Eric Roberts
CS 106B

Handout #14
January 18, 2013

Designing Classes

Classes and Objects

Dawson Zhou
CS 106B
January 18, 2013

Structures

- All modern higher-level languages offer some facility for representing ***structures***, which are compound values in which the individual components are specified by name.
- Given that C++ is a superset of the older C language, it is still possible to define classic C structures, which are defined using the following syntactic form:

```
struct typename {
   declarations of fields
};
```

- This definition creates a *type,* not a *variable.* If you want to declare variables of the structure type, you use the type name just as you would with any other declaration.

A Classic Example of a Structure

- One of the simplest examples of a data structure is a ***point***, which is composed of an *x* and a *y* component. For most graphical devices, coordinates are measured in pixels on the screen, which means that these components can be integers, since it is impossible to illuminate a fraction of a pixel.

```
struct Point {
   int x;
   int y;
};
```

- This definition allows you to declare a `Point` variable like this:

```
Point pt;
```

- Given the variable `pt`, you can select the individual fields using the dot operator (`.`), as in `pt.x` and `pt.y`.

Classes and Objects

- Object-oriented languages are characterized by representing most data structures as ***objects*** that encapsulate representation and behavior in a single entity. In C, structures define the representation of a compound value, while functions define behavior. In C++, these two ideas are integrated.
- As in Java, the C++ object model is based on the idea of a ***class***, which is a template describing all objects of a particular type. The class definition specifies the representation of the object by naming its ***fields*** and the behavior of the object by providing a set of ***methods***.
- New objects are created as ***instances*** of a particular class.

A Note on Inheritance

- As you know if you've studied other object-oriented languages such as Java, classes form hierarchies in which subclasses ***inherit*** the behavior and representation of their superclass.
- Inheritance also applies in C++, although the model is more complex, primarily because C++ allows classes to inherit from more than one superclass. This property is called ***multiple inheritance***.
- Partly because of this additional complexity and partly because C++ makes inheritance harder to use, CS 106B postpones the discussion of inheritance until later in the quarter, focusing instead on the use of classes for encapsulation.

The Format of a Class Definition

- In C++, the definition of a class typically looks like this:

```
class typename {
  public:
    prototypes of public methods
  private:
    declarations of private instance variables
    prototypes of private methods
};
```

- The entries in a class definition are divided into two categories:
  - A public section available to clients of the class
  - A private section restricted to the implementation

Implementing Methods

- A class definition usually appears as a `.h` file that defines the ***interface*** for that class. The class definition does not specify the implementation of the methods exported by the class; only the prototypes appear.
- Before you can compile and execute a program that contains class definitions, you must provide the implementation for each of its methods. Although methods can be implemented within the class definition, it is stylistically preferable to define a separate `.cpp` file that hides those details.
- Method definitions are written in exactly the same form as traditional function definitions. The only difference is that you write the name of the class before the name of the method, separated by a double colon. For example, if the class `MyClass` exports a `toString` method, you would code the implementation using the method name `MyClass::toString`.

Overloading Operators

- One of the most powerful features of C++ is the ability to extend the existing operators so that they apply to new types. Each operator is associated with a name that usually consists of the keyword `operator` followed by the operator symbol.
- When you define operators for a class, you can write them either as methods or as free functions. Each styles has its own advantages and disadvantages. For more details, you should look at the `Rational` class in Chapter 6.
- My favorite operator to overload is the `<<` operator, which makes it possible to print values of a type on an output stream. The prototype for the overloaded `<<` operator is

```
ostream & operator<<(ostream & os, type var)
```

Constructors

- In addition to method prototypes, class definitions typically include one or more ***constructors***, which are used to initialize an object.
- The prototype for a constructor has no return type and always has the same name as the class. It may or may not take arguments, and a single class can have multiple constructors as long as the constructors have different parameter sequences.
- The constructor that takes no arguments is called the ***default constructor***. If you don't define any constructors, C++ will automatically generate a default constructor with an empty body.
- The constructor for a class is *always* called when you create an instance of that class, even if you simply declare a variable.

The `point.h` Interface

```
/*
 * File: point.h
 * -------------
 * This interface exports the Point class, which represents a point
 * on a two-dimensional integer grid.
 */

#ifndef _point_h
#define _point_h

#include <string>

class Point {

public:
```

The `point.h` Interface

```
/*
 * Constructor: Point
 * Usage: Point origin;
 *        Point pt(xc, yc);
 * -----------------------
 * Creates a Point object.  The default constructor sets the
 * coordinates to 0; the second form sets the coordinates to
 * xc and yc.
 */

   Point();
   Point(int xc, int yc);
```

The `point.h` Interface

```
/*
 * Methods: getX, getY
 * Usage: int x = pt.getX();
 *        int y = pt.getY();
 * -----------------------
 * These methods returns the x and y coordinates of the point.
 */

   int getX();
   int getY();

/*
 * Method: toString
 * Usage: string str = pt.toString();
 * ----------------------------------
 * Returns a string representation of the Point in the form "(x,y)".
 */

   std::string toString();
```

The `point.h` Interface

```
private:

   int x;                    /* The x-coordinate */
   int y;                    /* The y-coordinate */

};

/*
 * Operator: <<
 * Usage: cout << pt;
 * ------------------
 * Overloads the << operator so that it is able to display Point
 * values.
 */

std::ostream & operator<<(std::ostream & os, Point pt);

#endif
```

The `point.cpp` Implementation

```
/*
 * File: point.cpp
 * ---------------
 * This file implements the point.h interface.
 */

#include <string>
#include "point.h"
#include "strlib.h"
using namespace std;

/* Constructors */

Point::Point() {
   x = 0;
   y = 0;
}

Point::Point(int xc, int yc) {
   x = xc;
   y = yc;
}
```

The `point.cpp` Implementation

```
/* Getters */

int Point::getX() {
   return x;
}

int Point::getY() {
   return y;
}

/* The toString method and the << operator */

string Point::toString() {
   return "(" + integerToString(x) + "," + integerToString(y) + ")";
}

ostream & operator<<(ostream & os, Point pt) {
   return os << pt.toString();
}
```

Methods in the `TokenScanner` Class

`scanner.setInput(str)` *or* **`scanner.setInput(infile)`** Sets the input for this scanner to the specified string or input stream.
`scanner.hasMoreTokens()` Returns **`true`** if more tokens exist, and **`false`** at the end of the token stream.
`scanner.nextToken()` Returns the next token from the token stream, and `""` at the end.
`scanner.saveToken(token)` Saves **`token`** so that it will be read again on the next call to **`nextToken`**.
`scanner.ignoreWhitespace()` Tells the scanner to ignore whitespace characters.
`scanner.scanNumbers()` Tells the scanner to treat numbers as single tokens.
`scanner.scanStrings()` Tells the scanner to treat quoted strings as single tokens.

Exercise: Implement a Symbol Table

- On Wednesday, you learned how to implement a symbol table using **`Map<string,double>`**. Suppose that you didn't have the **`Map`** class. Without trying to be at all efficient, how could you implement a **`SymbolTable`** class that supports the operations **`get`** and **`put`**?