Outline

Objectives

1. Generic Programming
2. Data Abstraction
3. Problem Solving Applied: Color Image Processing
4. Recursion
5. Class Templates
6. Inheritance
7. Virtual Methods
8. Problem Solving Applied: Iterated Prisoner's Dilemma

Objectives

Develop problem solutions in C++ containing:

- Function Templates
- Overloaded Operators
- Image Processing Examples
- Recursive Member Functions
- Class Templates
- Class Hierarchies
- An Implementation of the Iterated Prisoner's Dilemma Game.

Generic Programming

Generic Programming

- Generic programming supports the implementation of a type independent algorithm.
- Type independent algorithms can be defined in C++ using the keyword `template,` and at least one parameter.
- The compiler generates a unique instance of the template for each specified type.

Function Templates

- A function template is a parameterized function definition.

Syntax:

template<typename identifier1[**, typename** identifier2[,…]]**>**
return_type function_name**(**[parameter_list]**) { … }**

Example

```
template<typename Dtype>
void swapTwo(Dtype& a, Dtype& b) {
       Dtype temp = a;
       a = b;
       b = temp;
}
```

Prototype

```
template<typename Dtype>
void swapTwo(Dtype&, Dtype&);
```

Instantiating Templates

Example:

```
template<typename Dtype& a, Dtype& b>
    void swapTwo(Dtype& a, Dtype& b);
//prototype
…
void main() {
  double x(1.0), y(5.7);
  char ch1('n'), ch2('o');
  swapTwo(x,y);
  swapTwo(ch1, ch2);
  cout << x << ',' << y << endl
       << ch1 << ch2 << endl;
  ...
}
```

Output:

```
5.7, 1.0
on
```

Data Abstraction

Why Data Abstraction?

- It is common to have to program with concepts that are not available as built-in or predefined data types.
- It is often necessary to have to work with multiple programmers to develop problem solutions.
- Defining a new type to represent the concept ensures that all programmers work with same definition of the concept.
- Build increasingly complex types from

Data Abstraction

- C++ supports object-oriented programming through the use of programmer-defined data types.
  - Data Abstraction
- User-defined types can be as easy to use as pre-defined types.
  - Operator Overloading
- A well-designed type provides a good public interface while hiding the details of its implementation.
  - Encapsulation

Overloading Operators

- Overloading operators allows the programmer to redefine the behavior of existing operators to work with programmer-designed types.
- Restrictions:
  - It is not possible to define new operators.
  - Four operators cannot be overloaded:
    - ::   .   .*   ?:
  - Must adhere to C++ syntax for the operator.

friends

- Non-members of a class cannot access protected or private members of the class.
- Functions and other classes can be declared as a `friend` of the class.
  - Friends are not members of the class and thus are not affected by visibility specifiers.
  - Implementations of friend functions outside of the class cannot have the friend modifier.

Bitwise Operators

- Bitwise operators perform an
  operation on each of the bits in the operand.
- C++ supports 3 binary and 1 unary bitwise operators:
  - Bitwise or |
  - Bitwise and &
  - Bitwise exclusive or `
  - Bitwise not ~

Truth Table of Bitwise Operators

A	B	~A	A|B	A`B	A&B
0001	0001	1110	0001	0000	0001
0010	0010	1101	0010	0000	0010
0011	0100	1100	0111	0111	0000
0100	0111	10111	0111	0011	0100

Problem Solving Applied: Color Image Processing

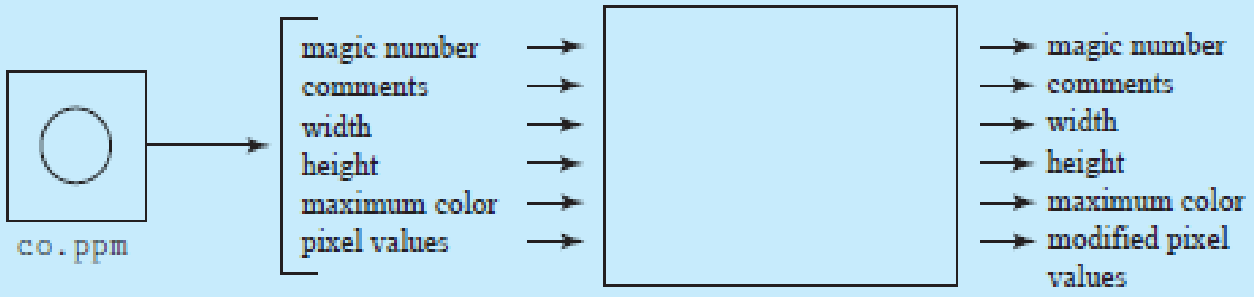
Problem Solving Applied: Color Image Processing

1. PROBLEM STATEMENT

Modify a color digital image by performing a smoothing process on the image.

2. INPUT/OUTPUT DESCRIPTION

The input to this program is the data in the image file Io.ppm. The output is the modified image. We must first read the header information to determine the size of the image. We can then use the information in the header to allocate memory and input the pixel values.



3. HAND EXAMPLE

To perform the smoothing process on the image, we will take an average of the current pixel and the four adjacent pixels; the pixel to the left, the pixel above, the pixel to the right, and the pixel below. We will replace the original pixel value with the smoothed pixel value as we perform the calculations. For our hand example, we will determine the smoothed value for one pixel from the image of Io.

