CHAPTER 6
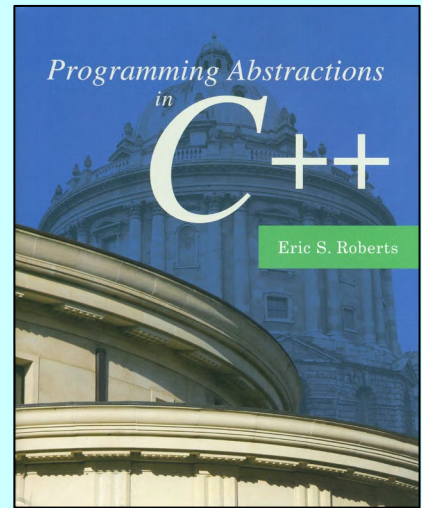
Designing Classes

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

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

Data Types in C++

- Primitive types include but not limited to:

`int`	This type is used to represent integers, which are whole numbers such as 17 or −53.
`double`	This type is used to represent numbers that include a decimal fraction, such as 3.14159265.
`bool`	This type represents a logical value (`true` or `false`).
`char`	This type represents a single ASCII character.

- Collections
- Compound data types:
  - Enumerated types
  - Structures
  - Classes (e.g., C++ strings and streams)

Enumerated types

- An enumerated type can be roughly considered as a subset of `int` with special names for each value in the subset:

  ```
  enum Direction { NORTH, EAST, SOUTH, WEST };
  ```

- When the C++ compiler encounters this definition, it assigns values to the constant names by numbering them consecutively starting with 0. Thus, NORTH is assigned the value 0, EAST is 1, SOUTH is 2, and WEST is 3.

- You can assign values by yourself:

  ```
  enum Coin {
      PENNY = 1,
      NICKEL = 5,
      DIME = 10,
      QUARTER = 25,
      HALF_DOLLAR = 50,
      DOLLAR = 100
  };
  ```
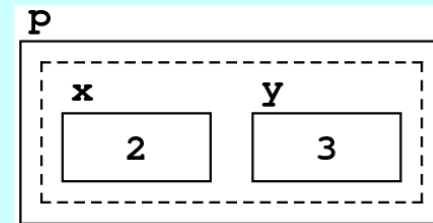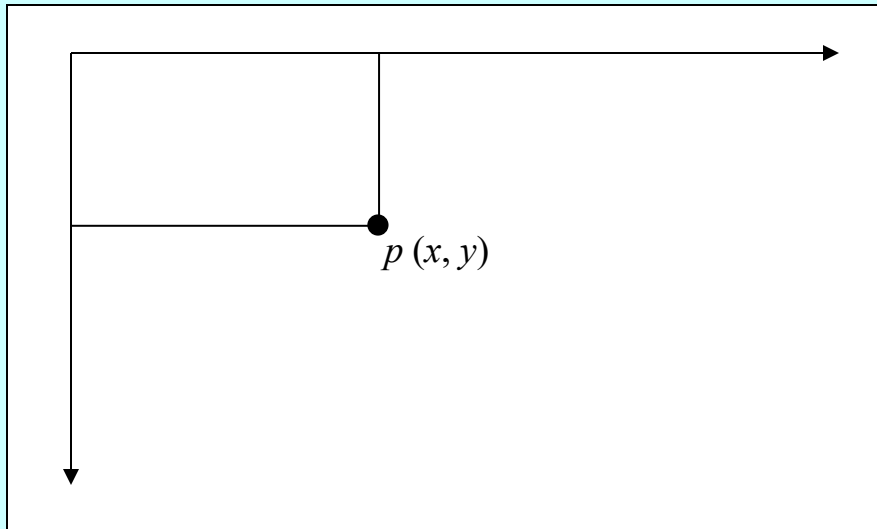
Structures

- 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.

Representing Points

- One of the simplest examples of a data structure is a *point*, which is composed of an *x* coordinate and a *y* coordinate 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.

Representing Points Using a Structure

- For most graphical devices, coordinates are measured in pixels on the screen, which means that these components *x* and *y* should 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* or *members* using the dot operator (`.`), as in `pt.x` and `pt.y`.

