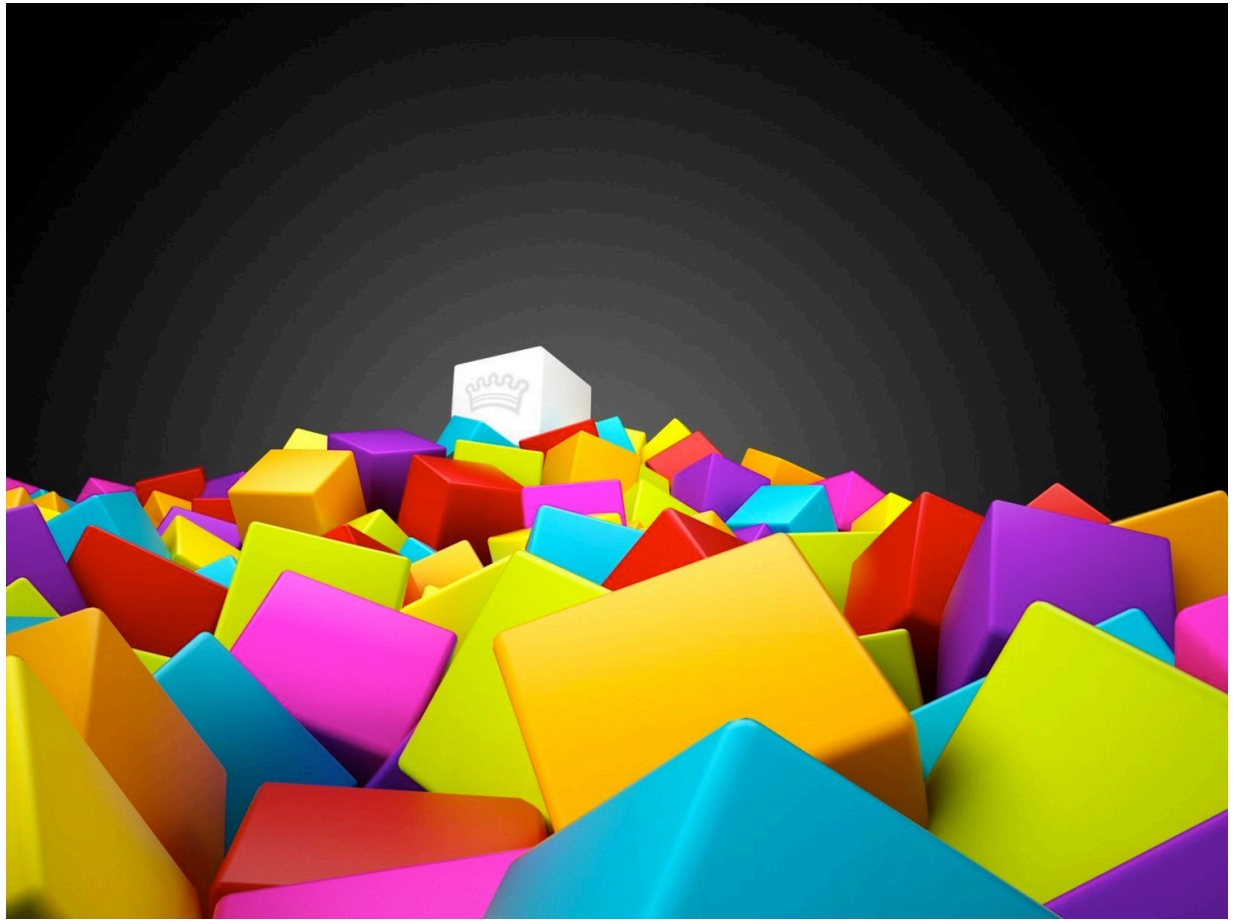
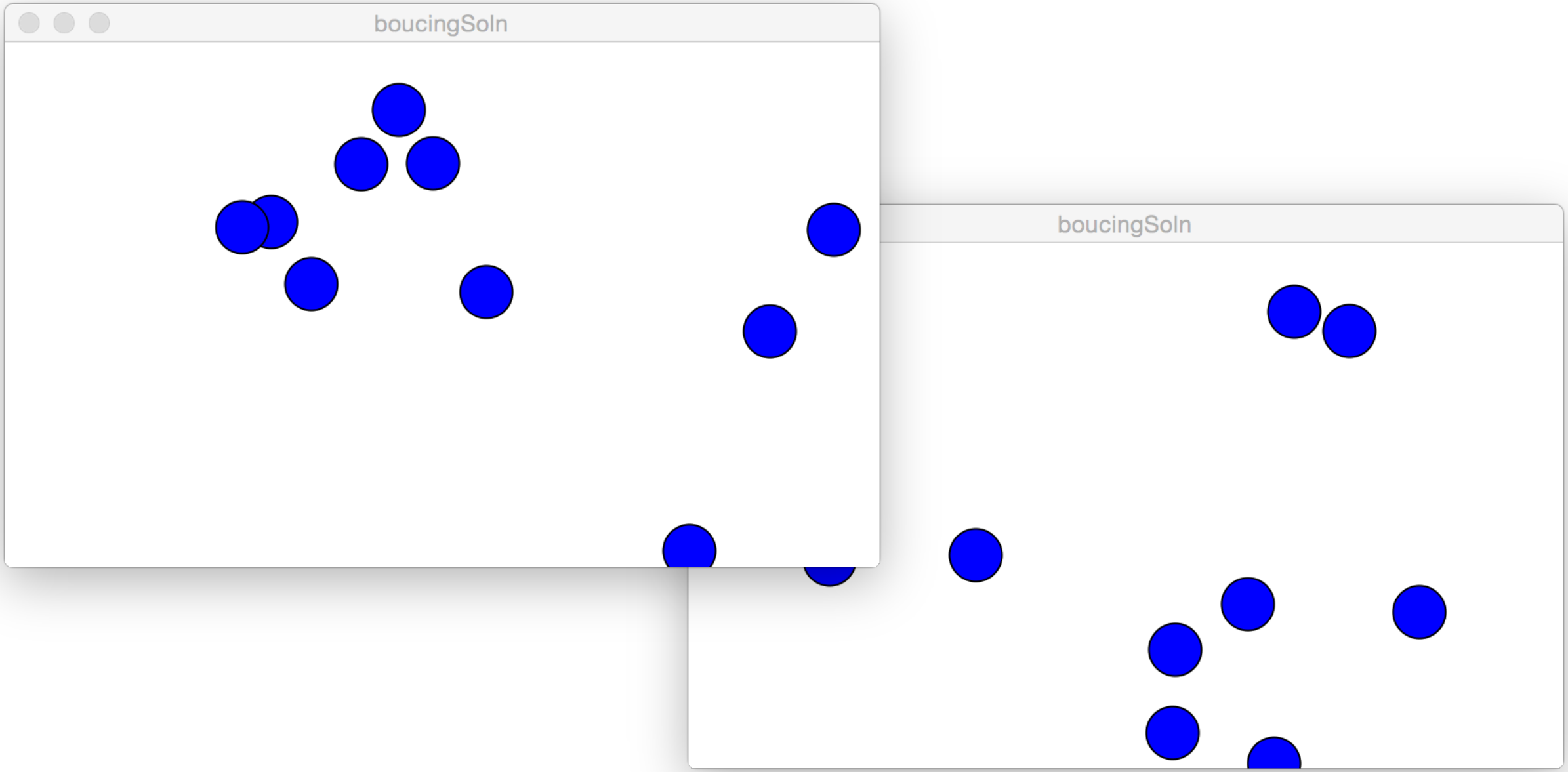
Classes

CS 106B

Programming Abstractions
Fall 2016
Stanford University
Computer Science Department



Bouncing Ball



Announcement

- Chris Piech office hours moved to tomorrow.
- Midterm materials online this afternoon.

