CS 106B
Lecture 13: Classes
Monday, April 29, 2018

Programming Abstractions
Spring 2018
Stanford University
Computer Science Department

Lecturer: Chris Gregg

reading:
Programming Abstractions in C++, Chapter 6
Today's Topics

• Logistics
  • Everyone who has contacted us about an alternate exam has been contacted — if you haven't heard yet, please reach out to us.
  • Review Session: Tuesday 7-8:30pm, 420-040
  • Midterm: Last names A-H in Bishop Auditorium, I-Z in Hewlett 200
  • Bluebook demo

• Assignment 4: Boggle!

• Classes
  • What are they and why are they important?
  • Bouncing Balls
  • Decomposition
  • Encapsulation
  • Elements of a Class, Header files
  • public / private
  • Constructors / Destructors
  • The keyword "this"
  • The Fraction Class
  • operator overloading
Assignment 4: Boggle

A classic board game with letter cubes (dice) that is not dog friendly: https://www.youtube.com/watch?v=2shOz1ZLw4c
In Boggle, you can make words starting with any letter and going to any adjacent letter (diagonals, too), but you cannot repeat a letter-cube.
Bouncing Balls Demo
Introduction to Classes

Remember how we said that structs are the Lunchables of the C++ world?

Structs gave us the ability to package *data* into one place:

```cpp
struct Lunchable {
    string meat;
    string dessert;
    int numCrackers;
    bool hasCheese;
};
```
But why stop at data? If we're packaging stuff up, let's also package up the functions.

It would be really nice if we could do this:

```cpp
struct Lunchable {
    string meat;
    string dessert;
    int numCrackers;
    bool hasCheese;
    int countCalories();
};
```
Guess what? **We can do this!** Even in structs!
(but not in C, only in C++)

Once we have the ability to package up data *and* functions into one structure, we have a super-powerful tool, called an "**object**" that *knows how to perform functions on itself, and carries around its own data.*

So-called "Object Oriented Programming" has led to the creation of most of the large programs we use today.
A C++ "class" is simply a very-slightly modified struct (details to follow).

As with structs, we sometimes want new 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.
• The *only* difference between a struct and a class is the notion of defaults regarding *encapsulation*. A struct defaults to "public" members, and a class defaults to "private" members.

• "Encapsulation" allows the designer of a class to build a "wall of abstraction" around her data:
The only difference between a struct and a class is the notion of defaults regarding encapsulation. A struct defaults to "public" members, and a class defaults to "private" members.

"Encapsulation" allows the designer of a class to build a "wall of abstraction" around her data:

```cpp
int main() {
    BankAccount checking("Bob", 42);
    checking.withdraw(80);
    cout << checking.getBalance() << endl;
}
```
The only difference between a struct and a class is the notion of defaults regarding encapsulation. A struct defaults to "public" members, and a class defaults to "private" members.

"Encapsulation" allows the designer of a class to build a "wall of abstraction" around her data:

```cpp
int main() {
    BankAccount checking("Bob", 42);
    checking.withdraw(80);
    cout << checking.getBalance() << endl;
}
```

Error: not enough funds!
The only difference between a struct and a class is the notion of defaults regarding encapsulation. A struct defaults to "public" members, and a class defaults to "private" members.

"Encapsulation" allows the designer of a class to build a "wall of abstraction" around her data:

```cpp
int main() {
    BankAccount checking("Bob", 42);
    checking.withdraw(80);
    cout << checking.getBalance() << endl;
}
```

Wall of abstraction
The reason we want encapsulation is so that the end user of our class does not have direct access to the data -- we, as the class designer, control the data completely:

```cpp
int main() {
    BankAccount checking("Bob", 42);
    checking.withdraw(80);
    cout << checking.getBalance() << endl;
    checking.balance = 81.2345;
}
```

If we allowed this, our internal class data might not be in a state we can handle (e.g., too many decimal places for a monetary value)
So, we block the ability for someone using our class to directly touch the data, and we force them to go through our own functions:

```cpp
int main() {
    BankAccount checking("Bob", 42);
    checking.withdraw(80);
    cout << checking.getBalance() << endl;
    checking.setBalance(81.2345);
}
```

Because we control the function, we can do a check on this, and enforce the "only two decimals" limit.
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 Divide

**Interface**

*name.h*

Client reads

Shows methods and states instance variables

**Source**

*name.cpp*

Implementer writes

Implements methods
Structure of a .h file

// classname.h
#pragma once

class ClassName {
    // class definition
};

This basically says, "if you see this file more than once while compiling, ignore it after the first time" (so the compiler doesn't think you're trying to define things more than once)
// classname.h
#pragma once

class ClassName {
    // class definition
};

This basically says, "if you see this file more than once while compiling, ignore it after the first time" (so the compiler doesn't think you're trying to define things more than once)