Original image:

```
143 159 211 142 160 210 142 160 210
136 158 208 135 158 208 136 156 207
140 153 206 142 151 206 142 151 206
```

The current pixel has a red value of 135, a green value of 158, and a blue value of 208. The smoothed value for this pixel is calculated as follows:

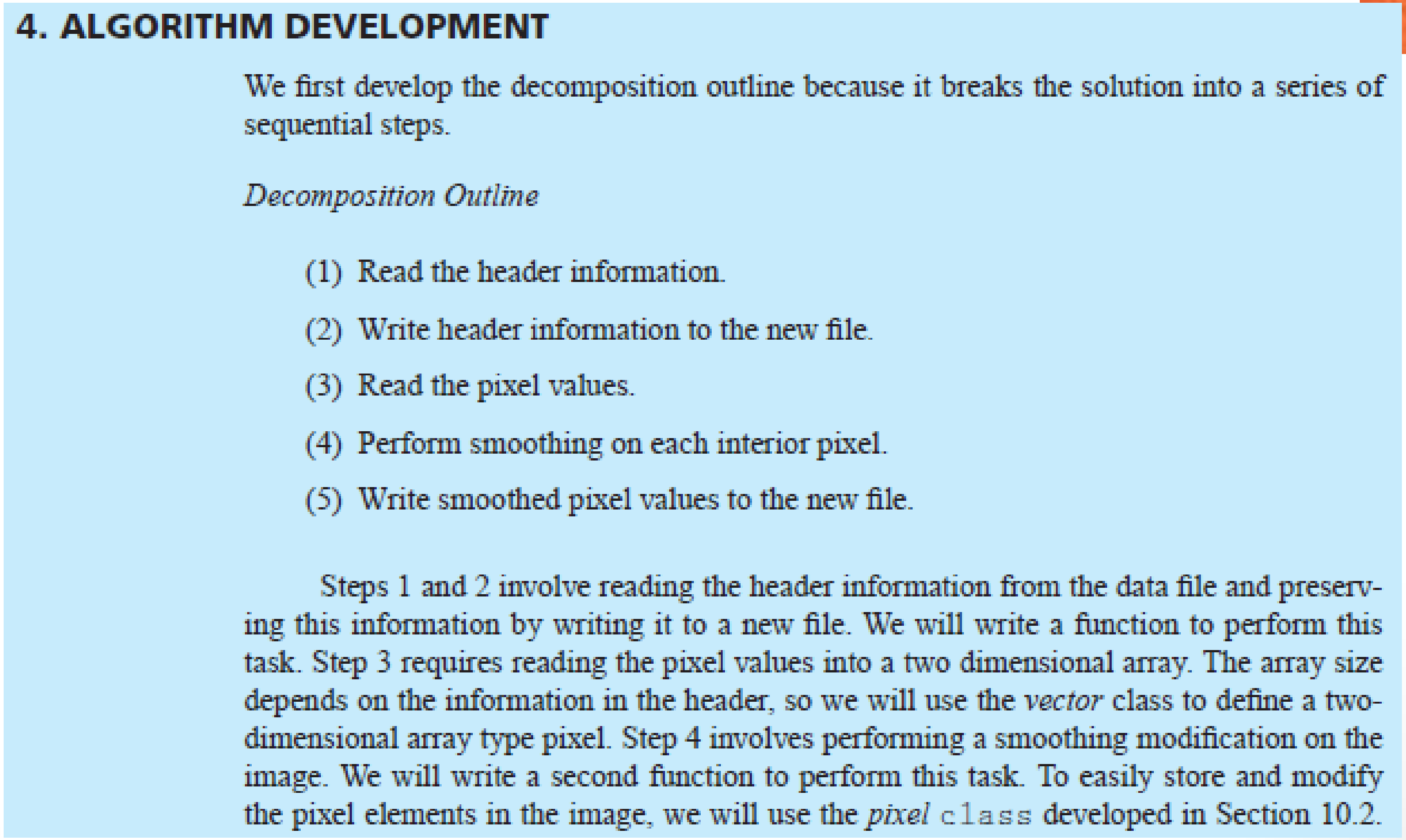
```
red value -> (135 + 136 + 142 +136 +142)/5 = 138
green value -> (158 +158 + 160 + 156 +151)/5 = 156
blue value -> (208 + 208 + 210 + 207 + 206)/5 = 207
```

Modified image:

```
143 159 211 142 160 210 142 160 210
136 158 208 138 156 207 136 156 207
140 153 206 142 151 206 142 151 206
```

This process is repeated for every interior pixel in the image. Pixels on the boundaries are missing one of the four adjacent pixels just described (corner pixels are missing two) and will not be modified in this application.

Problem Solving Applied: Color Image Processing



4. ALGORITHM DEVELOPMENT

We first develop the decomposition outline because it breaks the solution into a series of sequential steps.

Decomposition Outline

(1) Read the header information.

(2) Write header information to the new file.

(3) Read the pixel values.

(4) Perform smoothing on each interior pixel.

(5) Write smoothed pixel values to the new file.

Steps 1 and 2 involve reading the header information from the data file and preserving this information by writing it to a new file. We will write a function to perform this task. Step 3 requires reading the pixel values into a two dimensional array. The array size depends on the information in the header, so we will use the *vector* class to define a two-dimensional array type pixel. Step 4 involves performing a smoothing modification on the image. We will write a second function to perform this task. To easily store and modify the pixel elements in the image, we will use the *pixel* `class` developed in Section 10.2.

Recursion

Recursion

- Recursion is a powerful tool for solving certain classes of problems where:
  - the problem solution can be expressed in terms of the solution to a similar, yet smaller problem.
- Redefinition of the problem continues in an iterative nature until:
  - a unique solution to a small version of the problem is found.
- This unique solution is then used, in a reverse iterative nature until:
  - the solution to the original problem is returned.

Recursion

- Programming languages that support recursion allow functions to call themselves.
- Each time a function calls itself, the function is making a recursive function call.
- Each time a function calls itself recursively, information is pushed onto the runtime stack.
- Each time a recursive function calls returns, information is popped off the stack. The return value, along with the information on the stack, is used to solve the next iteration of the problem

Recursive Functions

- A recursive function requires two blocks:
  - a block that defines a terminating condition, or return point, where a unique solution to a smaller version of the problem is returned.
  - a recursive block that reduces the problem solution to a similar but smaller version of the problem.