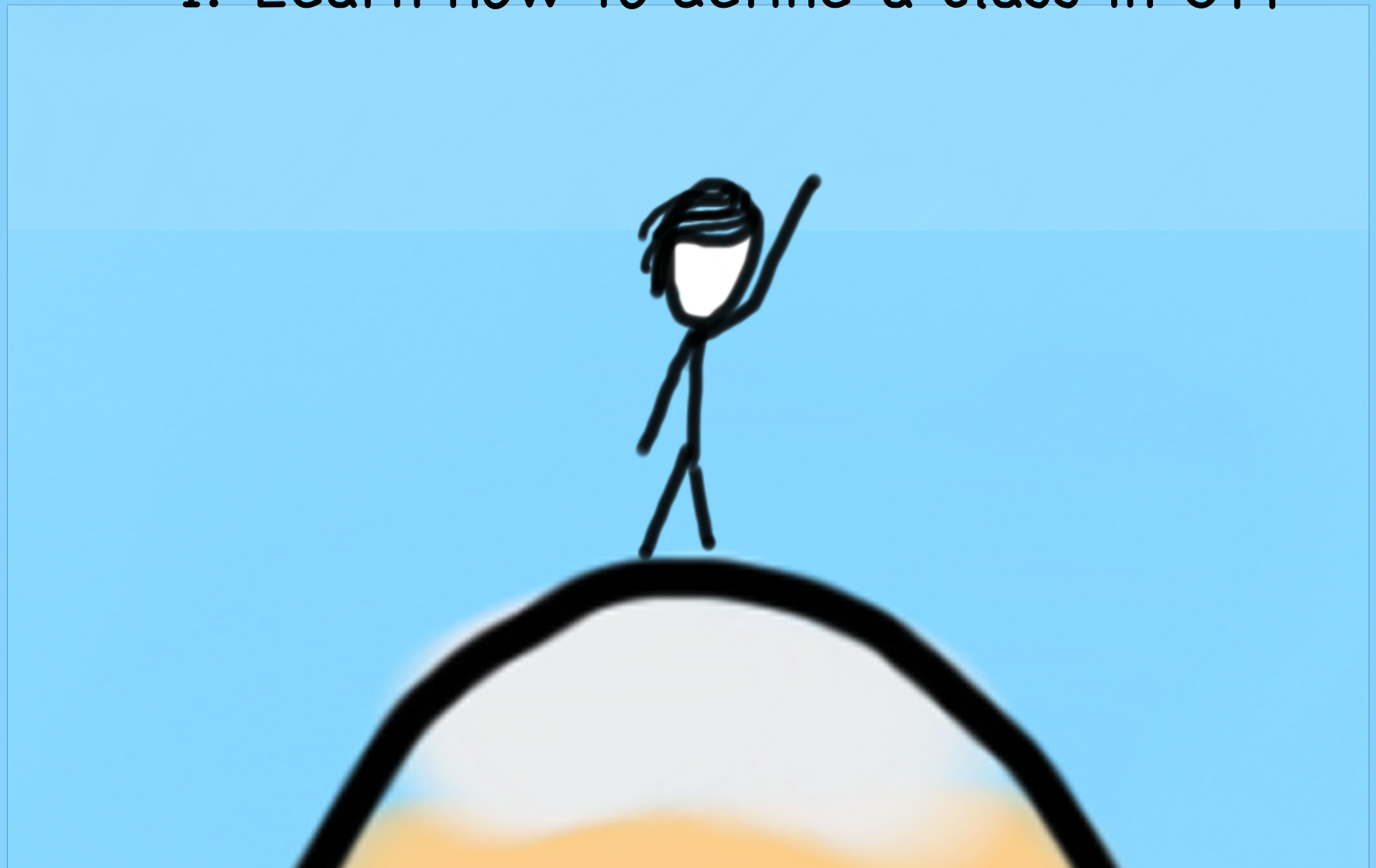
Midterm Review

Sunday morning review

Handouts today on the website

Today's Goal

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



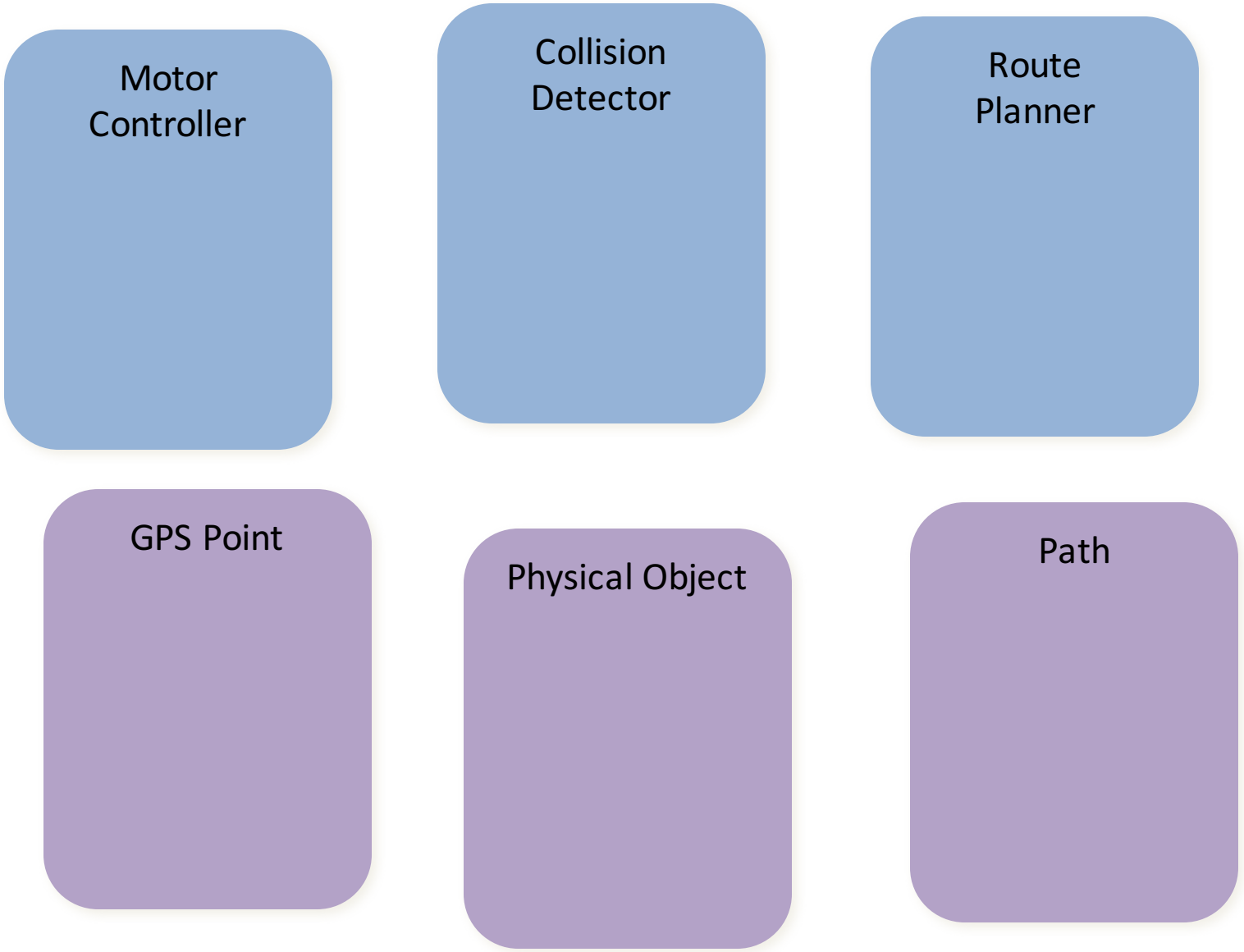
Some *large* programs are in C++


WORLD OF WARCRAFT
Google


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

How?

Decomposition Across Files



The Need For New Variable Types

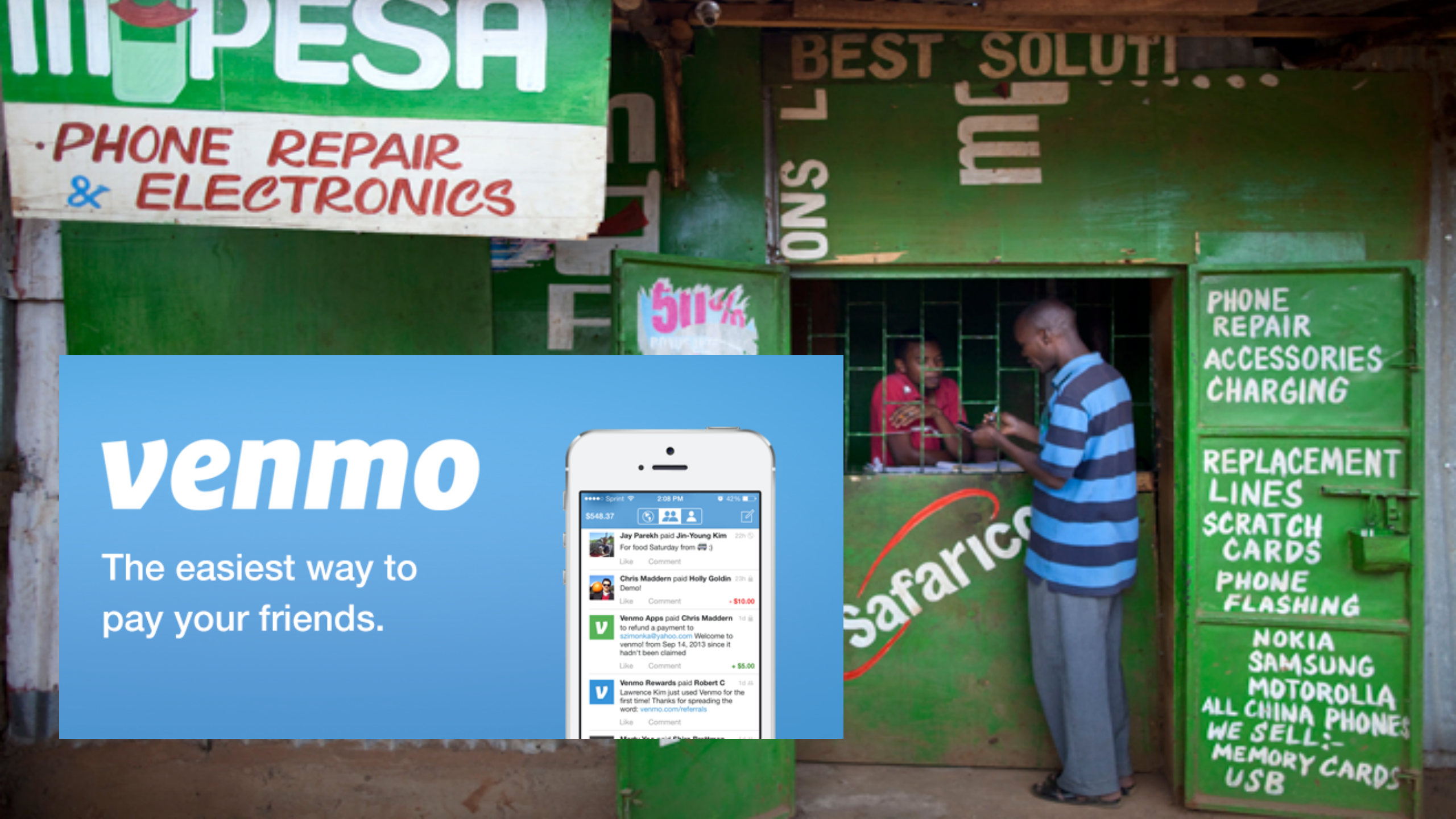
- A calendar program might want to store information about dates, but C++ does not have a **`Date`** type.

