CSCI 2270

Data Structures & Algorithms

Gabe Johnson

Lecture 24 March 13, 2013

Classes and Objects in C++

Upcoming Homework Assignment

HW #7 **Due: Friday, Mar 15**

Huffman Encoding

Get to encoding those characters! You have a driver this week, since the assignment is pretty tricky. Many of the functions are implemented for you already.

Questions?

Lecture Goals

1. Object Orientation
2. Classes and Objects in C++

LinkedList OO example

There's an object-oriented implementation of the LinkedList assignment sitting in the course github:

cs2270 / code / objects / LinkedList.cpp

cs2270 / code / objects / LinkedList.hpp

Object Orientation

Object Orientation is a way to bundle together data and behaviors into *objects*, rather than having data and behavior separate, as is the case with procedural programming.

A *class* is an object's type. We've used them already. We use the vector template class, and the string class, which makes it easier to work with text.

Classes vs Objects

A *class* is like a blueprint for a house. It is like a template that specifies how to build the house.

An *object* is like the house itself.

A builder can use a single blueprint to specify how to build a house. Or six. Or six hundred. There's one blueprint, but lots of houses.

Another Analogy

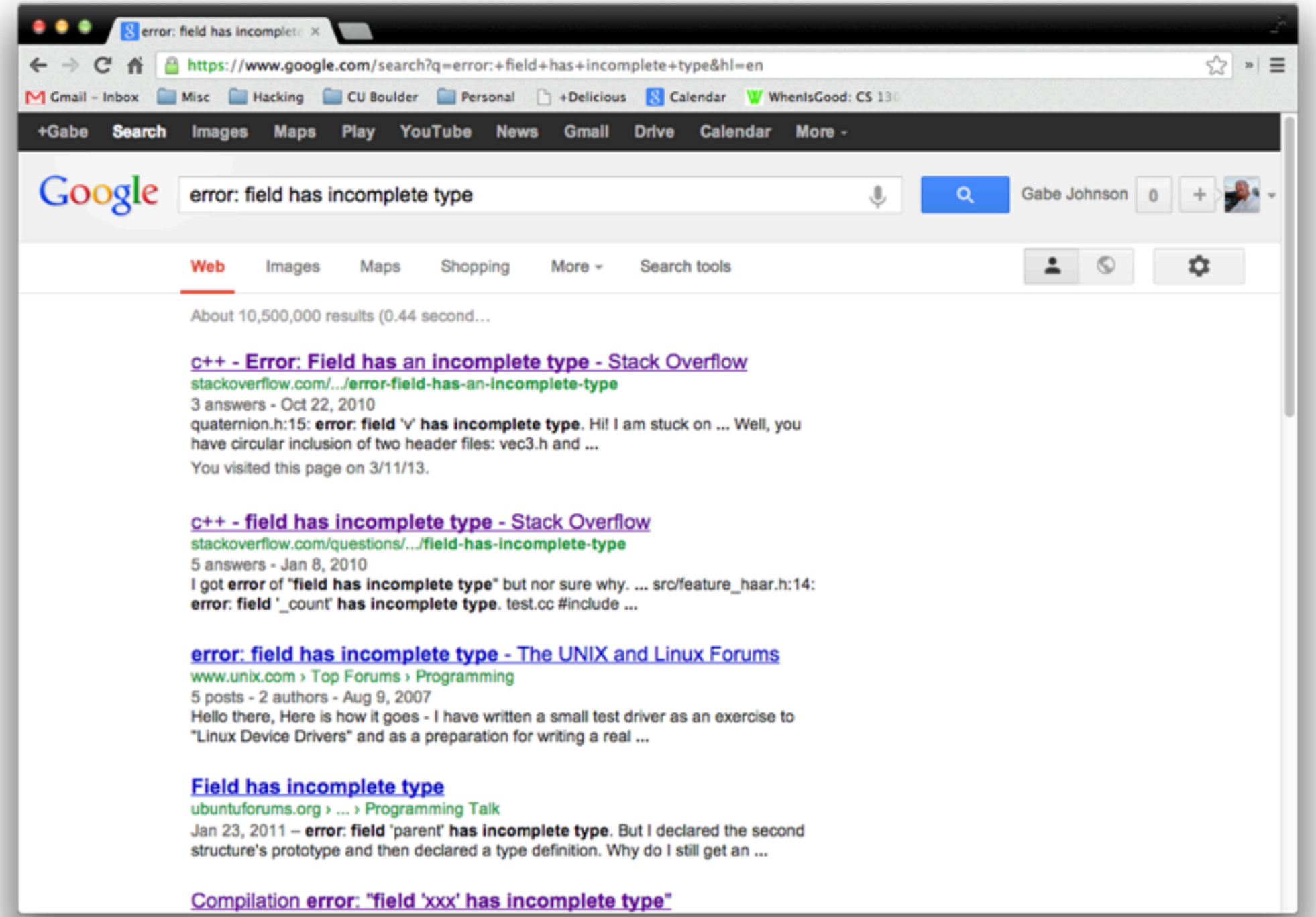
We have a machine that produces sprockets. We tell it how many teeth, what the inner and outer radii are, and several other things, and it spits out custom sprockets.