Example: Recursive Function

$$f(n) = n! = \{1 : n = 0, n * f(n-1) : n >= 1\}$$

- f(0) = f(1) = 1 Unique solution(s).
- f(n) = n*f(n-1) recursive definition.
- Thus f(n) can be determined for all integer values of n >= 0;

Recursive Factorial Function

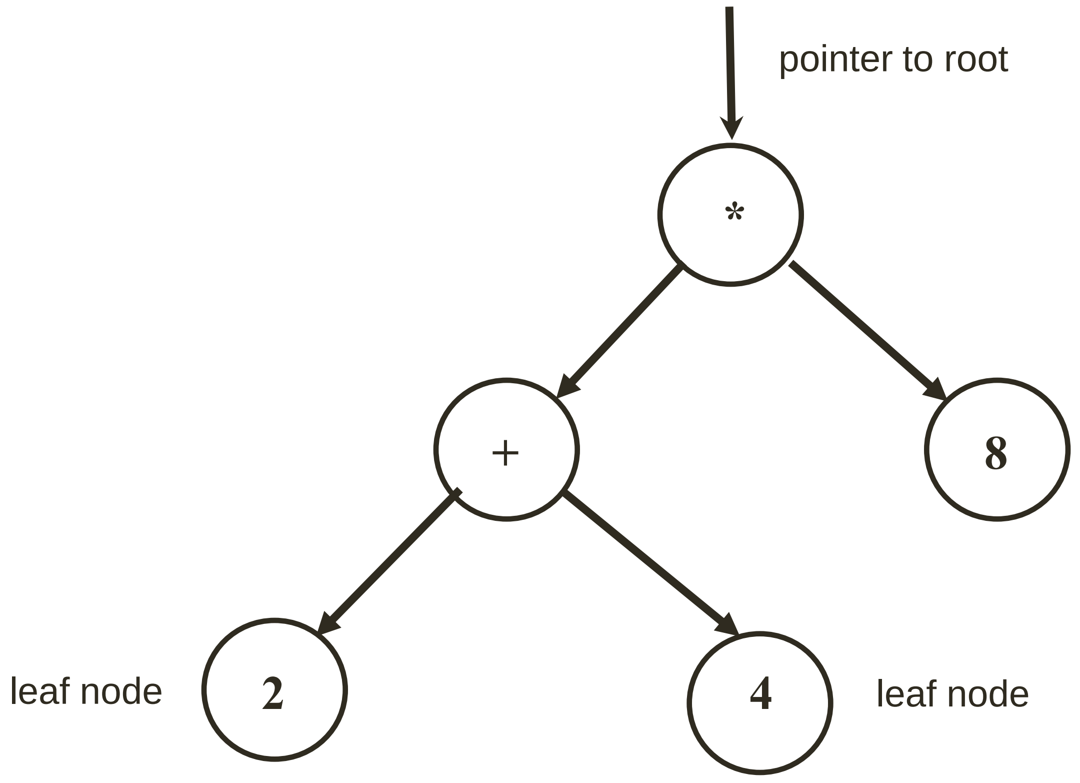
Example:

```
long factorial(int n) {
  //termination condition
  if (n < 2)
    return 1; //unique solution
  return n*factorial(n-1); // recurse
}
```

Binary Tree Abstraction

- A binary tree maintains two *links* between *nodes*.
- The links are referred to as the *left child* and the *right child*.
- The first node is called the *root*.
- Each child(left and right) may serve as the root to a *subtree.*
- A node without children is referred to as a *leaf* node.

Example Diagram



A BinaryTree Class Implementation

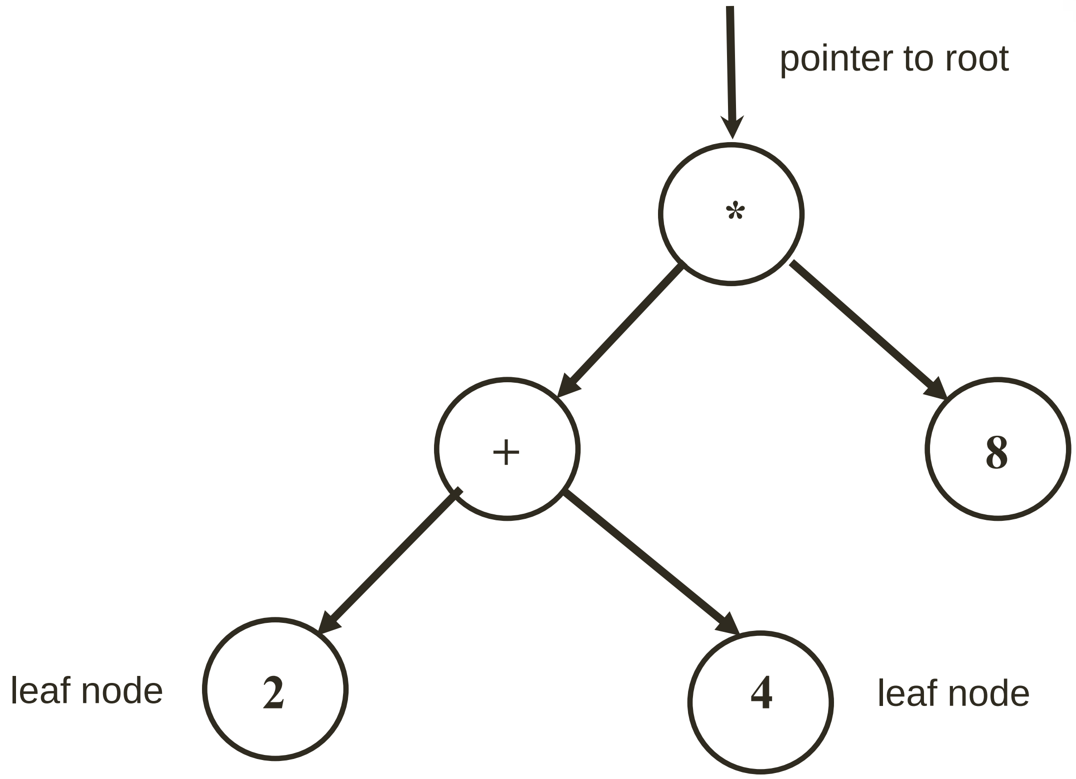
- A BinaryTree class with one attribute:
  - A pointer (node*) to the root of the binary tree.
- **Methods**:
  - insert()
  - delete()
  - print()
  - inOrder(), preOrder(), postOrder()