- 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.

mPesa


M-PESA
PHONE REPAIR
& ELECTRONICS
BEST SOLUT!
PHONE
REPAIR
ACCESSORIES
CHARGING
REPLACEMENT
LINES
SCRATCH
CARDS
PHONE
FLASHING
NOKIA
SAMSUNG
MOTOROLLA
ALL CHINA PHONES
WE SELL:-
MEMORY CARDS
USB
venmo
The easiest way to
pay your friends.
$548.37
Jay Parekh paid Jin-Young Kim
For food Saturday from :)
Like Comment
Chris Maddern paid Holly Goldin
Demo!
Like Comment - $10.00
Venmo Apps paid Chris Maddern
to refund a payment to
szimonka@yahoo.com Welcome to
venmo! from Sep 14, 2013 since it
hadn't been claimed
Like Comment + $5.00
Venmo Rewards paid Robert C
Lawrence Kim just used Venmo for the
first time! Thanks for spreading the
word: venmo.com/referrals
Like Comment

Structs, The Original G

- Venmo needs to store info about peoples bank accounts, but C++ has no **`BankAccount`** type.

Structs, The Original G

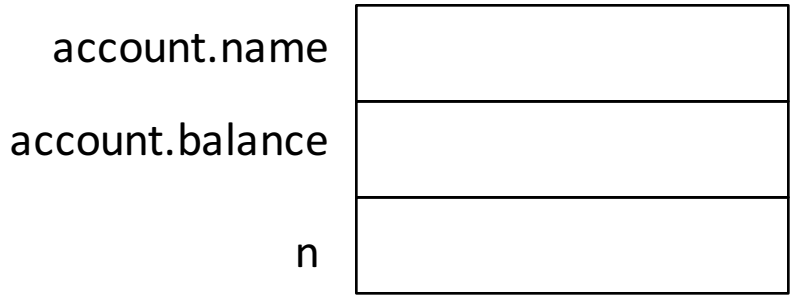
- Venmo needs to store info about peoples bank accounts, but C++ has no **`BankAccount`** type.

```
struct BankAccount {
    string name;
    double balance;
};
```

Structs, The Original G

- Venmo needs to store info about peoples bank accounts, but C++ has no **`BankAccount`** type.

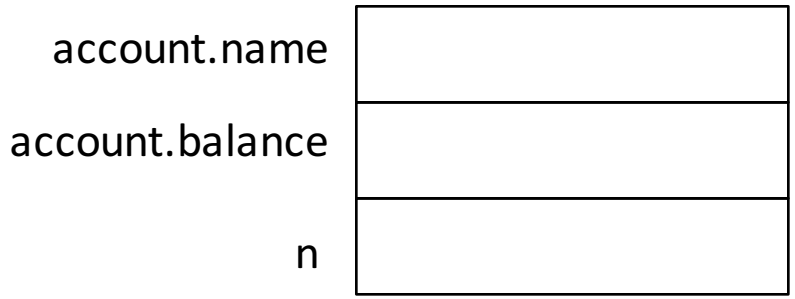
```
struct BankAccount {
    string name;
    double balance;
};


int main() {
    int n = 3;
    BankAccount account;
}
```



Structs, The Original G

- Venmo needs to store info about peoples bank accounts, but C++ has no **`BankAccount`** type.

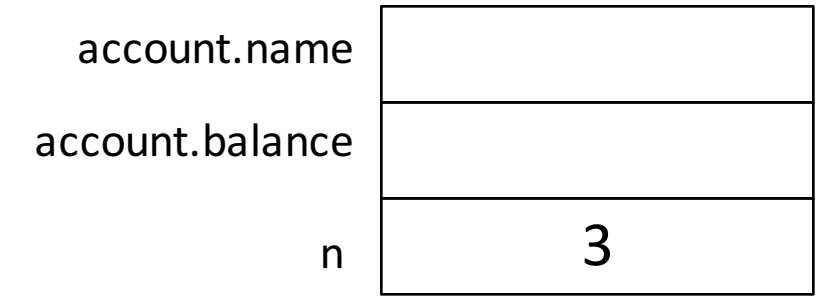
```
struct BankAccount {
    string name;
    double balance;
};


int main() {
    int n = 3;
    BankAccount account;
    account.name = “Alyssa”;
    account.balance = 25;
    cout << account.balance << endl;
}
```



Structs, The Original G

- Venmo needs to store info about peoples bank accounts, but C++ has no **`BankAccount`** type.

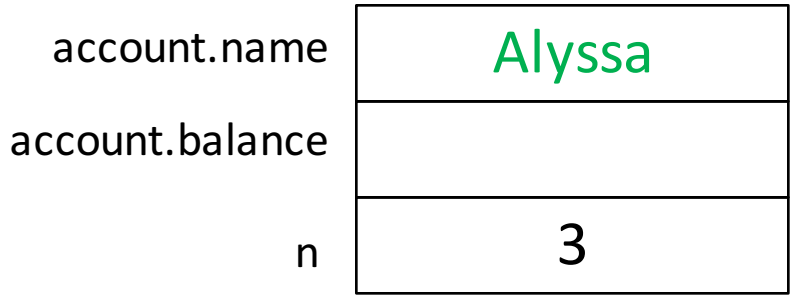
```
struct BankAccount {
    string name;
    double balance;
};


int main() {
    int n = 3;
    BankAccount account;
    account.name = “Alyssa”;
    account.balance = 25;
    cout << account.balance << endl;
}
```



Structs, The Original G

- Venmo needs to store info about peoples bank accounts, but C++ has no **`BankAccount`** type

```
struct BankAccount {
    string name;
    double balance;
};


int main() {
    int n = 3;
    BankAccount account;
    account.name = "Alyssa";
    account.balance = 25;
    cout << account.balance << endl;
}
```



Structs, The Original G

- Venmo needs to store info about peoples bank accounts, but C++ has no **`BankAccount`** type.

```
struct BankAccount {
    string name;
    double balance;
};


int main() {
    int n = 3;
    BankAccount account;
    account.name = "Alyssa";
    account.balance = 25;
    cout << account.balance << endl;
}
```

account.name	Alyssa
account.balance	25
n	3

Structs, The Original G

- Venmo needs to store info about peoples bank accounts, but C++ has no **`BankAccount`** type.

```
struct BankAccount {
    string name;
    double balance;
};


int main() {
    int n = 3;
    BankAccount account;
    account.name = “Alyssa”;
    account.balance = 25;
    cout << account.balance << endl;
}
```

account.name	Alyssa
account.balance	25
n	3

Structs, The Original G

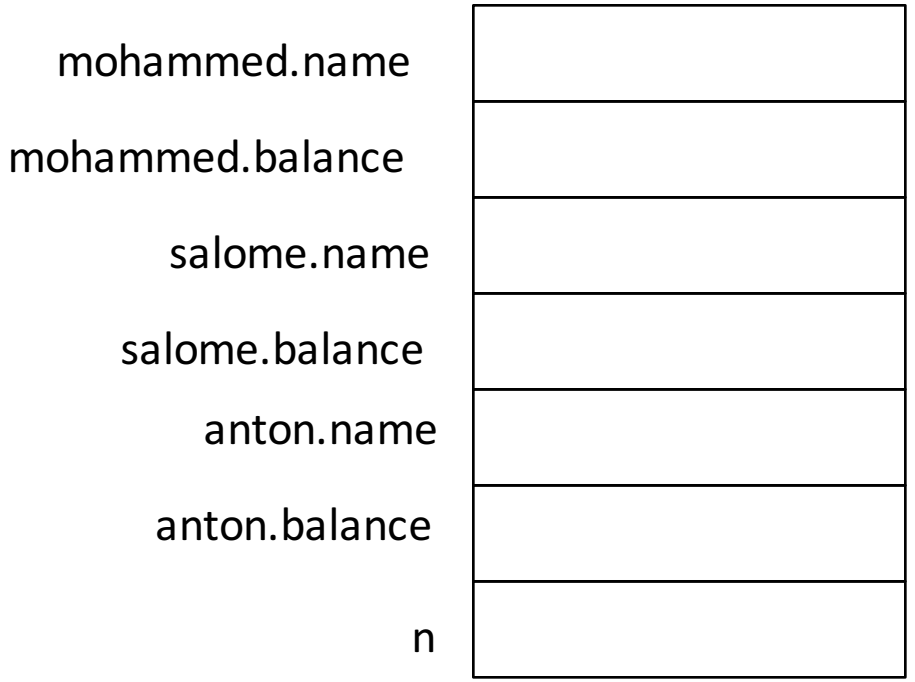
- Venmo needs to store info about peoples bank accounts, but C++ has no **`BankAccount`** type.

```
struct BankAccount {
    string name;
    double balance;
};


int main() {
}
```

