Chapter 14: More About Classes
14.1

Instance and Static Members
Instance and Static Members

- **instance variable**: a member variable in a class. Each object has its own copy.

- **static variable**: one variable shared among all objects of a class

- **static member function**: can be used to access static member variable; can be called before any objects are defined
Contents of Tree.h

1   // Tree class
2   class Tree
3   {
4       private:
5           static int objectCount; // Static member variable.
6       public:
7           // Constructor
8           Tree()
9               { objectCount++; } 
10          
11           // Accessor function for objectCount
12           int getObjectCount() const
13               { return objectCount; } 
14          
15       
16       // Definition of the static member variable, written
17       // outside the class.
18       int Tree::objectCount = 0;
Program 14-1

1 // This program demonstrates a static member variable.
2 #include <iostream>
3 #include "Tree.h"
4 using namespace std;
5
6 int main()
7 {
8     // Define three Tree objects.
9     Tree oak;
10    Tree elm;
11    Tree pine;
12
13     // Display the number of Tree objects we have.
14    cout << "We have " << pine.getObjectCount()
15          << " trees in our program!\n"
16    return 0;
17 }

Program Output
We have 3 trees in our program!
Three Instances of the Tree Class, But Only One `objectCount` Variable
**static member function**

- Declared with `static` before return type:
  ```cpp
  static int getObjectCount() const
  { return objectCount; }  
  ```

- Static member functions can only access static member data

- Can be called independent of objects:
  ```cpp
  int num = Tree::getObjectCount();  
  ```
Modified Version of Tree.h

1    // Tree class
2    class Tree
3    {
4        private:
5            static int objectCount;  // Static member variable.
6        public:
7            // Constructor
8                Tree()
9                { objectCount++; }
10
11        // Accessor function for objectCount
12                static int getObjectCount() const
13                { return objectCount; }
14    }
15
16    // Definition of the static member variable, written
17    // outside the class.
18    int Tree::objectCount = 0;

Now we can call the function like this:

cout << "There are " << Tree::getObjectCount()
     << " objects.\n";
14.2 Friends of Classes
Friends of Classes

- **Friend**: a function or class that is not a member of a class, but has access to private members of the class
- A friend function can be a stand-alone function or a member function of another class
- It is declared a friend of a class with `friend` keyword in the function prototype
friend Function Declarations

• Stand-alone function:
  
  ```cpp
  friend void setAVal(intVal&, int);
  // declares setAVal function to be
  // a friend of this class
  ```

• Member function of another class:
  
  ```cpp
  friend void SomeClass::setNum(int num)
  // setNum function from SomeClass
  // class is a friend of this class
  ```
friend Class Declarations

- Class as a friend of a class:
  ```cpp
class FriendClass
{
    ...
};
class NewClass
{
  public:
    friend class FriendClass; // declares
    // entire class FriendClass as a friend
    // of this class
    ...
};
```
14.3

Memberwise Assignment
Memberwise Assignment

• Can use = to assign one object to another, or to initialize an object with an object’s data

• Copies member to member.  e.g.,

\[ \text{instance2} = \text{instance1}; \]  \ means:

copy all member values from \text{instance1} and assign to the corresponding member variables of \text{instance2}

• Use at initialization:

\[ \text{Rectangle r2} = \text{r1}; \]
Program 14-5

// This program demonstrates memberwise assignment.
#include <iostream>
#include "Rectangle.h"
using namespace std;

int main()
{
    // Define two Rectangle objects.
    Rectangle box1(10.0, 10.0);  // width = 10.0, length = 10.0
    Rectangle box2(20.0, 20.0);  // width = 20.0, length = 20.0

    // Display each object's width and length.
    cout << "box1's width and length: " << box1.getWidth() << " " << box1.getLength() << endl;
    cout << "box2's width and length: " << box2.getWidth() << " " << box2.getLength() << endl;

    // Assign the members of box1 to box2.
    box2 = box1;

    // Display each object's width and length again.
    cout << "box1's width and length: " << box1.getWidth() << " " << box1.getLength() << endl;
    cout << "box2's width and length: " << box2.getWidth() << " " << box2.getLength() << endl;

    return 0;
}
Program Output

box1's width and length: 10 10
box2's width and length: 20 20

box1's width and length: 10 10
box2's width and length: 10 10
14.4

Copy Constructors
Copy Constructors

• Special constructor used when a newly created object is initialized to the data of another object of same class