BinaryTree Class

- **Implementation of insert:**
  - Insert into empty BinaryTree establishes the root.
  - Each subsequent node is inserted in following order:
    - values less than root are placed in the root's left subtree
    - values greater than root are placed in the root's right subtree

preOrder Traversal

- visit node
- visit left child
- visit right child

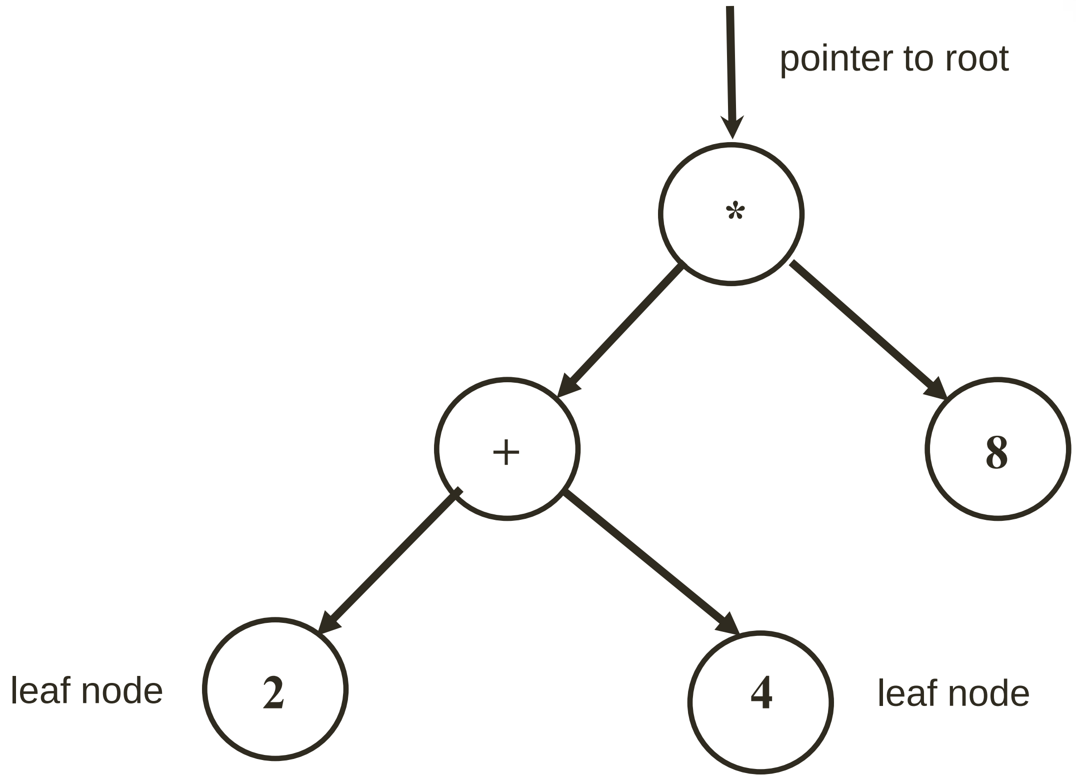


Traversal Order:

```
*+248
```

postOrder Traversal

- visit left child
- visit right child
- visit node

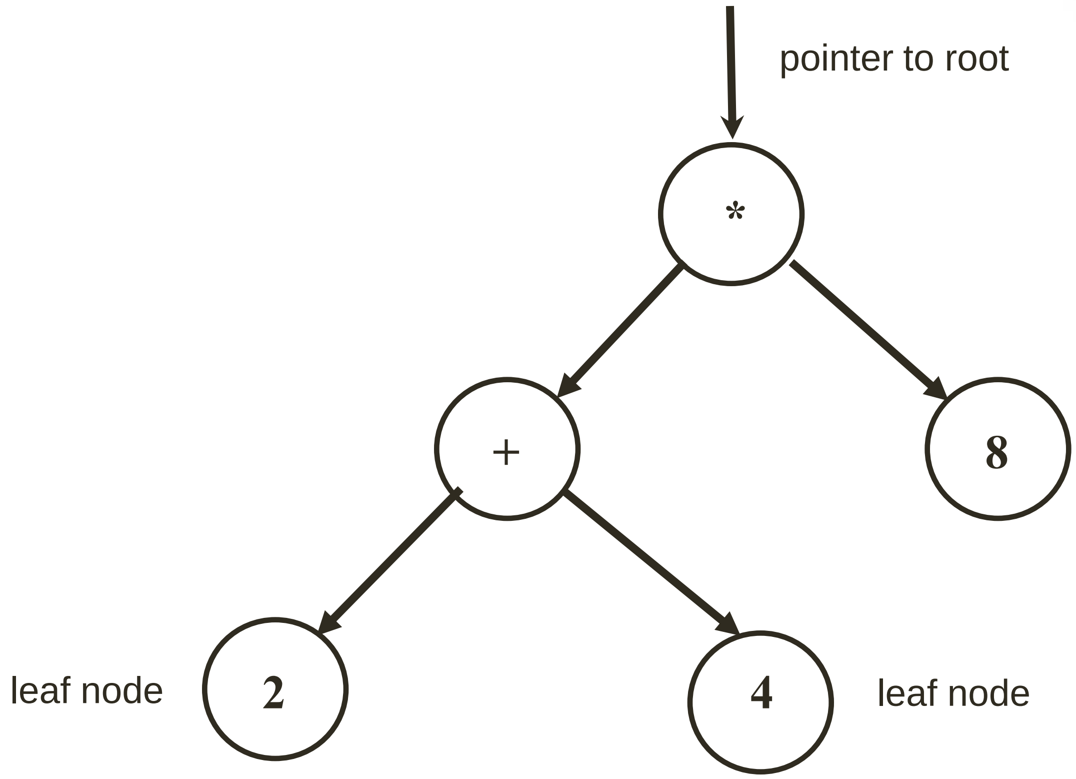


Traversal Order:

24+8*

inOrder Traversal

- visit left child
- visit node
- visit right child



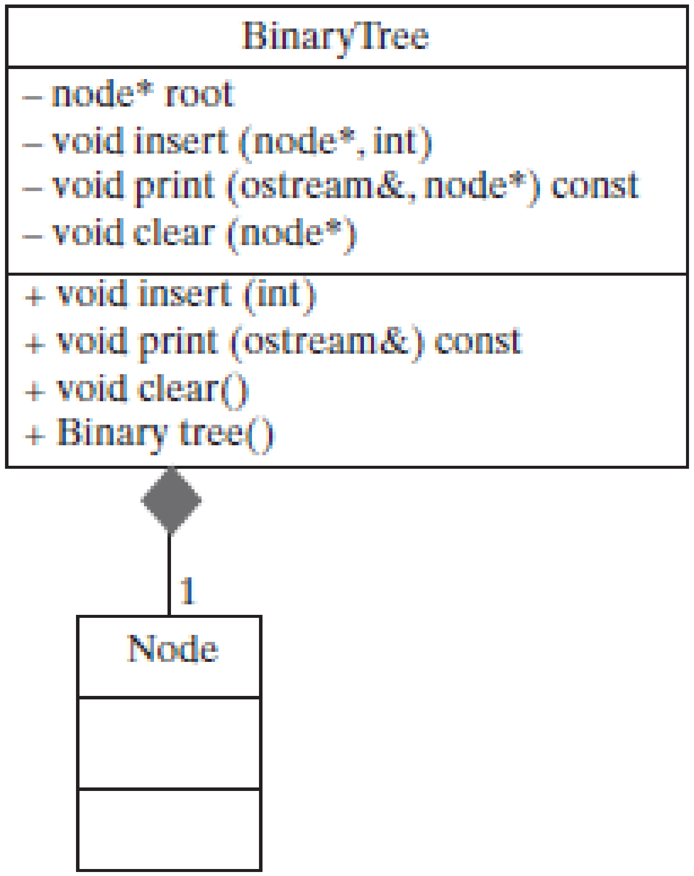
Traversal Order:

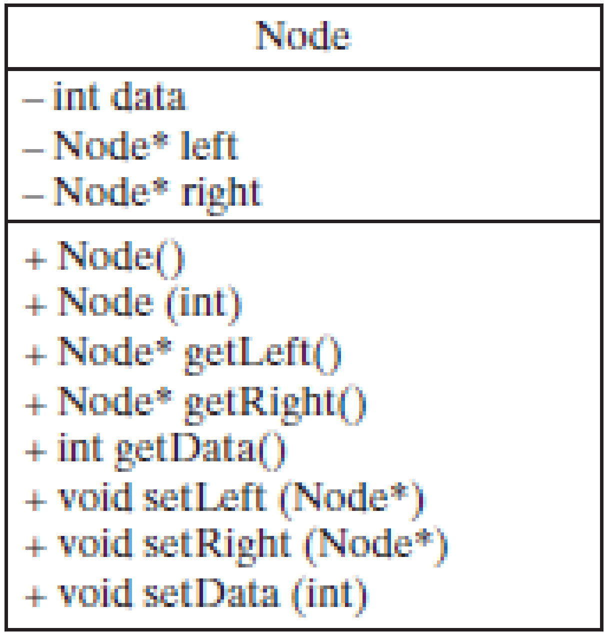
2+4*8

BinaryTree Class

- Recursive Methods:
  - print(), insert(), clear()
- Recursive methods are overloaded.
- public version is non-recursive.
- public version is called once.
- public version calls private recursive version.
- Recursive version calls itself.

UML Class Diagrams

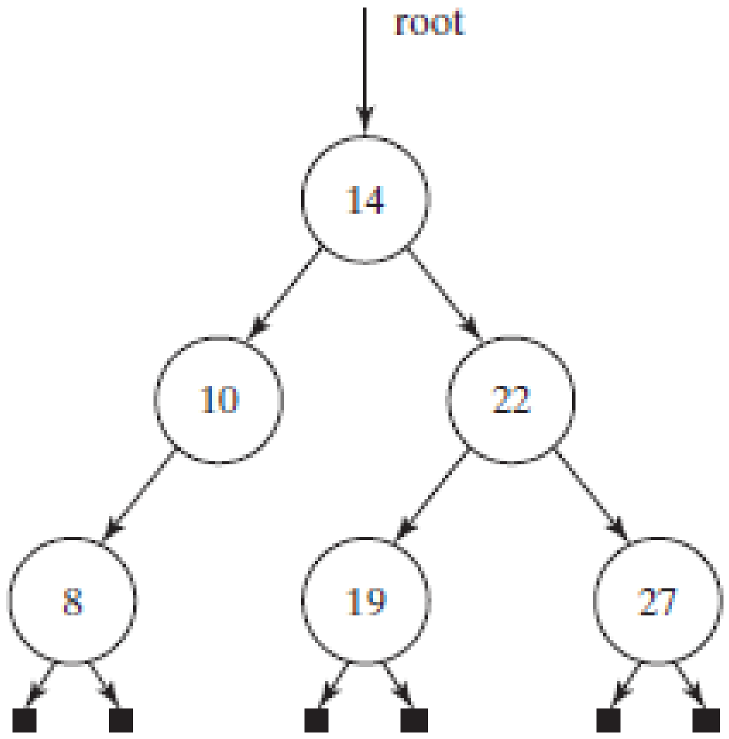




Binary Search Tree

- A Binary Search Tree is an example of an ordered binary tree where:
  - Each node has a value.
  - The left subtree of a node contains only nodes with values less that the node's value.
  - The right subtree of a node contains only nodes with values greater than the node's value.