Structs, The Original G

- Venmo needs to store info about peoples bank accounts, but C++ has no **`BankAccount`** type.

```
struct BankAccount {
    string name;
    double balance;
};


int main() {
    int n = 3;
    BankAccount anton;
    BankAccount salome;
    BankAccount mohammed;
}
```

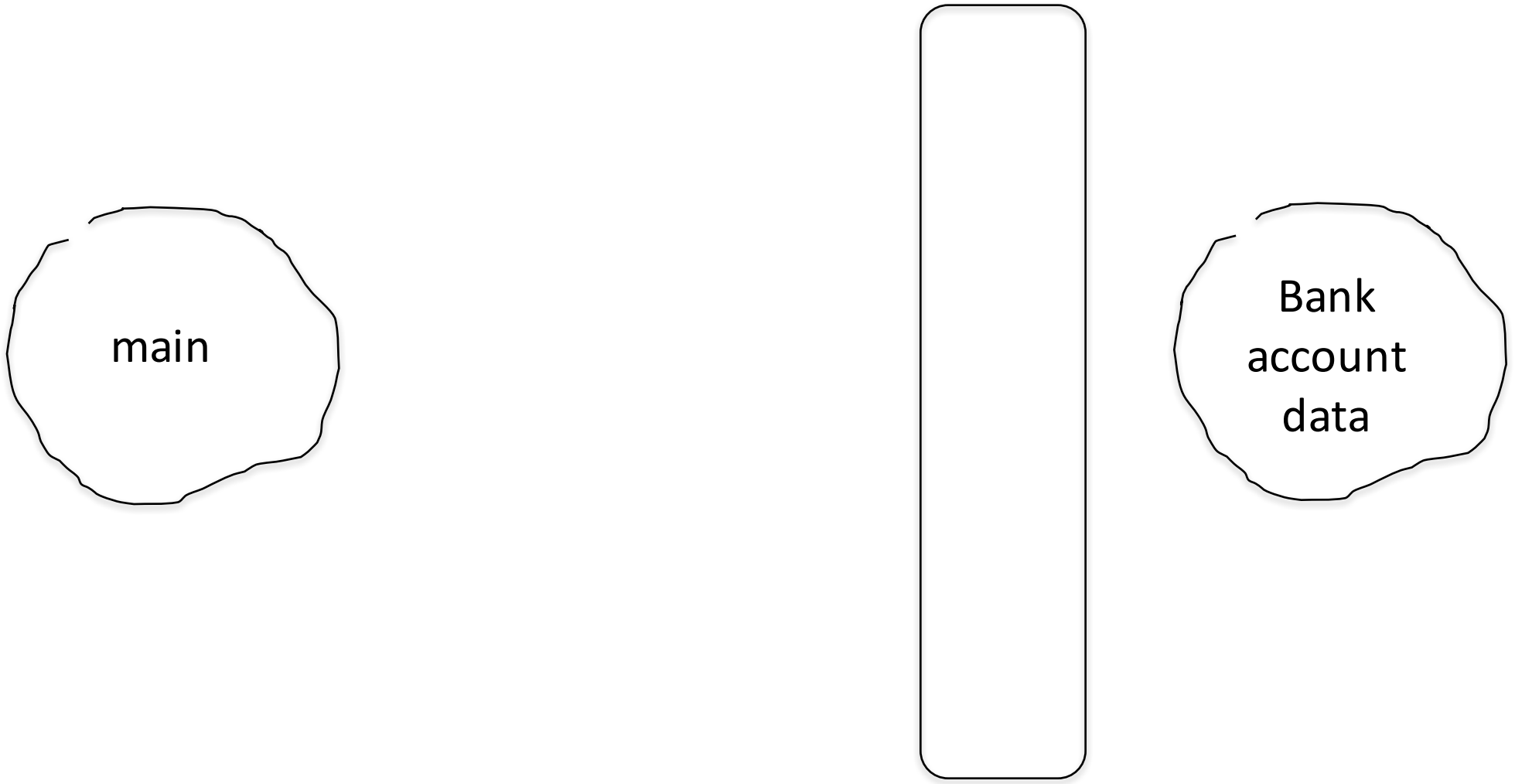


Bank Account parameter?

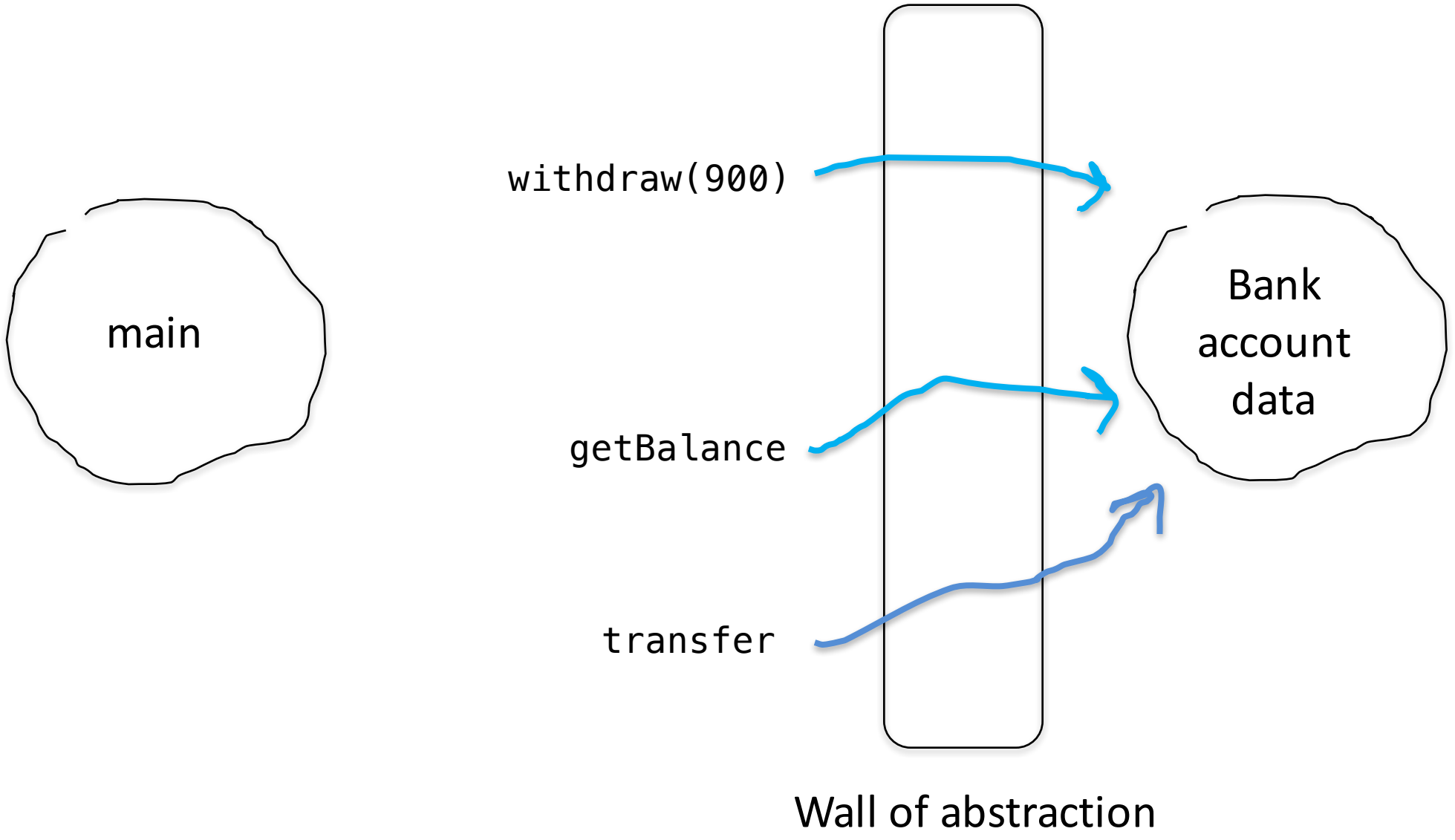
Vector<BankAccount>?