Use a Class to make an Object

To continue with the blueprint / house example:

```
class House {
  // declarations here, implementation
  // in some other file.
};

// this is how you'd create instances:
House my_house;
House your_house;
```

Dealing w/ error messages



Back in my day, we didn't have Google to answer our questions about asinine error messages. We had to slam our heads against the monitor until either we blacked out from the trauma, or we figured it out.

And we liked it!

But you should totally use Google. Pro engineers do this like ten times a day.

Header File

Name it foo.h or foo.hpp. Put class definition with function declarations.

```
class LinkedList {
  // ------------------------------------ private members
private:
  LinkedListNode* root;
  // ------------------------------------- public members
public:
  LinkedList();
  std::string report();
  void append(int data);
  void insert(int data, int offset);
  void remove(int offset);
  int size();
  bool contains(int data);
  friend std::ostream &operator << (std::ostream& out, LinkedList
list);
}; // you have to have a semicolon at the end.
```

Implementation File

The CPP file implements those functions. Note that identifiers are prefixed with `LinkedList::`

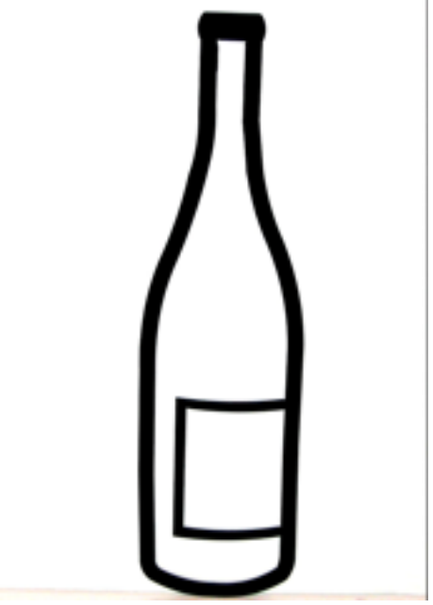
```
LinkedList::LinkedList() {
  cout << "  >> Creating LinkedList instance." << endl;
  root = NULL;
}

string LinkedList::report() {
  stringstream ss;
  if (root != NULL) {
    ss << "[ ";
    LinkedListNode* cursor = root;
    while (cursor != NULL) {
      ss << cursor->value << ", ";
      cursor = cursor->next;
    }
    ss << "]";
  }
  return ss.str();
}
```

OO Thinking

Object orientation matches how many people think about modeling real-world things in software. If this doesn't jive with you now, give it a shot anyway and see how you like it.

