Classes

CS 106B

Programming Abstractions
Fall 2016
Stanford University
Computer Science Department
Announcement

Chris Piech office hours moved to tomorrow.

Midterm materials online this afternoon.
Midterm Review

Sunday morning review

Handouts today on the website
1. Learn how to define a class in C++
Some *large* programs are in C++
almost all the code is written in C++.
- Sebastian Thrun
How?
Decomposition Across Files

Motor Controller

Collision Detector

Route Planner

GPS Point

Physical Object

Path
• 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.
venmo

The easiest way to pay your friends.
Venmo needs to store info about peoples bank accounts, but C++ has no `BankAccount` type.
• Venmo needs to store info about peoples bank accounts, but C++ has no **BankAccount** type.

```cpp
struct BankAccount {
    string name;
    double balance;
};
```
Venmo needs to store info about peoples bank accounts, but C++ has no `BankAccount` type.

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

int main() {
    int n = 3;
    BankAccount account;
}
```
Venmo needs to store info about peoples' bank accounts, but C++ has no **BankAccount** type.

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

int main() {
    int n = 3;
    BankAccount account;
    account.name = "Alyssa";
    account.balance = 25;
    cout << account.balance << endl;
}
```
• Venmo needs to store info about peoples bank accounts, but C++ has no `BankAccount` type.

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

int main() {
    int n = 3;
    BankAccount account;
    account.name = "Alyssa";
    account.balance = 25;
    cout << account.balance << endl;
}
```
• Venmo needs to store info about peoples bank accounts, but C++ has no **BankAccount** type

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

int main() {
    int n = 3;
    BankAccount account;
    account.name = "Alyssa";
    account.balance = 25;
    cout << account.balance << endl;
}
```
Venmo needs to store info about peoples bank accounts, but C++ has no **BankAccount** type.

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

int main() {
    int n = 3;
    BankAccount account;
    account.name = "Alyssa";
    account.balance = 25;
    cout << account.balance << endl;
}
```
Venmo needs to store info about peoples' bank accounts, but C++ has no BankAccount type.

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

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

<table>
<thead>
<tr>
<th></th>
<th>Alyssa</th>
</tr>
</thead>
<tbody>
<tr>
<td>account.name</td>
<td></td>
</tr>
<tr>
<td>account.balance</td>
<td>25</td>
</tr>
<tr>
<td>n</td>
<td>3</td>
</tr>
</tbody>
</table>
• Venmo needs to store info about peoples bank accounts, but C++ has no `BankAccount` type.

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

int main() {
}
```
Venmo needs to store info about peoples' bank accounts, but C++ has no `BankAccount` type.

```cpp
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

main

Wall of abstraction

Bank account data
Encapsulation

main

withdraw(900)

getBalance

transfer

Bank account data

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

Bank account data

withdraw(900)

Encapsulation

Wall of abstraction
int main() {
    BankAccount checking("Bob", 742);
    checking.withdraw(900);
    cout << checking.getBalance() << endl;
}
int main() {
    BankAccount checking("Bob", 742);
    checking.withdraw(900);
    cout << checking.getBalance() << endl;
}
```cpp
int main() {
    BankAccount checking("Bob", 742);
    checking.withdraw(900);
    cout << checking.getBalance() << endl;
}
```
**class**: A template for a new type of variable.

A blueprint is a helpful analogy.

It is under the hood a super struct.
**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

<table>
<thead>
<tr>
<th>Interface</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>name.h</em></td>
<td><em>name.cpp</em></td>
</tr>
<tr>
<td>Client reads</td>
<td>Implementer writes</td>
</tr>
<tr>
<td>Shows methods and states instance variables</td>
<td>Implements methods</td>
</tr>
</tbody>
</table>
// 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
// classname.h
#ifndef _classname_h
#define _classname_h

class declaration;

#endif

Exact same thing... just nastier syntax
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)
// 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;

• Think of a 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?
// 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
};
• In `ClassName.cpp`, we write bodies (definitions) for the member functions that were declared in the `.h` file:

```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.
// 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?
• It's annoying to take 3 lines to create a BankAccount and initialize it:

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

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

```cpp
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.
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:

```cpp
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.
• **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**.

```cpp
// 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?

main

withdraw(900)

getBalance

Bank account data

Wall of abstraction
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.
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");
  ```
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 😞
```cpp
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
1. Learn how to define a class in C++