Programming Abstractions

CS106B

Cynthia Bailey Lee
Julie Zelenski

Topics:

- **Classes**
  - Introduction to classes and object-oriented programming
  - Practice making our own classes

Course plan for the next few weeks

We have *used* many classes (our ADT implementations) made by others:

- Vector, Grid, Stack, Queue, Map, Set, Lexicon, GWindow, GPoint...

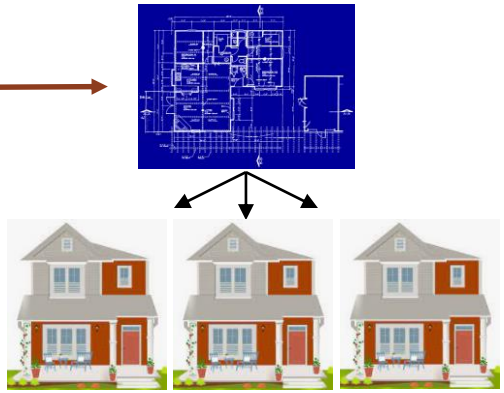
Now let's explore how to *make* a class of our own.

Classes and Objects

KEY VOCABULARY AND CONCEPTS

Classes and objects

- **Class:** Allows us to add new types to the language!
  A template for what the type holds and how it works



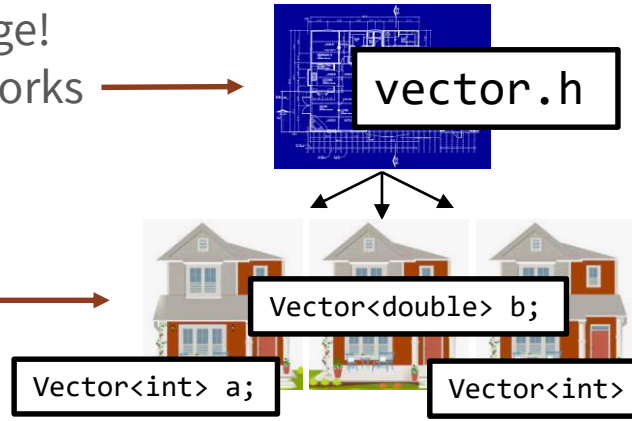
- **Object:** One instance of a class type

- **Object-oriented programming (OOP):** Programs that perform their behavior as interactions between objects.

- **Abstraction:** Separation between concepts and details.

Classes and objects

- **Class:** Allows us to add new types to the language!
  A template for what the type holds and how it works



- **Object:** One instance of a class type

- **Object-oriented programming (OOP):** Programs that perform their behavior as interactions between objects.

- **Abstraction:** Separation between concepts and details.

Elements of a class

Member variables: **State** inside each object

- Also called "instance variables" or "fields"
- Each object has a copy of each member variable

Member functions: **Behavior** each object can perform

- Also called "methods"
- The method can interact with the data inside that object

Abstraction: Interface vs. code

C++ separates classes into two kinds of code files:

- .h: A "header" file containing the interface (declarations)
- .cpp: A "source" file containing definitions (method bodies)

  - class Foo => **must write both `foo.h` and `foo.cpp`**

The content of .h files is #included inside .cpp files

- Makes them aware of the blueprint plans for the class and its members

Abstraction: Interface vs. code

C++ separates classes into two kinds of code file

- .h: A "header" file containing the interface (declarations)
- .cpp: A "source" file containing definitions (method bodies)

  - class Foo => **must write both `foo.h` and `foo.cp`**

Essentially a collection of function prototypes for the class methods (among other things)

The actual function definitions

The content of .h files is #included inside .cpp files

- Makes them aware of the blueprint plans for the class and its members

C++ Class Implementation

HOW TO ACTUALLY DO THIS!

Class declaration (.h)

```
#ifndef _classname_h
#define _classname_h

class ClassName {
public:                             // in ClassName.h
    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)
};

#endif
```

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

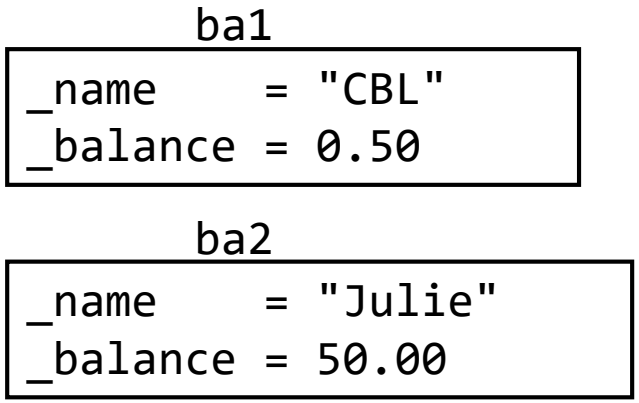
***IMPORTANT:** must put a semicolon at end of class declaration*