Binary Search Tree Example



Class Templates

Class Templates

- A binary tree is an ordered collection of nodes.
- Each node has a data **value**, a **right child** and a **left child**.
- The data type of the **right** and **left child** is `node*`.
- The data type of the node value is parameterized to form a `class template`.
- The binary tree `template` also parameterizes the node type.

Suggestions for Writing Templates

- Get a non-template version working first.
- Establish a good set of test cases.
- Measure performance and tune.
- Review implementation:
  - Which types should be parameterized?
  - Convert non-parameterized version into template.
  - Test template against established test cases.

Inheritance

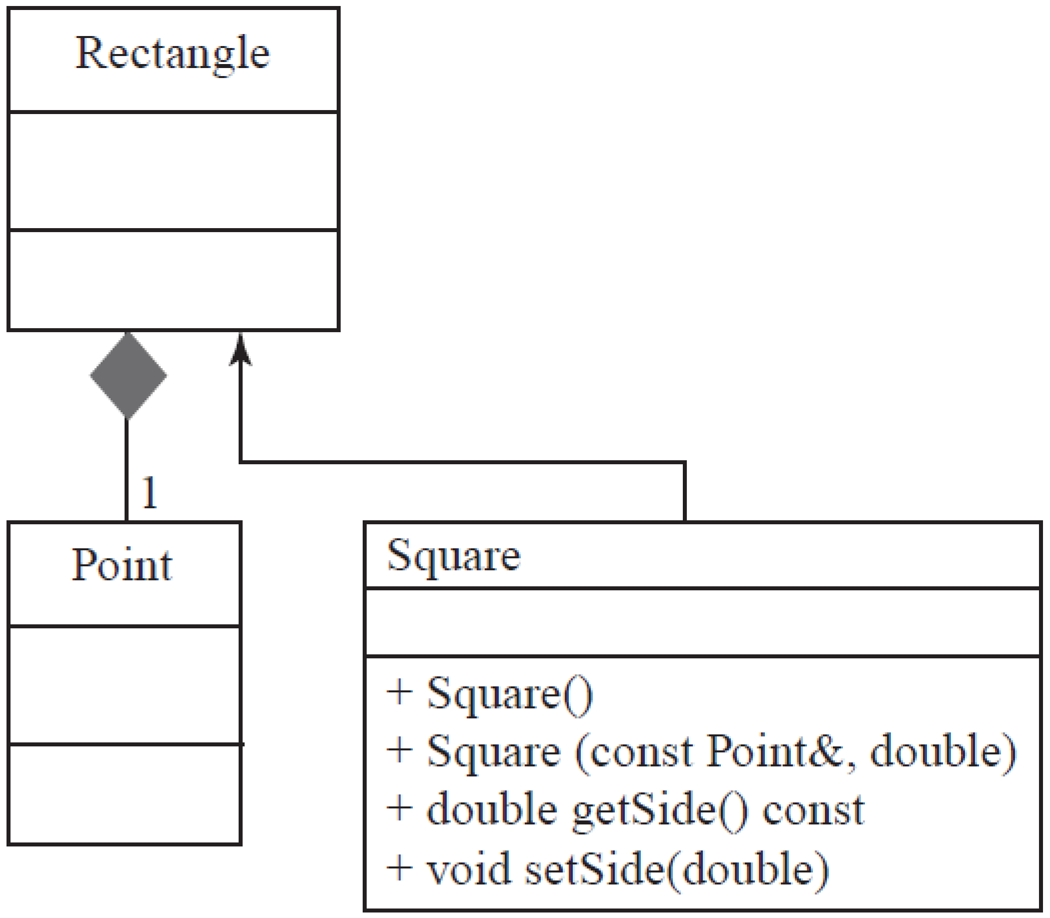
Inheritance

- Inheritance is a means by which one class acquires the properties--both attributes and methods--of another class.
- When this occurs, the class being inherited from is called the *base class*.
- The class that inherits is called the *derived class*.

C++ and Public Inheritance

- The `private` members of a base `class` are only accessible to the base `class` methods.
- The `public` members of the base `class`, are accessible to the derived `class` and to clients of the derived `class`.
- The `protected` members of the base `class`, are accessible to members of the derived `class`, but are not accessible to clients of the base `class` or clients of the derived `class`.

UML Inheritance Diagram



Base Class (Rectangle)

Example:

```
class Rectangle {
private:
  double width, height;
  Point origin;
public:
  Rectangle();
  Rectangle(double w, double h, double x, double y);
//Accessors

//Mutators
…
};
```

Derived Class (Square)

Example:

```
class Square: public Rectangle {
public:
  Square();
  Square(const Point&, double);
//Accessors
  double getSide() const;
//Mutators
  void setSide(double);
…
};
```

Constructors and Inheritance

- When an object is created, its constructor is called to build and initialize the attributes.
- With inheritance, the invocation of constructors starts with the most base class at the root of the class hierarchy and moves down the path to the derived classes constructor.
- If a class is to use the parameterized constructor of a base class, it must explicitly invoke it.

Constructors

Example:

```
Rectangle::Rectangle(double w, double h,
                     double x, double y) : origin(x,y) {
…
}

Square::Square(const Point& p,
           double side) : Rectangle(side, side, p.x, p.y) {
…
}
```

Virtual Methods

Methods

- All methods are, by default, non-virtual methods. Binding of method call is determined by static type of calling object.