If structs are so wonderful, why would they want something better?

Encapsulation

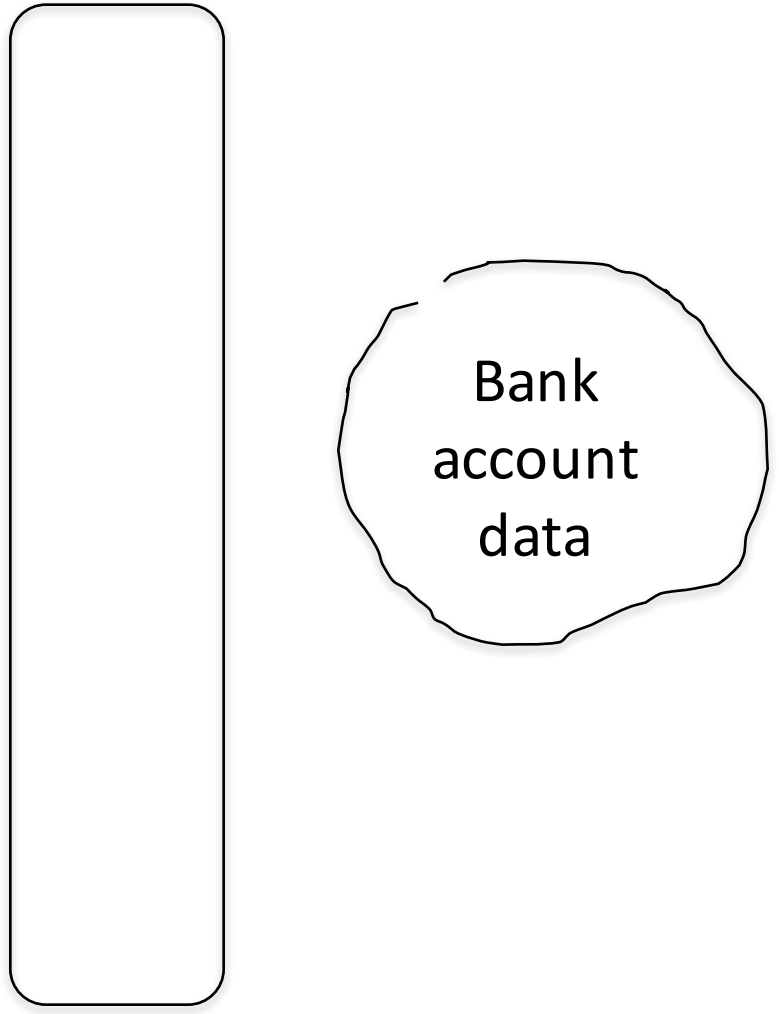


Encapsulation



Encapsulation

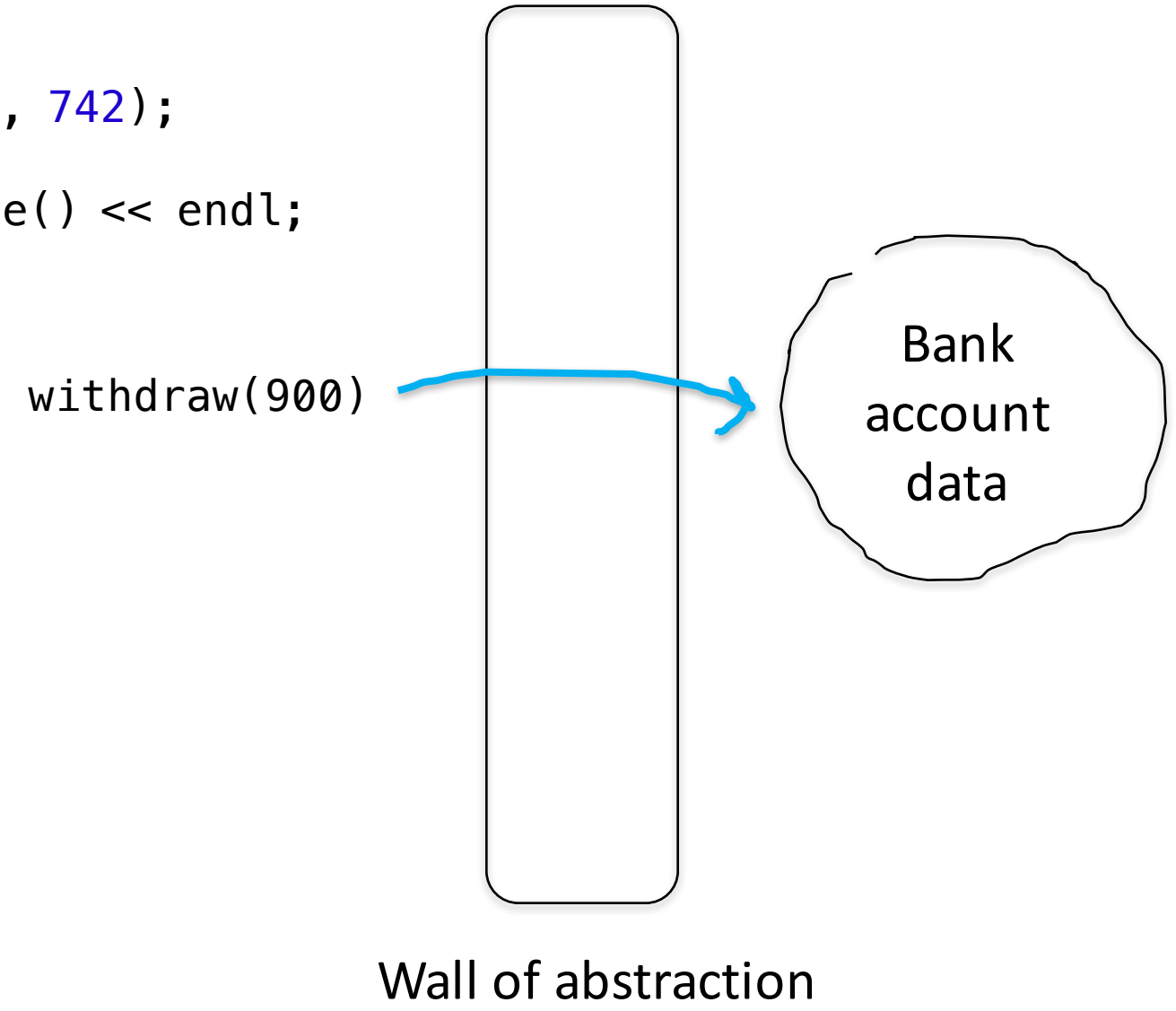
```
int main() {
    BankAccount checking(”Bob", 742);
    checking.withdraw(900);
    cout << checking.getBalance() << endl;
}
```