Representing Points Using a Class

- Although structure types are part of the history of C++ and the languages that came before it, they have largely been supplanted by classes, which offer greater power and flexibility. The `Point` structure from the preceding section is identical to the following class definition:

```
class Point {
public:
    int x;
    int y;
};
```

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

```
Point pt;
```

- As the x and y fields are included as part of the `public` section, which is visible to clients, you can select the public *fields* using the dot operator (`.`), as in `pt.x` and `pt.y`.

Classes and Objects

- In C, structures define the representation of a compound value, while functions define operations on these data. In C++, these two ideas are integrated.

- Object-oriented languages are characterized by representing most data structures as *objects* that encapsulate both the representation and the behavior of the objects (i.e., operations on the data) in a single entity.

- As in Python, 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* or *instance variables* (e.g., compound property/attribute values) and the behavior of the object by providing a set of *methods*.

- New objects are created as *instances* of a particular class, and the creation of new instances is called *instantiation*.

Class and Objects
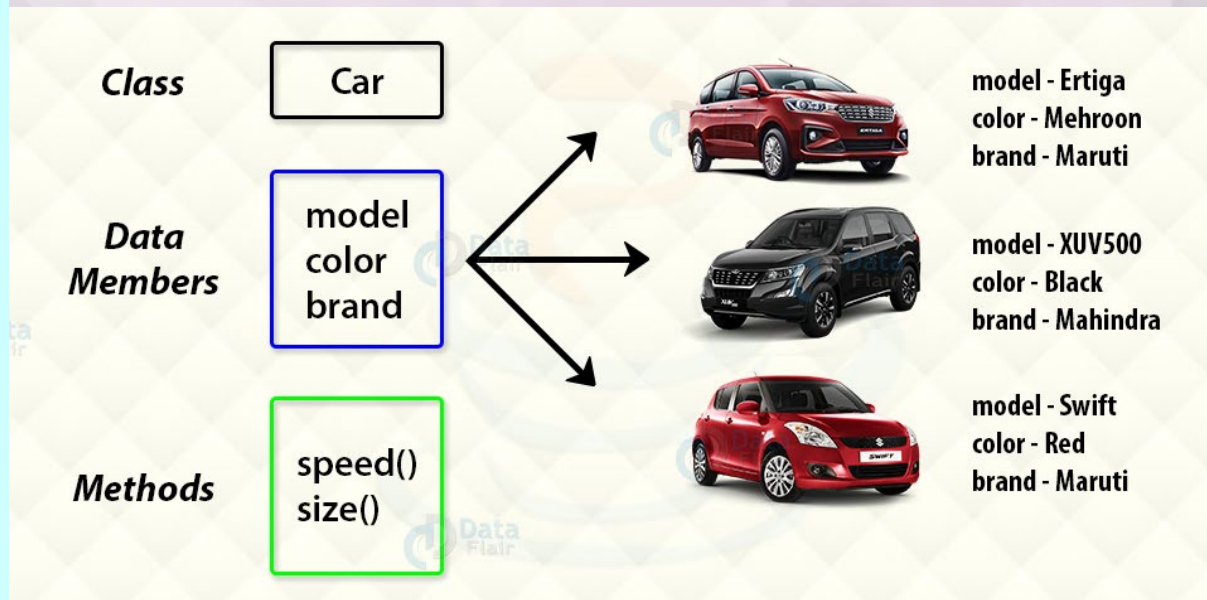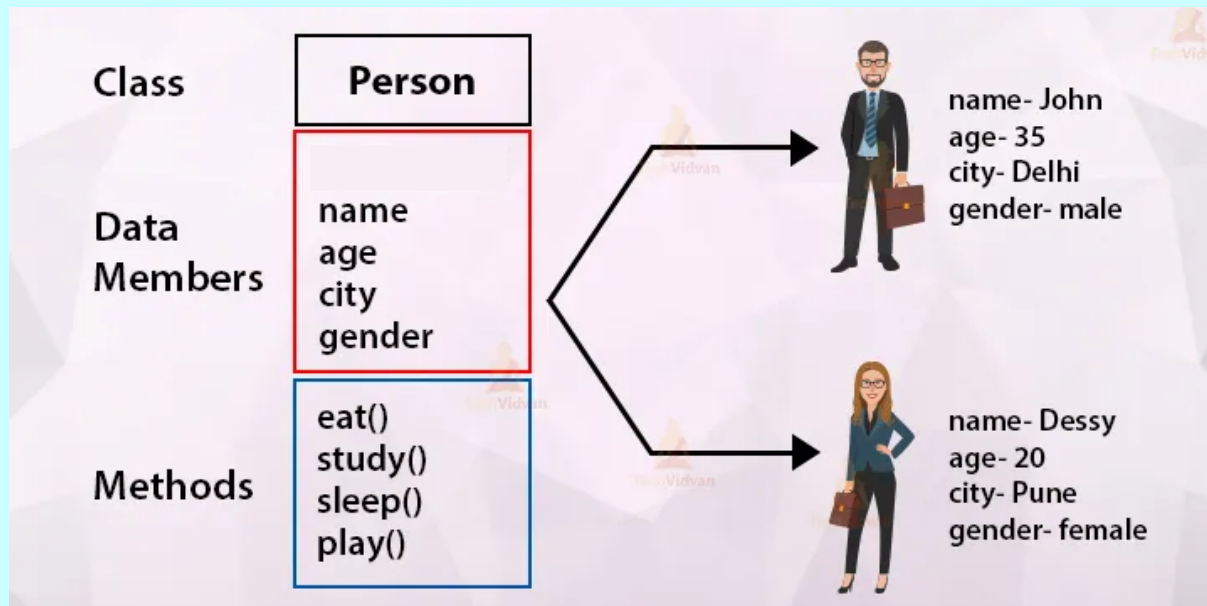
The Imperative Programming Paradigm

