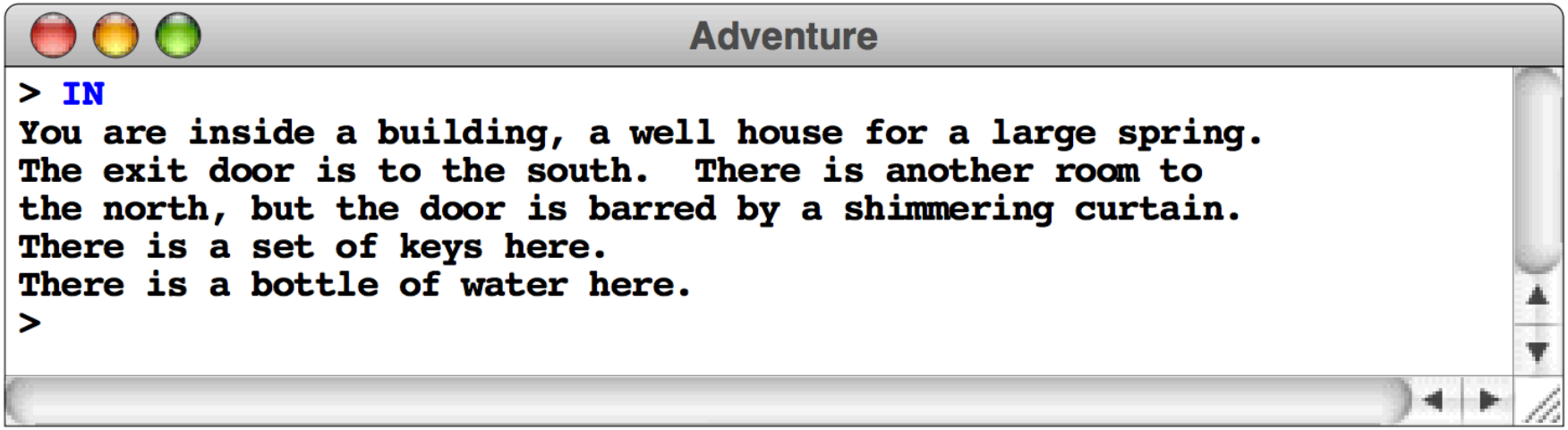


Challenge

Adventure Game



Adventure Game



Adventure Game

Before you go
send one class name to socrative

STUDENT

Email big ideas to
piech@cs.stanford.edu

Today's Goals

1. Learn how to define a class in C++