Programming Abstractions

CS106B

Cynthia Lee
Topics:

- **Classes**
  - Introduction to objects and object-oriented programming
  - Practice making our own classes
Course plan for the next few weeks

We have *used* many collections:
- Vector, Grid, Stack, Queue, Map, Set, Lexicon, ...

Now let's explore how they are implemented.
ADT implementations

We have used many collections:
- Vector, Grid, Stack, Queue, Map, Set, Lexicon, ...

Now let's explore how they are implemented.
- We must learn about classes, arrays, and memory.
- We will implement several collections:
  - stack, priority queue, linked list, set, map, hash set/map, graph, ...
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

**Member functions:** **Behavior** inside each object
- Also called "methods"
- 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
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 class and its members
#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

**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, double d); // constructor
    void deposit(double amount);    // methods
    void withdraw(double amount);

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

#endif
Using objects

// client code in bankmain.cpp
BankAccount ba1("Cynthia", 1.25);
ba1.deposit(2.00);

BankAccount ba2("Katherine", 99.00);
ba2.withdraw(5.00);

An object groups multiple variables together
- 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
Member func. bodies

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

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

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

- Member functions/constructors can refer to the object's member variables.
void BankAccount::withdraw(double amount) {
    if (_balance >= amount) {
        _balance -= amount;
    }
}

// client program
BankAccount cynth(...);
BankAccount kath(...);
...
cynthia.withdraw(5.00);
katherine.withdraw(5.00);
Constructors

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

**Constructor**: Initializes state of new objects as they are created.
- no return type is specified; implicitly "returns" the new object
- without constructor:
  ```
  BankAccount ba;
  ba._name = "Cynthia";
  ba._balance = 1.25; // tedious
  ```
- with constructor:
  ```
  BankAccount ba("Cynthia", 1.25); // better
  ```
Private data

private:
    type name;

**Encapsulation:** Hiding implementation details from client code.

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

```cpp
// "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;
}
```
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;
    }
    _name = name;
    _balance = balance;
}
The keyword const

Just like a const reference parameter can't be modified by the function:

```c
void foo(const BankAccount& ba) {
```

A const member function can't change the object's state:

```c
class BankAccount { ...
    double getBalance() const;
```
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.
Operator overloading (6.2)

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

Syntax:

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

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

- For example, for two variables of type Foo, \( a + b \) will use the code you write in:
  ```cpp
  Foo operator +(Foo& a, Foo& b) {
    // function body
  }
  ```

### unary:
+ - ++ -- * &
! ~ new delete

### binary:
+ - * / % += -=
*= /= %= & | && ||
^ == != < > <= >=
<< >> = [] -> ( ) ,
Make objects printable

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

```cpp
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.cpp
~ClassName(); 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 next week!)*