- **Imperative** programming paradigm: an explicit sequence of statements that change a program's state, specifying how to achieve the result.

  - **Structured**: Programs have clean, `goto`-free, nested control structures.

  - **Procedural**: Imperative programming with procedures operating on data.

  - **Object-Oriented**: Objects have/combine states/data and behavior/methods; Computation is effected by sending messages to objects (receivers).

    - **Class-based**: Objects get their states and behavior based on membership in a class.

    - **Prototype-based**: Objects get their behavior from a prototype object.

Representing Points Using a Class

- Declaring public instance variables, however, is discouraged in modern object-oriented programming. Today, the common practice is to make all instance variables **private**, which means that clients have no direct access to the internal variables.

- Clients instead use methods exported by the class to obtain access to any information the class contains. Keeping implementation details away from the client is likely to foster *simplicity*, *flexibility*, and *security*, as described in the previous lectures.

```
class Point {
public:
    prototypes of public methods to access x and y
private:
    int x;
    int y;
};
```

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. In some rare cases, however, some simple implementations can be included in the `.h` file, therefore a `.cpp` file is not always required.

- 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. E.g., the `Point` class exports a `toString` method, you would code the implementation using the method name `Point::toString`. ($\approx$ `__str__` in Python)

Getters and Setters

- In computer science, methods that retrieve the values of instance variables are formally called *accessors*, but are more often known as *getters*. By convention, the name of a getter method begins with the prefix `get` followed by the name of the field after capitalizing the first letter in its name. The getters for the `Point` class can therefore be defined as `getX` and `getY`.

- Methods that set the values of specific instance variables are called *mutators* or, more informally, *setters*. If the `Point` class were to export a `setX` and a `setY` method that allowed the client to change the values of these fields, you could easily replace any application that previously used the old structure type with a version that relies entirely on the new `Point` class.

- Remember the *getters* and *setters* in the collection classes?

Setters or not?

- It is, however, counter-intuitive to add setter methods to a class after deciding to make its instance variables private, because part of the reason for making them private is to ensure that clients don't have unrestricted access to them.

- In general, it is considerably safer to allow clients to *read* the values of the instance variables than to *write* those values. As a result, *setter* methods are far less common than *getters* in object-oriented design.