Older format, not as nice:

// classname.h
#ifndef _CLASSNAME_H
#define _CLASSNAME_H

class ClassName {
    // class definition
};

#endif

#endif

#define _CLASSNAME_H

// in ClassName.h
class ClassName {
public:
    ClassName(parameters); // constructor
    returnType func1(parameters); // member functions
    returnType func2(parameters); // (behavior inside
    returnType func3(parameters); // each object)

private:
    type var1; // member variables
    type var2; // (data inside each object)
    type func4(); // (private function)
};
Encapsulation defined in .h

// in MyClass.h
class MyClass {
public:
    MyClass(parameters); // constructor
    returnType func1(parameters); // member functions
    returnType func2(parameters); // (behavior inside
    returnType func3(parameters); // each object)

private:
    type var1; // member variables
    type var2; // (data inside each object)
    type func4(); // (private function)
};

Any class instance can directly use anything defined as public (but you never directly call a constructor):

    MyClass a;
    a.func1(arguments)
Encapsulation defined in .h

// in MyClass.h
class MyClass {
public:
    MyClass(parameters); // constructor
    returnType func1(parameters); // member functions
    returnType func2(parameters); // (behavior inside
    returnType func3(parameters); // each object)

private:
    type var1; // member variables
    type var2; // (data inside each object)
    type func4(); // (private function)
};

Class instances can **not** directly use anything defined as
private:

MyClass a;
a.var1 = 2; // error!
Constructors and (eventually) Destructors

// in MyClass.h
class MyClass {
public:
    MyClass(); // default constructor
    MyClass(parameters); // constructor
    ...
};

When a class instance is created, we say that it is "constructed":

string s1; // uses default constructor

string s2("I'm a string"); // uses a constructor
    // that takes 1 string parameter

string s3 = "I'm a string"; // different! (we'll get to that)
**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 `aaron.withdraw(...)`, the object named `aaron` 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.
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
As an example of a class, we're going to define a `Fraction` class that can deal with rational numbers directly, without decimals.

We are going to walk through the class one step at a time, demonstrating the various parts of a class as we go.
The Fraction Class

• Questions we must answer about the Fraction class:
  • What data should the class hold?
  • What kinds of functions (public / private) should our class have?
  • What constructors could we have?
    • What is a good value for a default fraction?
The Fraction Class

Class outline

class Fraction {
public:
    Things we want class users to see
private:
    Things we want to keep hidden from class users
};
class Fraction {
public:

private:
    int num;  // the numerator
    int denom;  // the denominator

};

The Fraction Class

Class outline

What data would a Fraction class have?

Why is it private?
The Fraction Class

What functions should a fraction class be able to do?

Why are they public?

What is this???
The Fraction Class

class Fraction {
public:
    void add(const Fraction &f);
    void mult(const Fraction &f);
    double decimal();
    int getNum();
    int getDenom();
    friend ostream& operator<<(ostream& out, Fraction &frac);

private:
    int num; // the numerator
    int denom; // the denominator
};

What is this???

This defines an operator "overload" to make it possible to use the "<<" operator with cout.

We will write this function in a few minutes.
We need to *construct* the class when it is called.

What should a "default" fraction look like?

1 / 1 probably makes the most sense (why not 0/0?)

Should we let the user create an initial fraction, e.g., 3/4?
class Fraction {
public:
    Fraction();
    Fraction(int num, int denom);
    void add(const Fraction &f);
    void mult(const Fraction &f);
    double decimal();
    int getNum();
    int getDenom();
    friend ostream& operator<<(ostream& out, Fraction &frac);

private:
    int num; // the numerator
    int denom; // the denominator
    void reduce(); // reduce the fraction
    int gcd(int u, int v);
};

Any other functions?

What about reduce?
(necessary for multiplication)

Reduce needs gcd()...
```cpp
#pragma once
#include <ostream>
using namespace std;

class Fraction {
public:
    Fraction();
    Fraction(int num, int denom);
    void add(const Fraction &f);
    void mult(const Fraction &f);
    double decimal();
    int getNum();
    int getDenom();
    friend ostream& operator<<(ostream& out, Fraction &frac);  
private:
    int num;       // the numerator
    int denom;     // the denominator
    void reduce(); // reduce the fraction
    int gcd(int u, int v);
};
```

The Fraction Class

Last, but not least...
Let's start writing our functions. We do this in our fraction.cpp file, and we have to define the class that each function belongs to. We also cannot forget to include our header file!

```
#include "fraction.h"
```

The **default constructor** is used when someone wants to just create a default fraction:

```
Fraction frac;
```
The Fraction Class

Let's start writing our functions. We do this in our fraction.cpp file, and we have to define the class that each function belongs to. We also cannot forget to include our header file!

```cpp
#include "fraction.h"
```