Class example (v1)

```
// BankAccount.h

#ifndef _bankaccount_h
#define _bankaccount_h

class BankAccount {
public:
    BankAccount(string n);     // constructor

    void deposit(double amount);      // methods
    void withdraw(double amount);
    void setName(string name);

private:
    string _name;      // each BankAccount object
    double _balance;   // has a name and balance
};

#endif
```

Using objects

```
// client code in bankmain.cpp
BankAccount ba1("Cynthia");
ba1.deposit(2.00);
ba1.withdraw(1.50);
ba1.setName("CBL");

BankAccount ba2("Julie");
ba2.deposit(60.00);
ba2.withdraw(5.00);
ba2.withdraw(5.00);
```



An object groups multiple variables together

- Each object contains its own name and balance field inside it
- We can get/set them individually
- Code that uses your objects is called client code

Member function bodies

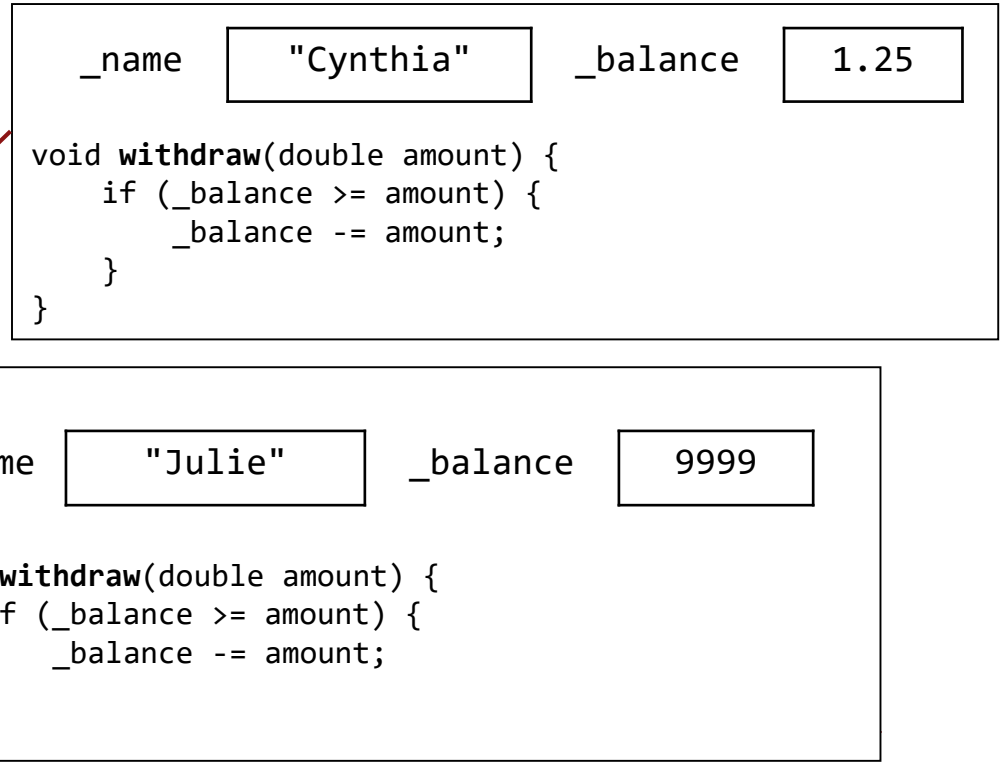
In `ClassName.cpp`, we write bodies (definitions) for the member functions that were declared in the .h file:

```
#include "ClassName.h"

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

- Member functions/constructors can refer to the object's member variables.

Member func diagram

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


// client program
BankAccount cynth(...);
BankAccount julie(...);
...
cynthia.withdraw(5.00);

julie.withdraw(5.00);
```



Constructors

```
ClassName::ClassName(parameters) {   // note no return type is specified
    statements to initialize the object;
}
```

Constructor: Initializes state of new objects as they are created.