- Many classes are designed in an even higher level of security by making it impossible to change the values of any instance variables after an object has been created (i.e., *immutable*). Although it is still possible to change the contents of an object by assigning another object to it, there is no way to change the individual fields in such an object independently. (Remember Python strings?)

Constructors

- In addition to method prototypes, class definitions typically include one or more ***constructors****,* which are used to initialize an object. (Remember `__init__()` in Python?)

- 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 (i.e., overloading) as long as the constructors have different signatures.

- 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 version being called is determined by the arguments.

Initializer List

- An ***initializer list*** is sometimes used in initializing the data members of a class. The initializer list appears just before the brace that begins the body of the constructor and is set off from the parameter list by a colon. The elements of the initializer list should be in one of the following forms:

  - The name of a field in the class, followed by an initializer for that field enclosed in parentheses.

  - A superclass constructor (will be discussed later with ***inheritance***).

Initializer List

- The following example of initialization by assignments

```
Point(int xc = 0, int yc = 0) {
    x = xc;
    y = yc;
}
```

can be optimized for efficiency by using the initializer list:

```
Point(int xc = 0, int yc = 0) : x(xc), y(yc) {}
```

- Using assignments, the members are initialized with default values, and then reassigned in the constructor body.

- Using the initializer list, the members are created and initialized only once, with the given value.

- Constant members can only be initialized using the initializer list, because assigning a value to a constant is not allowed.

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)".
 * E.g., cout << "pt = " << pt.toString() << endl;
 */

   std::string toString();
```

The `point.h` Interface

```cpp
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. E.g., cout << "pt = " << pt << endl;
 */

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

#endif
```

The `point.cpp` Implementation

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

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

/* Constructo

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

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

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.

But this approach does not keep the users from calling it with only one argument. And the default values of the parameters should be specified in the constructor prototype.

```cpp
// These two constructors c   e written into
// one, using default ar   ents:
Point(int xc = 0, int yc = 0) {
   x = xc;
   y = yc;
}
// An initializer list can simplify it to:
Point(int xc = 0, int yc = 0):x(xc), y(yc) {};
```

The `point.cpp` Implementation

```cpp
/* 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();
}
```

If `integerToString()` *is not allowed, how can you do this? Using string streams.*

Why no `Point::` *here? This is a free function.*

Overloading Operators

- Besides *methods*, you can also implement some useful *operators*, even those existing ones in C++.

- 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.

- For instance, one of the most useful operators to overload is the insertion operator `<<`, which makes it easy to display values of that class on an output stream.

- We can use the following line:

```
cout << "(" << pt.getX() << "," << pt.getY() << ")";
```

or define a string representation of the object:

```
cout << pt.tostring(); // ≈ print(pt.__str__()) in Python
```

but it would be better if we could just:

```
cout << pt; // ≈ print(pt) in Python
```

Overloading Operators

- The prototype and implementation for the overloaded **<<** in the **Point** class is:

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

- Since stream variables cannot be copied, the **ostream** argument must be passed by reference.

- The **<<** operator has a chaining behavior of returning the output stream, therefore, the definition of **operator<<** must also return its result by reference.

- Other operators that can be overloaded:

any of the following operators: + - * / % ^ & | ~ ! = < > += -= *= /= %= ^= &= |= << >> >>= <<= == != <= >= <=> (since C++20) && || ++ -- , ->* -> ( ) [ ]

Overloading Operators

- To overload the **==** operator, i.e., to allow statements like this:

```
if (p1 == p2)
```

We can treat this operator as a free function as if:
`operator==(p1,p2)`

- The free function approach:

```cpp
// in point.h, inside the class Point definition
friend bool operator==(Point p1, Point p2);
// in point.h, outside the class Point definition
bool operator==(Point p1, Point p2);
// in point.cpp
bool operator==(Point p1, Point p2) {
    return p1.x == p2.x && p1.y == p2.y;
    return p1.getX() == p2.getX() && p1.getY() == p2.getY();
}
```

What if there are no public getters?

- The method-based style:

