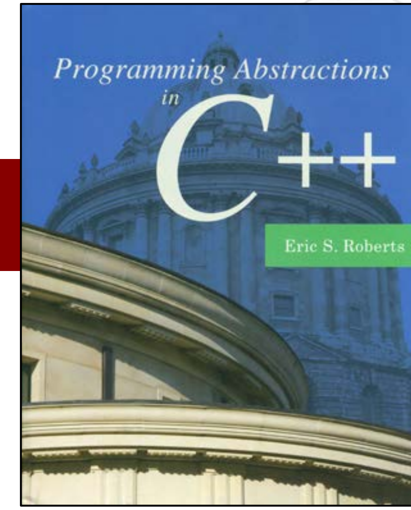
CHAPTER 6

Designing Classes



You don't understand. I coulda had class. . . .

—Marlon Brando's character in
On the Waterfront, 1954

Representing Points

- One of the simplest examples of a data structure is a ***point***, which is composed of an *x* and a *y* component.
- C++ offers more than one model for representing a point value. The older style, which C++ inherits from C, is to defined the **`Point`** type as a ***structure***. The more modern approach is to define **`Point`** as a ***class***. The next several slides explore each of these models.

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. If you define **`Point`** as a structure, the definition looks like this:

```
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.

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

THE UNIVERSITY OF ALABAMA
DEPARTMENT OF COMPUTER SCIENCE

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, which are outlined in the section on 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();
}
```

Rational Numbers

- As a more elaborate example of class definition, section 6.3 defines a class called `Rational` that represents *rational numbers*, which are simply the quotient of two integers.
- Rational numbers can be useful in cases in which you need exact calculation with fractions. Even if you use a `double`, the floating-point number 0.1 is represented internally as an approximation. The rational number 1 / 10 is exact.
- Rational numbers support the standard arithmetic operations:

Addition:

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$$

Multiplication:

$$\frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd}$$

Subtraction:

$$\frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd}$$

Division:

$$\frac{a}{b} \div \frac{c}{d} = \frac{ad}{bc}$$

Implementing the `Rational` Class

- The next several slides show the code for the `rational.h` interface and the `rational.cpp` implementation.
- As you read through the code, the following features are worth special attention:
  - *The constructors for the class are overloaded.* Calling the constructor with no argument creates a `Rational` initialized to 0, calling it with one argument creates a `Rational` equal to that integer, and calling it with two arguments creates a fraction.
  - *The constructor makes sure that the number is reduced to lowest terms.* Moreover, since these values never change once a new `Rational` is created, this property will remain in force.
  - *The class overloads the standard arithmetic operations to allow the use of conventional mathematical notation.* Thus, if you want to add the rational numbers `r1` and `r2`, you write

```
r1 + r2
```

The `rational.h` Interface

```
/*
 * File: rational.h
 * ----------------
 * This interface exports a class representing rational numbers.
 */

#ifndef _rational_h
#define _rational_h

#include <string>
#include <iostream>

/*
 * Class: Rational
 * ---------------
 * The Rational class is used to represent rational numbers, which
 * are defined to be the quotient of two integers.
 */

class Rational {
```

The `rational.h` Interface

```
public:

/*
 * Constructor: Rational
 * Usage: Rational zero;
 *        Rational num(n);
 *        Rational r(x, y);
 * ------------------------
 * Creates a Rational object.  The default constructor creates the
 * rational number 0.  The single-argument form creates a rational
 * equal to the specified integer, and the two-argument form
 * creates a rational number corresponding to the fraction x/y.
 */

   Rational();
   Rational(int n);
   Rational(int x, int y);
```

The `rational.h` Interface

```
/*
 * Operators: +, -, *, /
 * ---------------------
 * Define the arithmetic operators.
 */

   Rational operator+(Rational r2);
   Rational operator-(Rational r2);
   Rational operator*(Rational r2);
   Rational operator/(Rational r2);

/*
 * Method: toString()
 * Usage: string str = r.toString();
 * ---------------------------------
 * Returns the string representation of this rational number.
 */

   std::string toString();
```

The `rational.h` Interface

```
private:

/* Instance variables */

   int num;    /* The numerator of this Rational object   */
   int den;    /* The denominator of this Rational object */

};

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

std::ostream & operator<<(std::ostream & os, Rational rat);

#endif
```

The `rational.cpp` Implementation

```
/*
 * File: rational.cpp
 * ------------------
 * This file implements the Rational class.
 */

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

/* Function prototypes */

int gcd(int x, int y);

/* Constructors */
```

The `rational.cpp` Implementation

```
Rational::Rational() {
   num = 0;
   den = 1;
}

Rational::Rational(int n) {
   num = n;
   den = 1;
}

Rational::Rational(int x, int y) {
   if (x == 0) {
      num = 0;
      den = 1;
   } else {
      int g = gcd(abs(x), abs(y));
      num = x / g;
      den = abs(y) / g;
      if (y < 0) num = -num;
   }
}
```

The `rational.cpp` Implementation

```
/* Implementation of the arithmetic operators */

Rational Rational::operator+(Rational r2) {
   return Rational(num * r2.den + r2.num * den, den * r2.den);
}

Rational Rational::operator-(Rational r2) {
   return Rational(num * r2.den - r2.num * den, den * r2.den);
}

Rational Rational::operator*(Rational r2) {
   return Rational(num * r2.num, den * r2.den);
}

Rational Rational::operator/(Rational r2) {
   return Rational(num * r2.den, den * r2.num);
}
```

The `rational.cpp` Implementation

```
string Rational::toString() {
   if (den == 1) {
      return integerToString(num);
   } else {
      return integerToString(num) + "/" + integerToString(den);
   }
}

int gcd(int x, int y) {
   int r = x % y;
   while (r != 0) {
      x = y;
      y = r;
      r = x % y;
   }
   return y;
}

ostream & operator<<(ostream & os, Rational rat) {
   os << rat.toString();
   return os;
}
```

The End