C++ doesn't exactly make this easy, but there are lots of other OO languages that don't have as much vestigial 1972 in there. *(that's when C was released)*

Members: Vars & Functions



Consider a simple ‘Bottle’ class. Instances contain various kinds of liquid, they have maximum capacities, and some current content. We can also put in or take some out.

Data (Variables)

* Liquid Type
* Max Liquid
* Current Liquid

Behaviors (Functions)

* Pour
* Fill

Bottle Definition

Data (Variables)

* Max Liquid
* Current Liquid
* Liquid Type

Behaviors (Functions)

* Pour
* Fill

```
class Bottle {
  float max_liquid;
  float current_liquid;
  string liquid_type;
  void pour(float amount);
  void fill(float amount);
};
```

Encapsulation / Hiding

We have this great Bottle class, but there's nothing preventing a malicious user or CS 2270 student from setting the current_liquid value to something crazy, like -10 or max_liquid + 10.

Classes give us the ability to *encapsulate* the data so we protect it behind functions that are intended to **prevent abuse** and/or **bugs**.

Encapsulation

```
class Bottle {
private:
  float max_liquid;
  float current_liquid;
public:
  string liquid_type;
  void pour(float amount);
  void fill(float amount);
  float get_max_liquid();
  void set_max_liquid();
  float get_current_liquid();
};
```

Hide the liquid values behind **accessor** and **mutator** functions. Our implementation of get/set max liquid would prevent it from going negative. We only give *read access* to `current_liquid` and *read/write* access to `max_liquid`.

Encapsulation

```
class Bottle {
private:
  float max_liquid;
  float current_liquid;
public:
  string liquid_type;
  void pour(float amount);
  void fill(float amount);
  float get_max_liquid();
  void set_max_liquid();
  float get_current_liquid();
};
```

If we tried to access max_liquid or current_liquid from outside one of the Bottle functions, the compiler would yell at us. This is for your mental well-being. Trust me.

Encapsulation cuts down on information overload *and* lets class implementers do their own runtime QA.

Creating Objects

We create objects by using a *Constructor*. This is a function that has an implied return type and is named the same as the class.

You don’t need to define a constructor, and C++ gives you a *default constructor* that initializes variables for you. This can can be dangerous (giving you unwanted defaults and such) so I recommend making your own constructor(s).

Custom Constructors

A custom constructor lets you give custom values, possibly based on parameters. If we want to be able to create Bottle objects with an initial current and max value, we could do it there:

```
Bottle::Bottle(float initial_max, float
initial_current) {
   max_liquid = initial_max;
   current_liquid = initial_current;
}
```

Using Constructors

To use a the default constructor (or a custom constructor that takes no parameters), just declare the variable type:

```
Bottle default_bottle;
```

To use a custom constructor with params:

```
Bottle fancy_bottle(22.8, 17.5);
```

‘new’ operator

We can use the ‘new’ operator to create objects on the heap, rather than the stack:

```
Bottle* ptr = new Bottle(22.3, 10.0);
```

This lets our objects persist after the function returns, and we can pass the pointer around rather than copying the thing by value. Access members with the arrow operator:

```
ptr->pour();
```

Class def caveat

You can't refer to objects of a class inside of that class:

```
class Node {
  int value;
  Node next; // bzzt
};
```

You can't refer to objects of a class inside of that class:

```
class Node {
  int value;
  Node* next; // ok
};
```

Destroying Objects

Objects have a ‘destructor’ function that cleans up when it is deleted or when it is popped off the stack. This is just the class name with a tilde character in front. Like this:

```
class Bottle {
  // ...other stuff...
  ~Bottle();
};
```

Private and Protected

The ‘private’ classification is a useful tool to keep your users (possibly this is yourself) happy. They don’t care how your class is implemented, so you should keep those dirty details *private*.