```cpp
// in point.h, inside the class Point definition
bool operator==(Point pt);
// in point.cpp
bool Point::operator==(Point pt) {
    return x == pt.x && y == pt.y;
}
```

`p1` is the receiver and the argument `p2` is copied to parameter `pt`. The private data of a class are private to the class and not to the object.

Overloading Operators

- When you define operators for a class, you can write them either as class methods (object-oriented programming style) or as free functions (procedural programming style). Each style has its own advantages and disadvantages.

    – Defining an operator as a class method means that the operator is part of the class and therefore has free access to the private instance variables and other methods. But when you use this style to overload a binary operator, the left operand is the receiver object and the right operand is passed as a parameter.

    – Defining an operator as a free function often produces code that is easier to read (the operands for a binary operator are both passed as parameters as usual), but also means that the operator function must be designated as a `friend` to refer to the private data that do not have public getters.

A Strategy for Defining New Classes

- Think generally about how clients are likely to use the class.
- Determine what information belongs in the private state of each object.
- Define a set of constructors to create new objects.
- Enumerate the operations (including the overloaded functions and operators) that will become the public methods of the class.
- If necessary, define free functions that can manipulate the objects outside the class definition, but make them friends to the class.
- Code (as the implementer) and test (as the client) the implementation.
- Keeping implementation details away from the client is likely to foster *simplicity*, *flexibility*, and *security*, as described in the previous lectures (compare this to libraries).

Example: 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 (e.g., 1/3 = 0.3̇)
- Even if you think 1/10 = 0.1 is exact, when

> *Try the following code:*
> `cout<<setprecision(20)<<0.1;`

  `double`, 0.1 is represented internally as an approximation ([Why 0.1 Does Not Exist In Floating-Point?](#)).
- 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 private instance variables: ***numerator*** and ***denominator***.

- 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 following properties should be enforced: (why?)

  - The fraction is always expressed in lowest term.

  - The denominator is always positive.

  - The rational number 0 is always represented as the fraction 0/1.

  Since these values never change once a new `Rational` is created, these properties will remain in force.

- The class overloads the standard arithmetic operations (`+` , `-` , `*` , `/`) to allow the use of conventional mathematical notation.

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

```cpp
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

```cpp
/*
 * 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 */
```

Why is the prototype of `gcd` *not in* `rational.h` *or the* `Rational` *class definition?*

The `rational.cpp` Implementation

```cpp
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;
    }
}
```
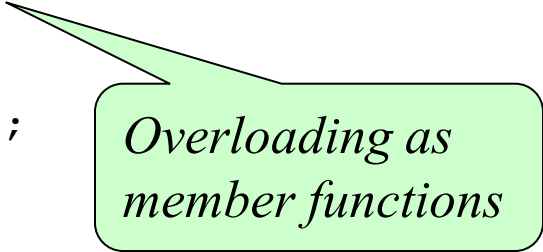
The `rational.cpp` Implementation

```cpp
/* 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);
}
```

Overloading as member functions

The `rational.cpp` Implementation

```cpp
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;
}
```

Exercise: overloading as free functions?

```cpp
// in rational.h, inside the class Rational definition
friend Rational operator+(Rational r1, Rational r2);
friend Rational operator-(Rational r1, Rational r2);
friend Rational operator*(Rational r1, Rational r2);
friend Rational operator/(Rational r1, Rational r2);
// in rational.h, outside the class Rational definition
Rational operator+(Rational r1, Rational r2);
Rational operator-(Rational r1, Rational r2);
Rational operator*(Rational r1, Rational r2);
Rational operator/(Rational r1, Rational r2);
// in rational.cpp
Rational operator+(Rational r1, Rational r2) {
    return Rational(r1.num * r2.den + r2.num * r1.den, r1.den * r2.den);
}
Rational operator-(Rational r1, Rational r2) {
    return Rational(r1.num * r2.den - r2.num * r1.den, r1.den * r2.den);
}
Rational operator*(Rational r1, Rational r2) {
    return Rational(r1.num * r2.num, r1.den * r2.den);
}
Rational operator/(Rational r1, Rational r2) {
    return Rational(r1.num * r2.den, r1.den * r2.num);
}
```

Designing a token scanner class

- Many applications need to divide a string into words, or more generally, into logical units that may be larger than a single character. In computer science, such units are typically called *tokens*.