- without constructor:

```
BankAccount ba;
ba.setName("Cynthia");      // tedious, also what is the balance??
```

- with constructor:

```
BankAccount::BankAccount(string name) {
    _name = name;
    _balance = 0.0;
}

BankAccount ba("Cynthia");  // convenient, clearly starts $0.0 balance
```

Private data

```
private:
    type name;
```

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;
}
```

Your Turn!

I want to add a second constructor to my BankAccount class

- Current constructor takes the name and initializes
- I'd like to have one that takes both a name and an initial account balance

In PollEv: write the line of code I would need to add to the .h file to do this.
In discussion: what new code goes in the new .cpp file?

```
// BankAccount.h

#ifndef _bankaccount_h
#define _bankaccount_h

class BankAccount {
public:
    BankAccount(string n);      // constructor

    void deposit(double amount);       // methods
    void withdraw(double amount);

private:
    string _name;       // each BankAccount object
    double _balance;    // has a name and balance
};

#endif
```

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) {
        error("Balance must be positive.");
    }
    _name = name;
    _balance = balance;
}
```

Bouncing Ball Demo

APPLYING WHAT WE LEARNED WITH THE BANK CLASS TO A NEW PROBLEM

Bouncing Ball demo

Write a class Ball that represents a bouncing ball.

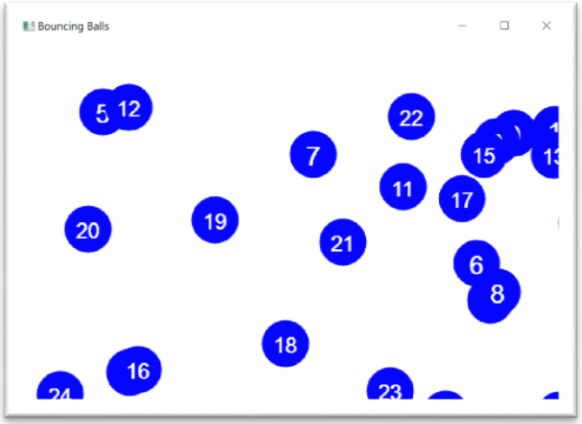
- What state (private instance variables) should each ball store?
- window functions: setColor and drawOval

Finish the provided client code to draw many balls in a window.

- Make each ball appear at a random location.
- Make the balls move at random velocities and "bounce" if they hit window edges.

Enhance the provided client code to add colors.

- Make each ball appear a random color choice.



Extra Slides

MORE COOL TRICKS WITH C++ CLASSES

Operator overloading (6.2)

operator overloading: Redefining the behavior of a common operator in the C++ language.

unary:	`+ - ++ -- * &`
	`! ~ new delete`
binary:	`+ - * / % += -=`
	`*= /= %= & | && ||`
	`^ == != < > <= >=`
	`<< >> = [] -> () ,`

Syntax:

```
returnType operator op(parameters);       // in the .h file for the class

returnType operator op(parameters) {      // in the .cpp file for the class
    statements;
};
```

- For example, for two variables of type Foo, **`a + b`** will use the code you write in:

```
Foo operator +(Foo& a, Foo& b) {
    // function body
}
```

Make objects printable

To make it easy to print your object to cout, overload <<

```
ostream& operator <<(ostream& out, Type& name) {
    statements;
    return out;
}
```

- `ostream` is a base class that represents cout, file output streams, ...

<< overload example

```
// BankAccount.h
class BankAccount {
   ...
};
// notice operators go OUTSIDE of the class' closing }; brace!
ostream& operator <<(ostream& out, BankAccount& ba);
```

```
// BankAccount.cpp
ostream& operator <<(ostream& out, BankAccount& ba) {
    out << ba.getName() << ": $" << ba.getBalance();
    return out;
}
```

== overload example

```
// BankAccount.h
class BankAccount {
   ...
};

bool operator ==(const BankAccount& ba1,
                 const BankAccount& ba2);
```

```
// BankAccount.cpp
bool operator ==(const BankAccount& ba1,
                 const BankAccount& ba2) {
    return ba1.getName() == ba2.getName()
        && ba1.getBalance() == ba2.getBalance();
}
```

Destructor (12.3)

```
// ClassName.h
~ClassName();
```

```
// ClassName.cpp
ClassName::~ClassName() { ...
```

Destructor: Called when the object is deleted by the program.

- (when the object falls out of {} scope)

- Useful if your object needs to free any memory as it dies.
  - delete any pointers stored as private members
  - delete[] any arrays stored as private members
  - *(we haven't learned about delete yet, that's in a couple weeks!)*