CSI 1430
Week 11 Resource

Colin Burdine

3/28/21

Major Topics:

1. Class Functions
2. Streams

Keywords: *objects, classes, class functions, streams*

Reminder: Group tutoring sessions for this course will be held online every Monday from 7:00pm to 8:00pm CDT. For more information on how to sign up for these sessions, go to: https://www.baylor.edu/tutoring.

1 Notes on Major Topics:

1.1 Class Functions

Last week we discussed how Classes could use the special functions called *constructors* to restrict the ways in which instances of classes (i.e. objects) can be initialized. This week we will wrap up our discussion of classes by introducing some of the common functions and operators that can be used with objects.

In some situations, we might want to have the convenience of extending certain operations to objects themselves. For example, if we define a `point_t` class, we might want to be able to add, subtract, and test the equality of any two points. This act of extending a function such that it has the same name, but different parameters is called *overloading*, and is common in object-oriented programming. Using these overloaded operators in C++, we can effectively perform the same arithmetic operations that we can on primitive types like

`int` and `double`:

```
// initialize points with constructor:
point_t a = point_t(0.0, 0.0);
point_t b = point_t(1.2, 3.4);

// overload the '+' operator:
point_t c = a + b;

// overload the '==' operator:
if(b == c){
    // this should be printed:
    cout << "a + b equals c" << endl;
} else {
    cout << "a + b does not equal c" << endl;
}
```

As it turns out, C++ provides mechanisms by which we can do this. Inside the body of our point class, we can define the functions `point_t operator+(const point_t& p)` and provide the prototype for `friend point_t operator==(const point_t& p)`. Note the use of the `operator` keyword followed by the symbol corresponding to the operator.

We observe that the declaration of `operator==` is special, because it is declared as a `friend` of the `point_t` class. By C++ language specification, the `operator==` function cannot be defined inside of a class; it must be defined externally. This is due to C++ allowing for something you will learn in the coming semesters, called *run-time polymorphism*. By declating `operator==` a friend of the `point_t` we grant is special privileges to access the private members of `point_t`. The implementation of these functions looks like:

```
// point_t.h

class point_t {
private:
    double x, y;

public:
    ...

    // overloaded '+' operator:
    point_t operator+(const point_t& p);

    // this signifies the external function "operator==" for
    //point_t types has access to private members of point_t:
    friend bool operator==(const point_t& p1, const point_t& p2);
```

```
}

// this is the actual prototype of the overloaded '==' operator:
bool operator==(const point_t& p1, const point_t& p2);
```

```
// point_t.cpp

point_t point_t::operator+(const point_t& p){

    // return a new point that is the sum of the
    // corresponding coordinates:
    return new point_t(x+p.x,y+p.y);
}

bool operator==(const point_t& p1, const point_t& p2){

    // return true only if the coordinates are the same:
    return ((p1.x == p2.x) && (p1.y == p2.y));
}
```

2 Streams

Input and Output "I/O" is a critical function of any programming language. Thankfully, C++ provides a useful set of classes called *streams* that facilitate I/O. The two stream objects you may be used to seeing are `cin` and `cout`. These are in fact objects of the `iostream` class. C++ also allows for file I/O through the use of the `ifstream` class. Finally, C++ also allows for the use of input and output operations to be performed on strings themselves through the `sstream` ("string stream") class. I will not be going into details concerning these classes, since you have probably seen each of these at some point int he semester. For your reference, I have provided a table to summarize some of this information:

I/O type	Include directive	Usage	Link
Console (standard in/out)	`#include <iostream>`	Input to/from the console	Docs
File I/O	`#include <fstream>`	Input to/from files	Docs
String I/O	`#include <sstream>`	Input to/from strings	Docs
I/O Formatting	`#include <iomanip>`	Formatted I/O to/from streams	Docs

3 Programming Example Problem

To illustrate how we can use classes to solve problems, consider the following example programming problem:

Create a class `angle_t` to represent a (decimal-values) angle between zero and 360 degrees. Include a default constructor that initializes an angle at 0 and a parameterized constructor. In this class, angles should always be truncated to a value between 0 and 360 degrees. In the class, overload the following operators:

```
operator+ // the "addition" operator (adds two angles)
operator- // the "subtraction" operator (subtracts two angles_
operator= // the assignment operator
operator< // the "less than" operator
operator== // the "equal to" operator
```

If you would like, you may declare the entire class inline (i.e. define all functions in the header).

My solution to this problem is on the next page. You can look at my answer if you are stuck, but I would strongly encourage you to sketch out a design first, and then proceed to write your solution in C++. This exercise is intended to check your understanding of the concepts from this week and last week.

```cpp
#ifndef ANGLE_T_H
#define ANGLE_T_H
#include <cmath> // for fmod:

class angle_t {
private:
    double theta;
public:
    // default constructor:
    angle_t(){
        theta = 0.0;
    }
    // parameterized constructor:
    angle_t(double t){
        // ensure angle is on [0,360):
        theta = fmod(t, 360.0);
        if(theta < 0.0){
            theta += 360.0;
        }
    }

    // +/- operators:
    angle_t operator+(const angle_t& a){
        return angle_t(theta + a.theta);
    }

    angle_t operator-(const angle_t& a){
        return angle_t(theta - a.theta);
    }

    // assignment operator:
    const angle_t& operator=(const angle_t& a){
        theta = a.theta;
        return a;
    }

    // comparison operators:
    friend bool operator<(const angle_t& a1, const angle_t& a2);
    friend bool operator==(const angle_t& a1, const angle_t& a2);
};

// comparison operator prototypes:
bool operator<(const angle_t& a1, const angle_t& a2){
    return (a1.theta < a2.theta);
}
bool operator==(const angle_t& a1, const angle_t& a2){
    return (a1.theta == a2.theta);
}
#endif /* ANGLE_T_H */
```