Wall of abstraction

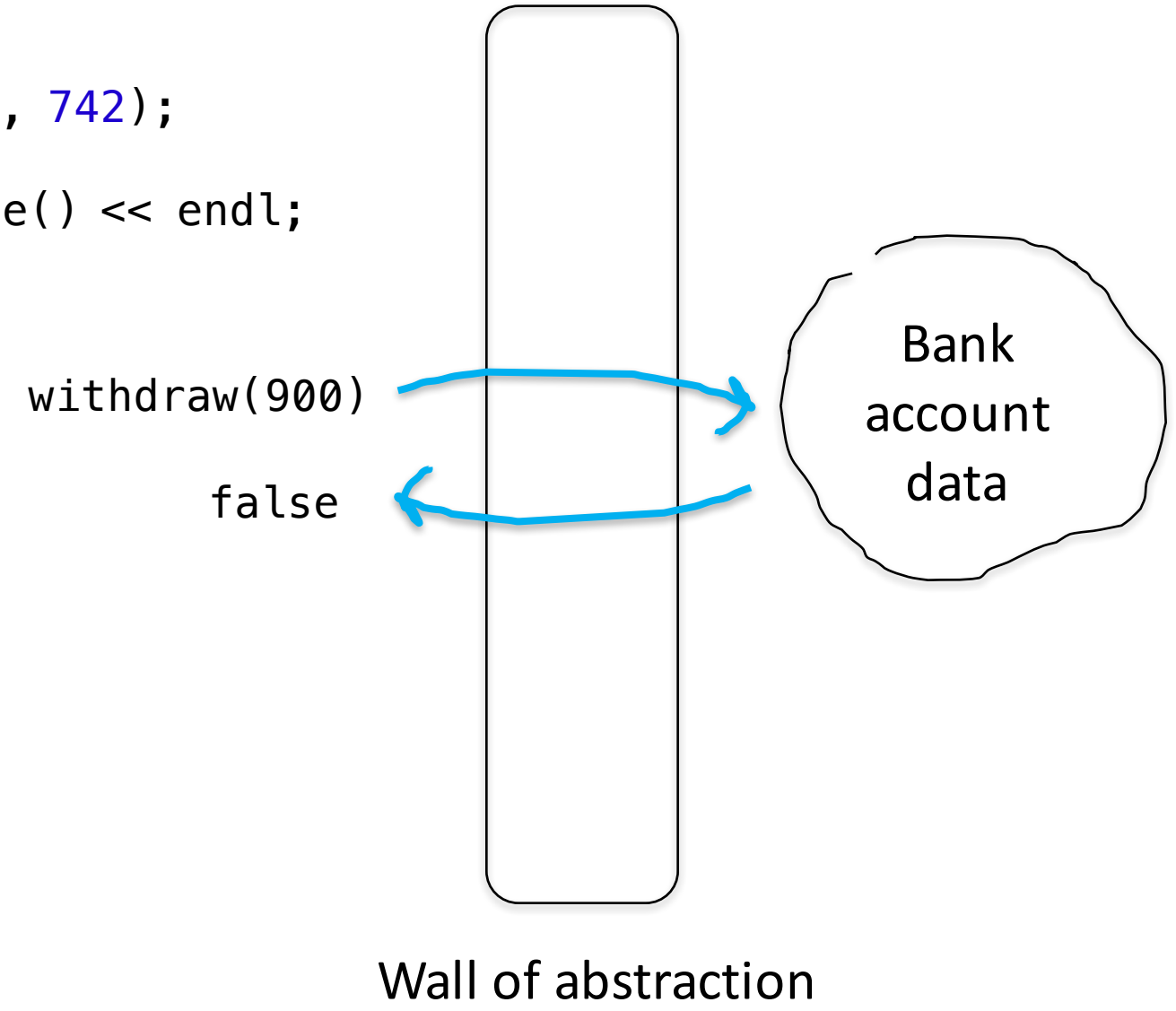
Encapsulation

```
int main() {
    BankAccount checking(”Bob", 742);
    checking.withdraw(900);
    cout << checking.getBalance() << endl;
}
```



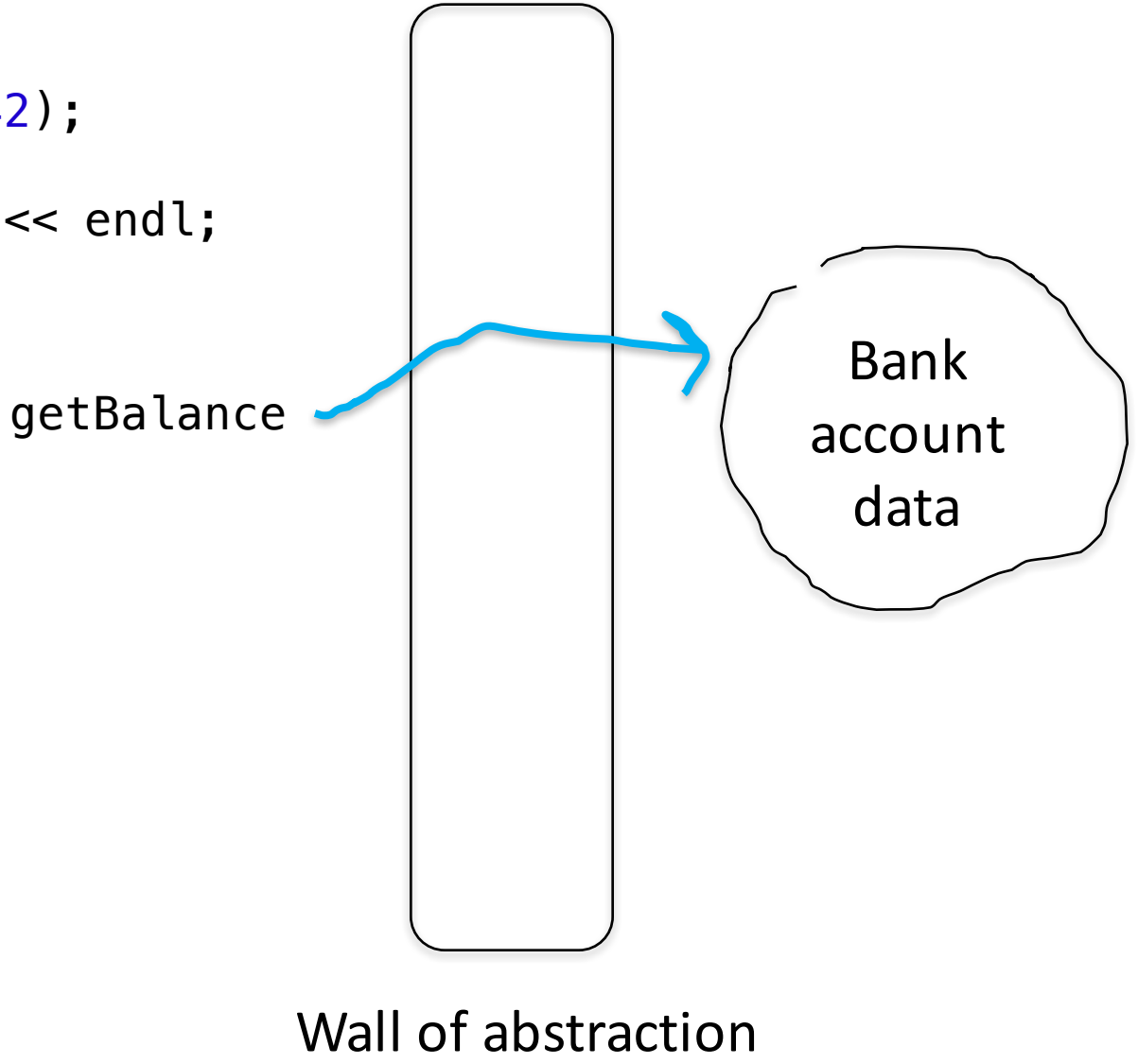
Encapsulation

```
int main() {
    BankAccount checking(”Bob", 742);
    checking.withdraw(900);
    cout << checking.getBalance() << endl;
}
```



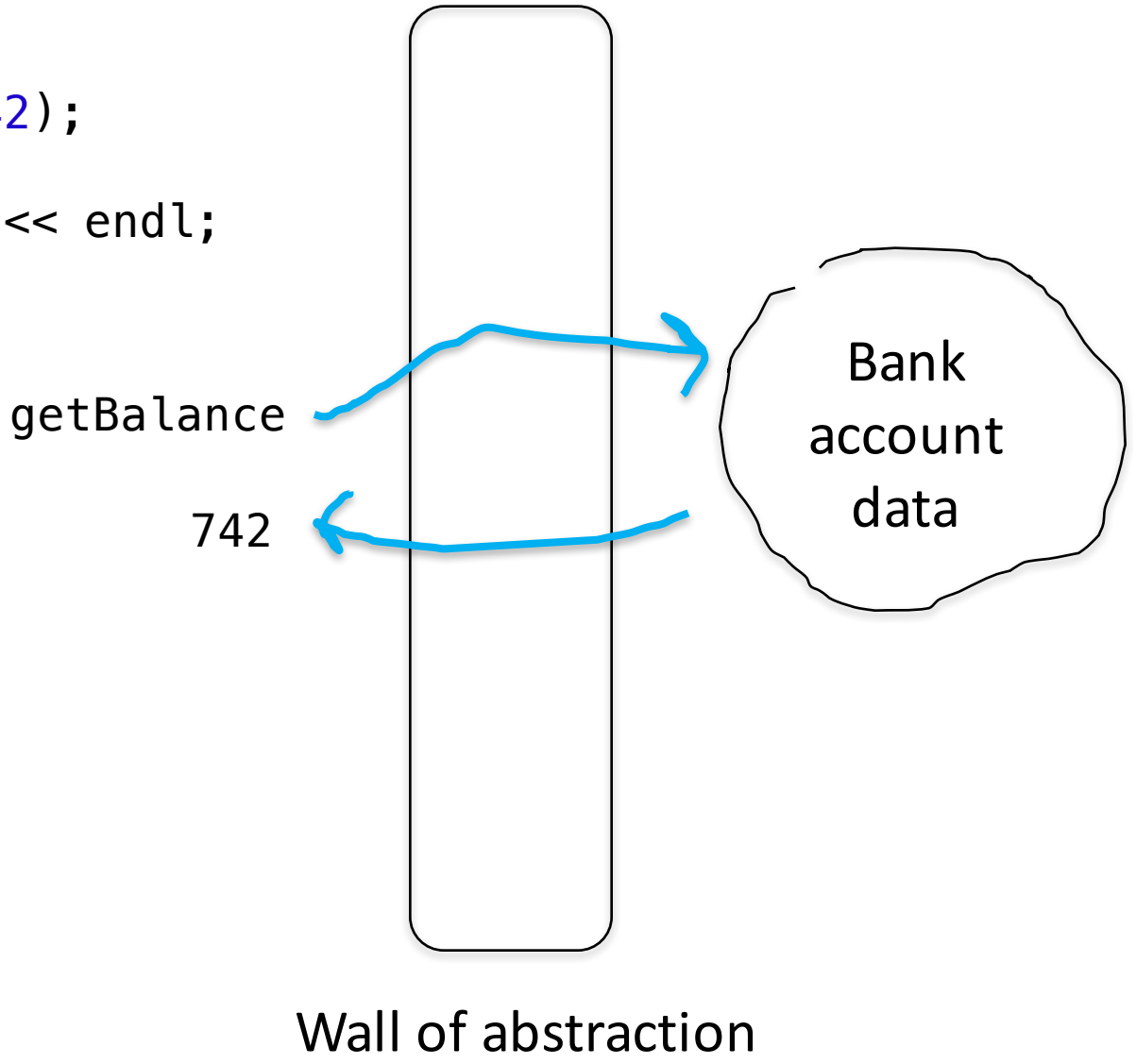
Encapsulation

```
int main() {
    BankAccount checking(”Bob", 742);
    checking.withdraw(900);
    cout << checking.getBalance() << endl;
}
```