Example:

```
Rectangle r1;
Square s1;
r1 = s1;
r1.print(cout); //calls print defined in Rectangle
```

Virtual Methods

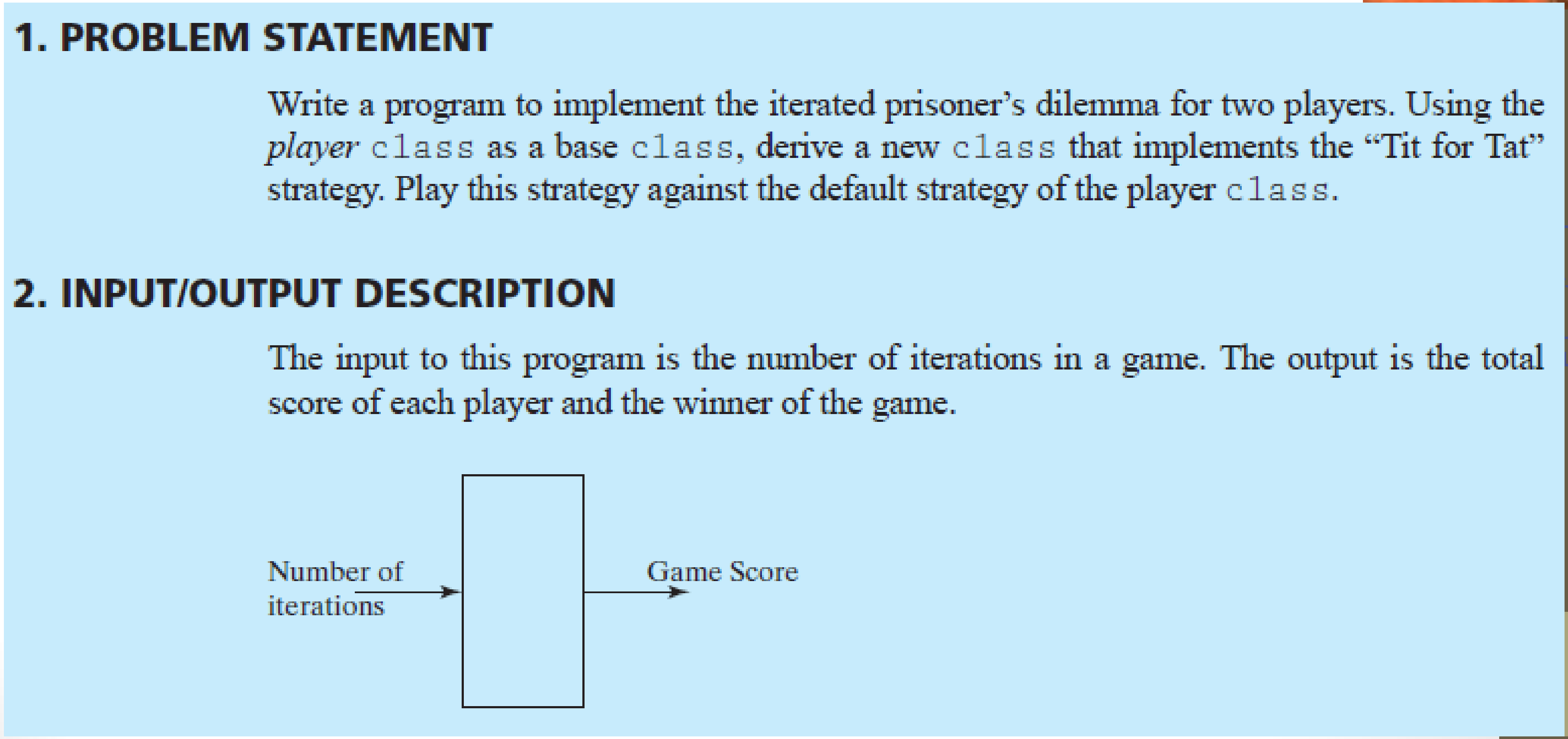
- If a method is defined to be `virtual`, and pointers or references to objects are used, then dynamic binding is supported.

Example:

```
Rectangle *r1;
Point p2;
Square s1(p2,5);
r1 = &s1;
R1->print(cout); //calls print defined in Square
```

Problem Solving Applied: Iterated Prisoner's Dilemma

Problem Solving Applied: Iterated Prisoner's Dilemma

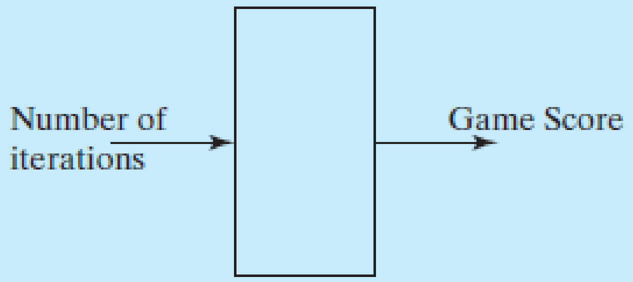


1. PROBLEM STATEMENT

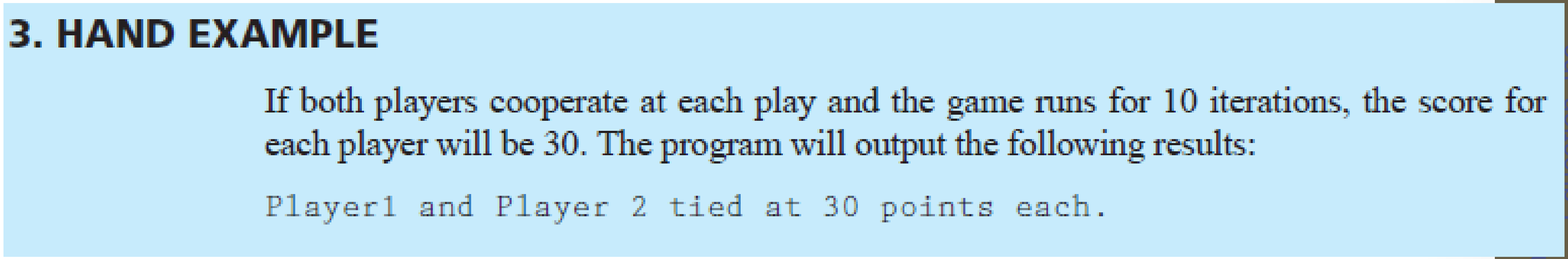
Write a program to implement the iterated prisoner's dilemma for two players. Using the *player* `class` as a base `class`, derive a new `class` that implements the "Tit for Tat" strategy. Play this strategy against the default strategy of the player `class`.