The **default constructor** is used when someone wants to just create a default fraction:

```cpp
Fraction frac;
```

// purpose: the default constructor
// to create a fraction of 1 / 1
// arguments: none
// return value: none
// (constructors don't return anything)

```cpp
Fraction::Fraction()
{
    // This tells the compiler what class we are creating. The double-colon is called the "scope resolution operator" because it helps the compiler resolve the scope of the function.
}
```
The Fraction Class

Let's start writing our functions. We do this in our fraction.cpp file, and we have to define the class that each function belongs to. We also cannot forget to include our header file!

```
#include "fraction.h"
```

The **default constructor** is used when someone wants to just create a default fraction:

```
Fraction frac;
```

Pretty simple! We are just setting our two class variables to default values.
The Fraction Class

We also have an overloaded constructor that takes in two values that the user sets. It is called as follows:

```
// purpose: an overloaded constructor
//          to create a custom fraction
//          that immediately gets reduced
// arguments: an int numerator
//            and an int denominator
Fraction::Fraction(int num, int denom) {
}
```
The Fraction Class

We also have an overloaded constructor that takes in two values that the user sets. It is called as follows:

```cpp
// purpose: an overloaded constructor
//          to create a custom fraction
//          that immediately gets reduced
// arguments: an int numerator
//            and an int denominator
Fraction::Fraction(int num, int denom) {
    this->num = num;
    this->denom = denom;

    // reduce in case we were given
    // an unreduced fraction
    reduce();
}
```

// create a
// 1/2 fraction
Fraction fracA(1,2);

// create a
// 4/6 fraction
Fraction fracB(4,6);
Let's write some more functions...

// create two fractions
Fraction fracA(1,2);
Fraction fracB(2,3);

fracA.mult(fracB);
// fracA now holds 1/3

// purpose: to multiply another fraction // with this one with the result being
// stored in this fraction
// arguments: another Fraction
// return value: none
void Fraction::mult(const Fraction &other) {

}
Let's write some more functions...

// create two fractions
Fraction fracA(1,2);
Fraction fracB(2,3);

fracA.mult(fracB);
// fracA now holds 1/3

// purpose: to multiply another fraction // with this one with the result being
// stored in this fraction
// arguments: another Fraction
// return value: none
void Fraction::mult(const Fraction &other) {
    // multiplies a Fraction // with this Fraction
    num *= other.num;
    denom *= other.denom;

    // reduce the fraction
    reduce();
}
The Fraction Class

Let's write some more functions...

```cpp
// get the decimal value
Fraction fracA(1,2);
double f = fracA.decimal();
cout << f << endl;

output: 0.5
```

```cpp
// purpose: To return a decimal value of our fraction
// arguments: None
// return value: the decimal value of this fraction
double Fraction::decimal()
{
}
```
The Fraction Class

Let's write some more functions...

```cpp
// get the decimal value
Fraction fracA(1, 2);
double f = fracA.decimal();
cout << f << endl;
```

Output:
0.5

```cpp
// purpose: To return a decimal
// value of our fraction
// arguments: None
// return value: the decimal
// value of this fraction
double Fraction::decimal()
{
    // returns the decimal
    // value of our fraction
    return (double)num / denom;
}
```
void Fraction::reduce() {
    // reduce the fraction to lowest terms
    // find the greatest common divisor
    int frac_gcd = gcd(num, denom);
    // reduce by dividing num and denom
    // by the gcd
    num = num / frac_gcd;
    denom = denom / frac_gcd;
}
int Fraction::gcd(int u, int v) {
    if (v != 0) {
        return gcd(v, u % v);
    }
    else {
        return u;
    }
}
The Fraction Class: overloading `<<`

Yes, this syntax is a bit strange.

Basically, we are telling the compiler how to `cout` our `Fraction`. You can do something very similar for Boggle.

```cpp
// purpose: To overload the `<<` operator
// for use with `cout`
// arguments: a reference to an `outstream` and the
// fraction we are using
// return value: a reference to the `outstream`
ostream& operator<<(ostream& out, Fraction &frac) {
    out << frac.num << "/" << frac.denom;
    return out;
}
```
• **Advanced Reading**
  - Overloading the assignment operator:
  - Constructors and Destructors:

• **References:**
  - [https://www.tutorialspoint.com/cplusplus/cpp_classes_objects.htm](https://www.tutorialspoint.com/cplusplus/cpp_classes_objects.htm)
The Copy Constructor

Vector<int> a;
a.add(0);
a.add(1);
a.add(1);
a.add(2);
Vector<int> b = a;

or
Vector<int> b(a); // b gets constructed
// with the same elements as b

This doesn't work automatically!

The assignment overload:
Vector<int> a;
a.add(0);
a.add(1);
Vector<int> b;
b.add(8);
b = a; // a gets copied into b