A ‘protected’ classification is used for subclasses. If I make a WineBottle class that is a subclass of Bottle, it has access to the public and protected variables but *not* the private ones. We’d need to upgrade those variables to be protected if we did this.

Public Members

Your class's public members define the interface that the rest of the world will use.

It is important that these be clear, and the parameters be named sensibly and be arranged in a consistent order. This is because documentation sucks. Yes it can be helpful, but once you know how something works, you just need a *hint*, and a nicely designed **public API** will do this for you without lengthy documentation.

PBR for Objects

We can pass objects by *reference*, just like we do with other variables. This is actually advantageous, because when we pass by *value* the input has to be copied into newly allocated memory, and this can be pretty involved for some objects. Same syntax:

```
public do_something(Bottle& bottle) {
   bottle.pour(); // pour the original bottle, not a copy.
}
```

PBR for Objects

```
public do_something(const Bottle& bottle) {
  // can't modify bottle!
   bottle.pour(); // error!
   cout << "current value of liquid: " <<
bottle.get_current_liquid() << endl; // ok.
}
```

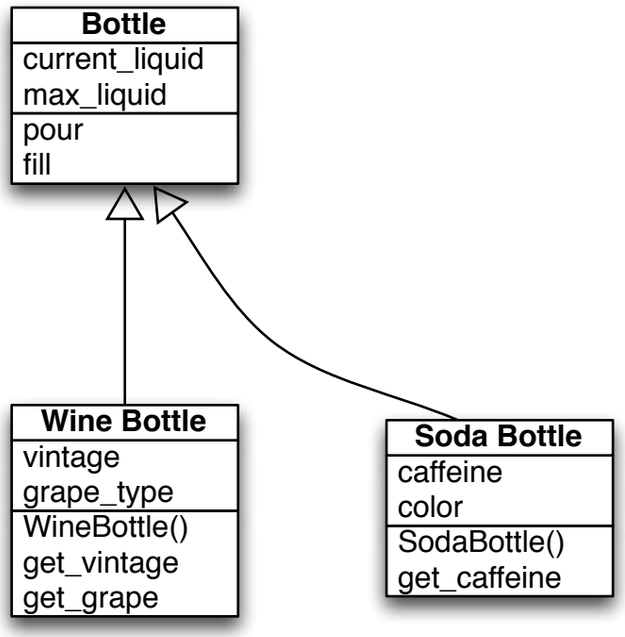
Class Hierarchies

Can subclass and inherit members that aren't private.

Can also have 'protected' members. These are designed to be used with subclassing.

C++ has multiple inheritance. Don't use it.

Why Subclass?



Some software engineers *love* these diagrams. They help you visualize the relationship among classes. Here you can see there are three classes, and Wine Bottle and Soda Bottle are subclasses of plain old Bottle. A subclass can inherit the (non-private) variables and functions of the superclass. We can also treat the subclass as its superclass.

Objects Are Types

Just like structs can contain other structs, objects can contain other objects.

```
class Rectangle {
  Point top_left;
  Dimension size;
  float angle;
};
```

```
class Point {
  float x;
  float y;
};

class Dimension {
  int w;
  int h;
};
```

Operator Overloading

Since objects are types, just like every other variable, we can apply operations to them.

Operators can be member functions of a class, or they can be defined separately. Sadly, we can't do this for operators like << that take the stream as the first argument, since the object would be the first (implied) argument.

Op. Overload Example

```
// LinkedList.hpp
class LinkedList {
  friend std::ostream &operator <<
         (std::ostream& out, LinkedList list);
};

// LinkedList.cpp
ostream &operator << (ostream& out, LinkedList list) {
  if (list.root != NULL) {
    out << "[ ";
    LinkedListNode* cursor = list.root;
    while (cursor != NULL) {
      out << cursor->value << ", ";
      cursor = cursor->next;
    }
    out << "]";
  }
  return out;
}
```

Then we can use it with cout:
LinkedList george;
cout << “list is: “ << george << endl;