- Different applications, however, define tokens in all sorts of different ways, which means that the `TokenScanner` class must give the client some control over what types of tokens are recognized. E.g., the following line of text can be scanned into many different possible sequences of tokens.

```
cout << "hello, world" << endl;
```

- The Stanford C++ library includes a `TokenScanner` class that offers considerable flexibility without sacrificing simplicity.

Designing a token scanner class

- From the client's point of view:

> *Set the input for the token scanner to be some string or input stream.*
> **while (***more tokens are available***) {**
>   *Read the next token.*
>   *Perform other operations using the token.*
> **}**

- From the implementer's point of view:

    - A **setInput** method that allows clients to specify the token source, ideally, overloaded to take either a string or an input stream.

    - A **hasMoreTokens** method that tests whether the token scanner has any tokens left to process.

    - A **nextToken** method that scans and returns the next token.

`...etInput(str)` *or* **`scanner.setInput(infile)`** ...t for this scanner to the specified string or input stream.
`...asMoreTokens()` 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.

TUTORIAL

Encapsulating Programs as Classes

- Particularly as programs grow larger and more complex, it is often useful to construct an object that represents the program, as opposed to writing the code directly inside the `main` function. Doing so has several advantages, including making it possible to associate instance variables with the program.

- The text uses a revised version of the checkout-line simulation from Chapter 5 to illustrate this technique. The original `main` method has the following form:

```
int main() {
    int nServed;
    int totalWait;
    int totalLength;
    runSimulation(nServed, totalWait, totalLength);
    printReport(nServed, totalWait, totalLength);
    return 0;
}
```

Example: Simulate a checkout line

```cpp
void runSimulation(int & nServed, int & totalWait, int & totalLength) {
    Queue<int> queue;
    int timeRemaining = 0, serveTime = 0, waitTime = 0, totalCustomer = 0;
    nServed = 0;
    totalWait = 0;
    totalLength = 0;
    for (int t = 0; t < SIMULATION_TIME; t++) {
        if (randomChance(ARRIVAL_PROBABILITY)) {
            queue.enqueue(t);
            cout << t << ", Customer No." << ++totalCustomer << " comes in." << endl;
        }
        if (timeRemaining > 0)
            if (--timeRemaining == 0)
                cout << t << ", Customer No." << nServed << " was served for: " << serveTime << endl;
        else
            if (!queue.isEmpty()) {
                waitTime = t - queue.dequeue();
                totalWait += waitTime;
                nServed++;
                cout << t << ", Customer No." << nServed << " waited for: " << waitTime << endl;
                timeRemaining = randomInteger(MIN_SERVICE_TIME, MAX_SERVICE_TIME);
                serveTime = timeRemaining;
            }
        totalLength += queue.size();
    }
}
```

Encapsulating Programs as Classes

- After encapsulating the code in a class called **CheckoutLineSimulation**, and the **main** method has the following form:

```
class CheckoutLineSimulation {
public:
    void runSimulation() {...}
    void printReport() {...}
private:
    int nServed;
    int totalWait;
    int totalLength;
};
int main() {
    CheckoutLineSimulation simulation;
    simulation.runSimulation();
    simulation.printReport();
    return 0;
}
```

Encapsulating Programs as Classes

- The primary advantage of doing so is that classes provide better encapsulation. The fact that access to any private data is limited to the class itself means that it is much safer to use private instance variables to share information than it is to use global variables, which offer no such security.

- The information shared between these functions can now be stored in instance variables and need not be passed as arguments. Being able to share access to such data among all the methods in the class substantially reduces the size and complexity of the parameter lists.

A Note on Object-Oriented Programming

- *Encapsulation*, *Polymorphism*, and *Inheritance*.

- As you may know from other object-oriented languages such as Python, 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, we postpone the discussion of inheritance until later in the lectures, focusing instead on the use of classes for *encapsulation*.

- We have also touched upon *polymorphism* by showing you some examples of function/operator overloading. We will discuss it more when we learn about template later.

The End