Encapsulation

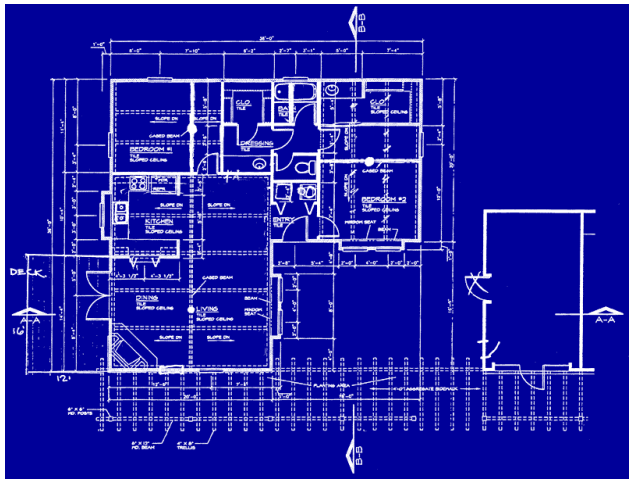
```
int main() {
    BankAccount checking(”Bob", 742);
    checking.withdraw(900);
    cout << checking.getBalance() << endl;
}
```



Classes

class: A template for a new type of variable.

A blueprint is a helpful analogy

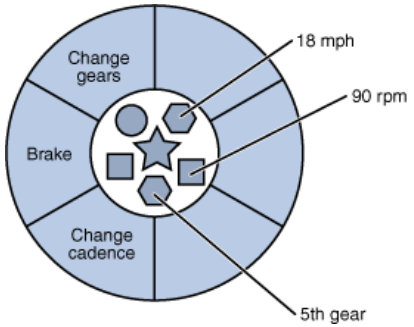


It is under the hood a super struct

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.

Class Interface Devide

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

Structure of a .h

```
// classname.h
#pragma once

class declaration;
```


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

Structure of a .h

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

class declaration;

#endif
```

Exact same thing… just nastier syntax

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)

Bank Account V1

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

#pragma once

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

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

What about the nice functions?

Bank Account V1

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

#pragma once

class BankAccount {
public:
    bool withdraw(double money); // our first function
    string name;         // each BankAccount object
    double balance;      // has a name and balance
};
```

Member Functions!

- 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.

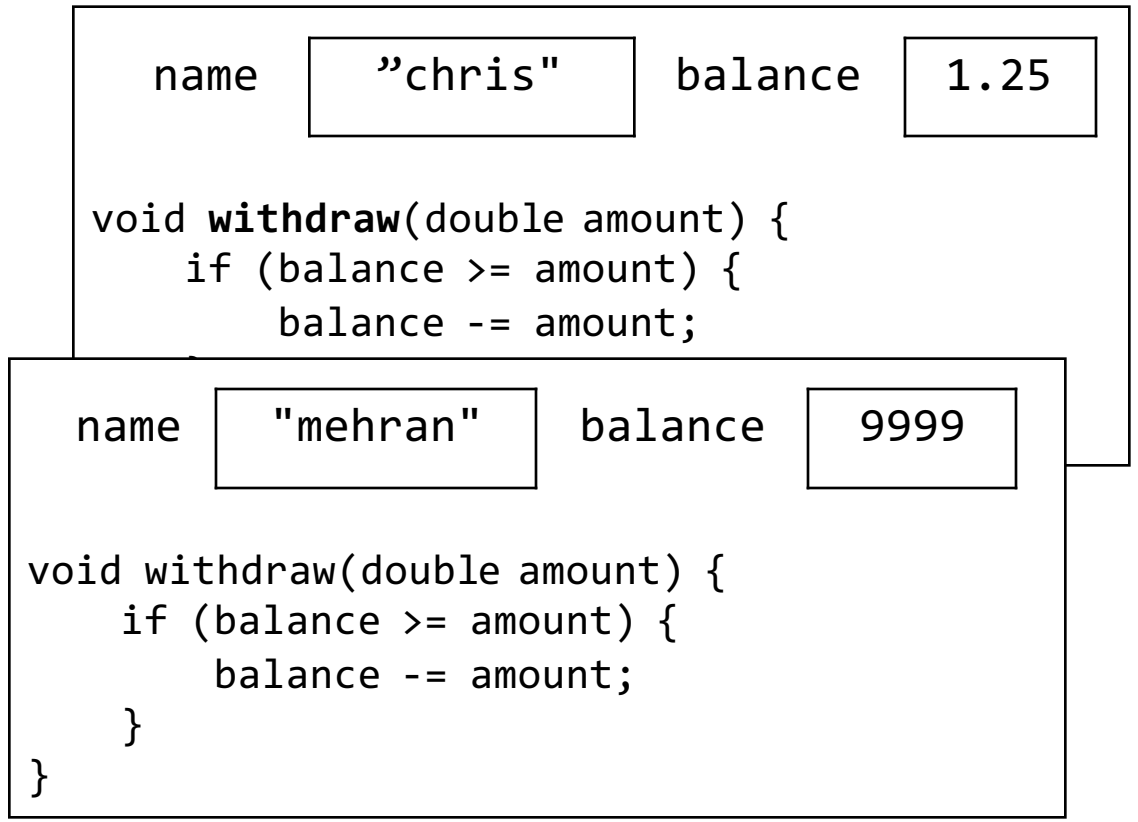
The Implicit Parameter

- **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 that object's 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 Function Diagram

```
// BankAccount.cpp
bool BankAccount::withdraw(double amount) {
    if (balance >= amount) {
        balance -= amount;
        return true;
    }
    return false;
}
```

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

mehran.withdraw(99.00);
```



What about *constructing* a new one?

Initialization

- It's annoying 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

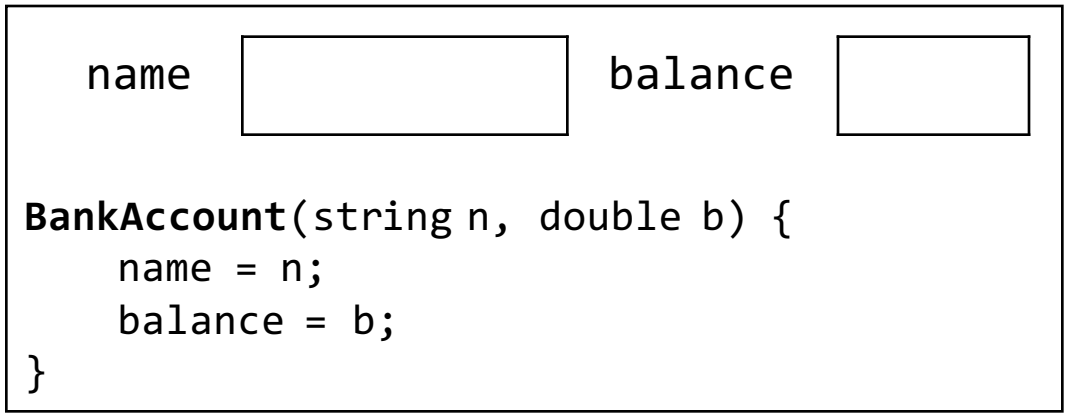
```
ClassName::ClassName(parameters) {
    statements to initialize the object;
}
```

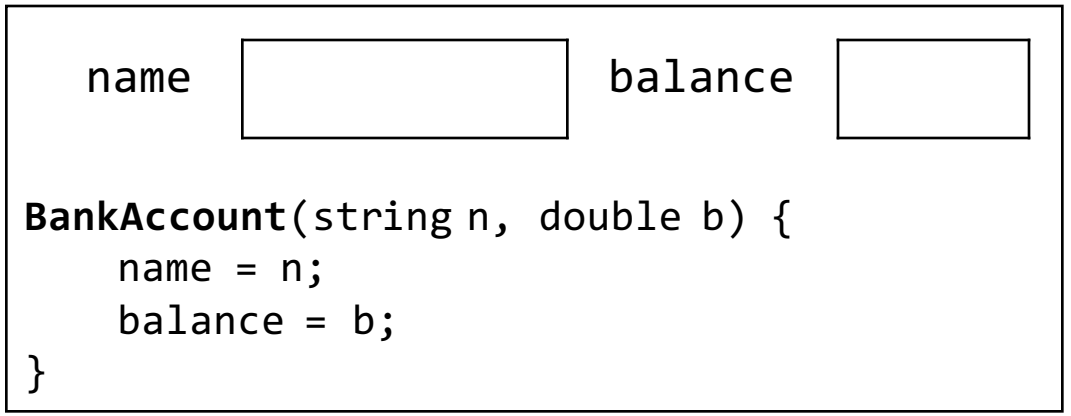
- **constructor**: Initializes state of new objects as they are created.
  - runs when the client declares a new object
  - no return type is specified;
    it implicitly "returns" the new object being created
  - If a class has no constructor, C++ gives it a *default constructor* with no parameters 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;
}
```