2. INPUT/OUTPUT DESCRIPTION

The input to this program is the number of iterations in a game. The output is the total score of each player and the winner of the game.



Problem Solving Applied: Iterated Prisoner’s Dilemma

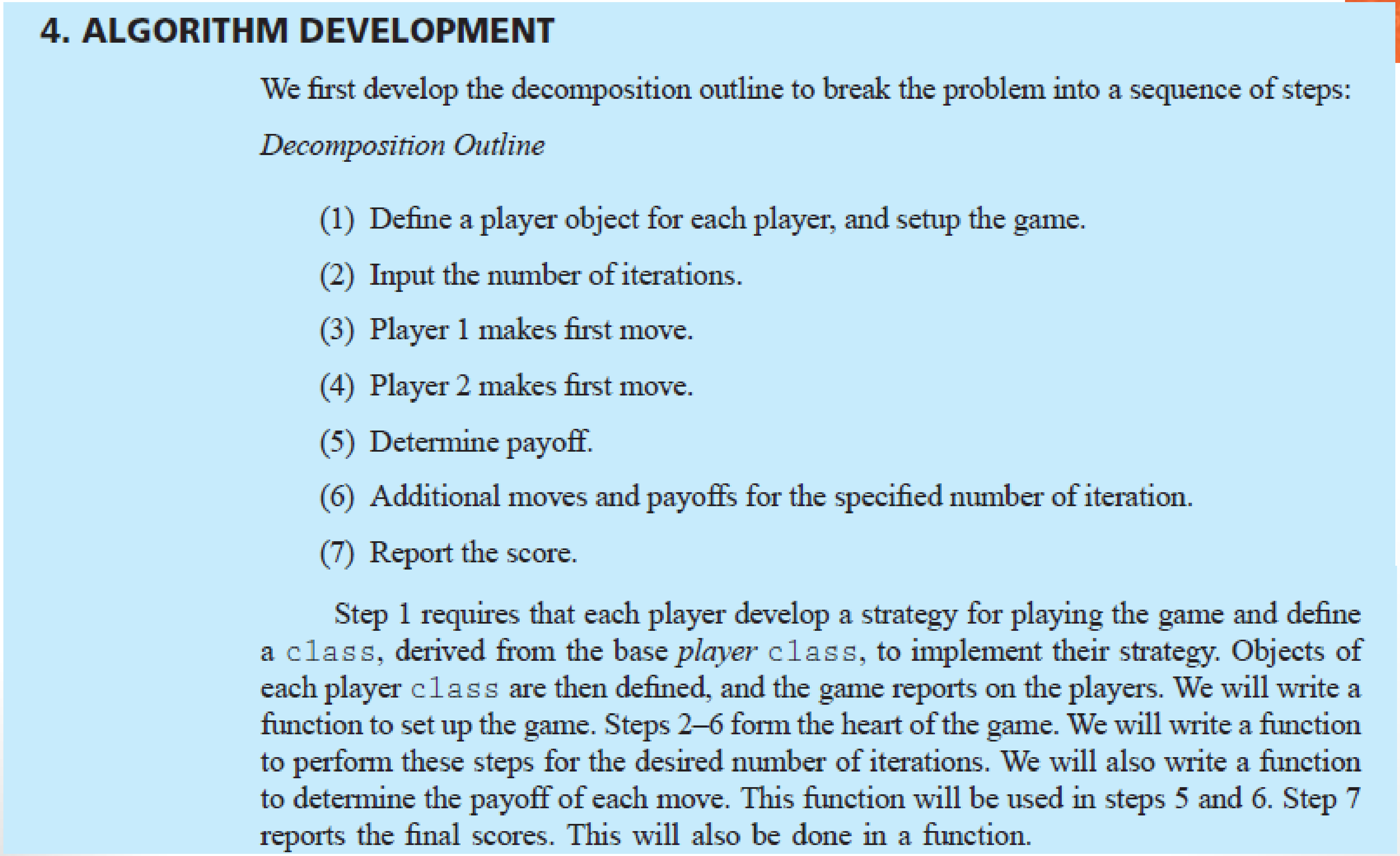


3. HAND EXAMPLE

If both players cooperate at each play and the game runs for 10 iterations, the score for each player will be 30. The program will output the following results:

```
Player1 and Player 2 tied at 30 points each.
```

Problem Solving Applied: Iterated Prisoner's Dilemma



4. ALGORITHM DEVELOPMENT

We first develop the decomposition outline to break the problem into a sequence of steps:

Decomposition Outline

(1) Define a player object for each player, and setup the game.
(2) Input the number of iterations.
(3) Player 1 makes first move.
(4) Player 2 makes first move.
(5) Determine payoff.
(6) Additional moves and payoffs for the specified number of iteration.
(7) Report the score.

Step 1 requires that each player develop a strategy for playing the game and define a `class`, derived from the base *player* `class`, to implement their strategy. Objects of each player `class` are then defined, and the game reports on the players. We will write a function to set up the game. Steps 2–6 form the heart of the game. We will write a function to perform these steps for the desired number of iterations. We will also write a function to determine the payoff of each move. This function will be used in steps 5 and 6. Step 7 reports the final scores. This will also be done in a function.