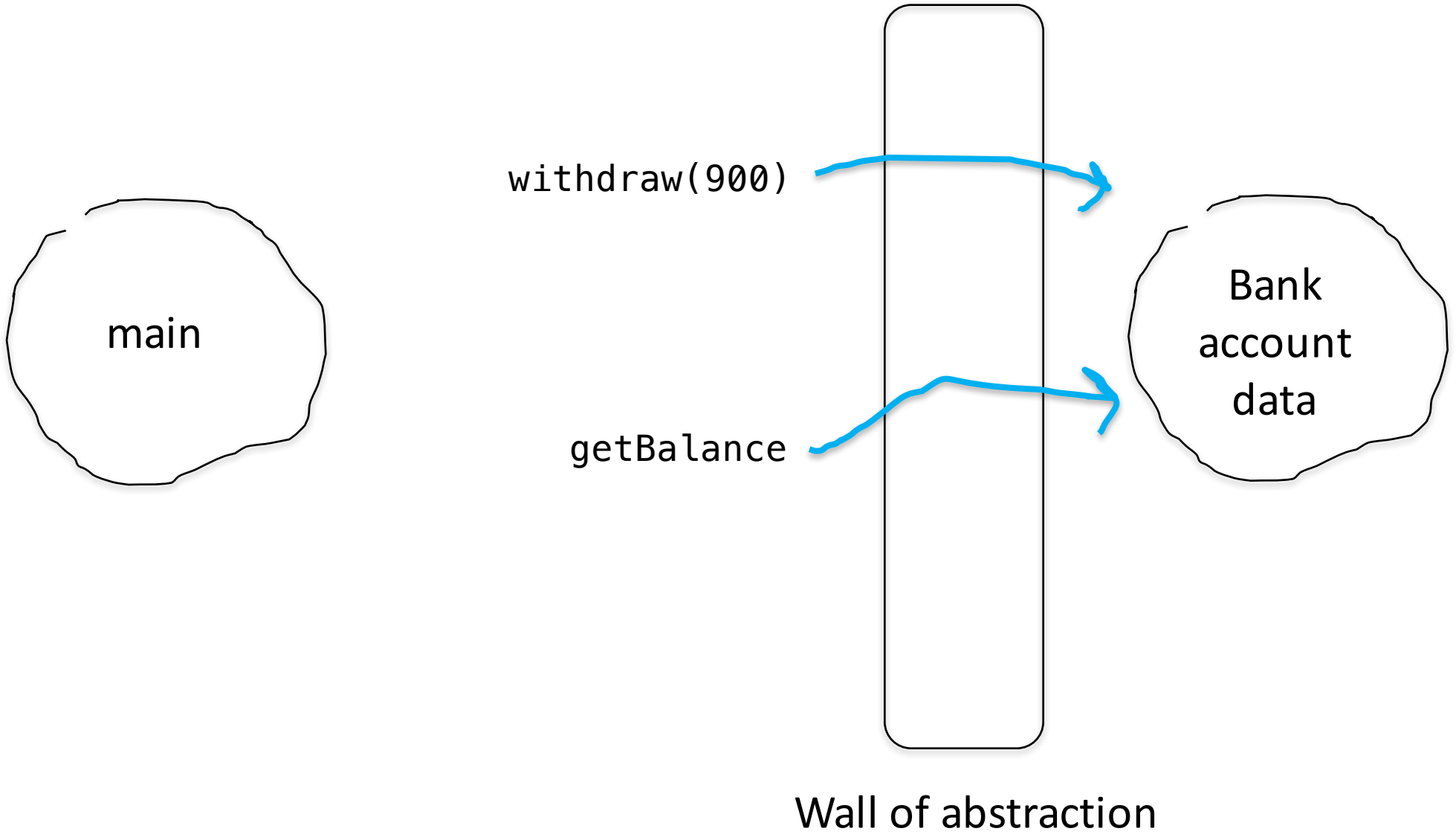
`this` uses `->` not `.` because it is a "pointer"; we'll discuss that later

A Broken Promise

- **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 can **throw 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;
}
```

Encapsulation?



Adding Privacy

```
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

- 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.

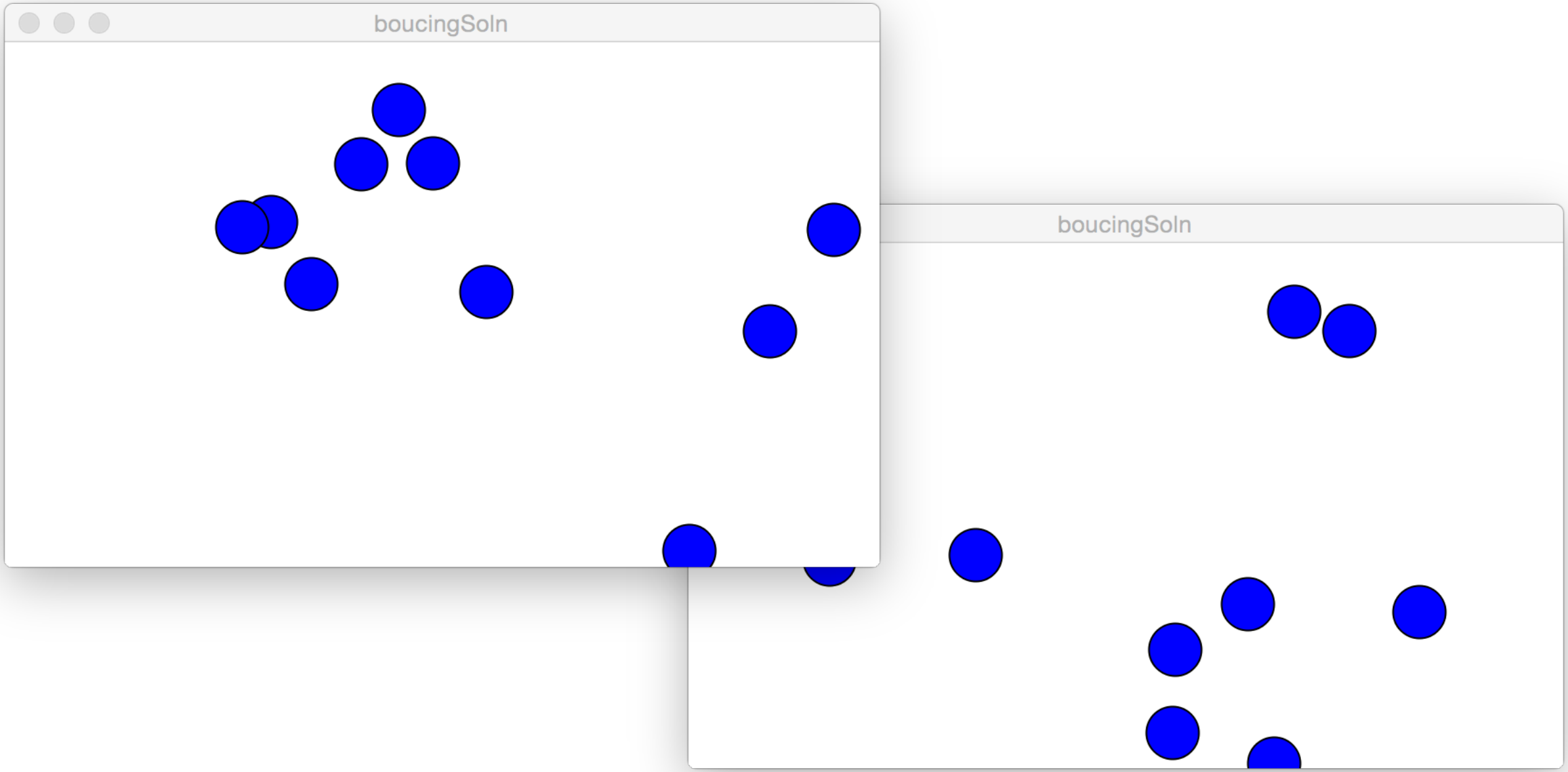
Extra Example: Calendar

C++ has no Dates ☹

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;
}
```

Bouncing Ball



Today's Goal

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