• Default copy constructor copies field-to-field

• Default copy constructor works fine in many cases
Copy Constructors

Problem: what if object contains a pointer?

class SomeClass
{
    public:
        SomeClass(int val = 0)
            {value=new int; *value = val;}
        int getVal();
        void setVal(int);
    private:
        int *value;
}
Copy Constructors

What we get using memberwise copy with objects containing dynamic memory:

```cpp
SomeClass object1(5);
SomeClass object2 = object1;
object2.setVal(13);
cout << object1.getVal();  // also 13
```

![Diagram showing object copies with values](image-url)
Programmer-Defined Copy Constructor

• Allows us to solve problem with objects containing pointers:

```cpp
SomeClass::SomeClass(const SomeClass &obj) {
    value = new int;
    *value = obj.value;
}
```

• Copy constructor takes a reference parameter to an object of the class
Programmer-Defined Copy Constructor

- Each object now points to separate dynamic memory:

```cpp
SomeClass object1(5);
SomeClass object2 = object1;
object2.setVal(13);
cout << object1.getVal(); // still 5
```

![Diagram showing object1 and object2 with different values]
Programmer-Defined Copy Constructor

• Since copy constructor has a reference to the object it is copying from,
  \[
  \text{SomeClass}::\text{SomeClass}(\text{SomeClass} &\text{obj})
  \]
  it can modify that object.
• To prevent this from happening, make the object parameter \texttt{const}:
  \[
  \text{SomeClass}::\text{SomeClass}(\text{const SomeClass} &\text{obj})
  \]
Contents of StudentTestScores.h (Version 2)

1 #ifndef STUDENTTESTSCORES_H
2 #define STUDENTTESTSCORES_H
3 #include <string>
4 using namespace std;
5 
6 const double DEFAULT_SCORE = 0.0;
7 
8 class StudentTestScores
9 {
10     
11 private:
12     string studentName; // The student's name
13     double *testScores; // Points to array of test scores
14     int numTestScores; // Number of test scores
15     
16     // Private member function to create an
17     // array of test scores.
18     void createTestScoresArray(int size)
19     { numTestScores = size;
20         testScores = new double[size];
21         for (int i = 0; i < size; i++)
22             testScores[i] = DEFAULT_SCORE; }
23 
24 public:
25     // Constructor
26     StudentTestScores(string name, int numScores)
27     { studentName = name;
createTestScoresArray(numScores); }

// Copy constructor
StudentTestScores(const StudentTestScores &obj)
{ studentName = obj.studentName;
  numTestScores = obj.numTestScores;
  testScores = new double[numTestScores];
  for (int i = 0; i < numTestScores; i++)
    testScores[i] = obj.testScores[i]; }

// Destructor
~StudentTestScores()
{ delete [] testScores; }

// The setTestScore function sets a specific
// test score's value.
void setTestScore(double score, int index)
{ testScores[index] = score; }

// Set the student's name.
void setStudentName(string name)
{ studentName = name; }

// Get the student's name.
string getStudentName() const
{ return studentName; }
// Get the number of test scores.
int getNumTestScores() const
{ return numTestScores; }

// Get a specific test score.
double getTestScore(int index) const
{ return testScores[index]; }
14.5

Operator Overloading
Operator Overloading

• Operators such as =, +, and others can be redefined when used with objects of a class
• The name of the function for the overloaded operator is `operator` followed by the operator symbol, e.g., `operator+` to overload the `+` operator, and `operator=` to overload the `=` operator
• Prototype for the overloaded operator goes in the declaration of the class that is overloading it
• Overloaded operator function definition goes with other member functions
Operator Overloading

• Prototype:

```cpp
void operator=(const SomeClass &rval)
```

  return type  
  function name

  parameter for object on right side of operator

• Operator is called via object on left side
Invoking an Overloaded Operator

- Operator can be invoked as a member function:
  
  ```
  object1.operator=(object2);  
  ```

- It can also be used in more conventional manner:
  
  ```
  object1 = object2;  
  ```
Returning a Value

• Overloaded operator can return a value

```cpp
class Point2d {
   public:
      double operator-(const Point2d &right) {
         return sqrt(pow((x - right.x), 2) + pow((y - right.y), 2));
      }
   ...  
   private:
      int x, y;
};

Point2d point1(2, 2), point2(4, 4);
// Compute and display distance between 2 points.
cout << point2 - point1 << endl; // displays 2.82843
```
Returning a Value

• Return type the same as the left operand supports notation like:
  
  object1 = object2 = object3;

• Function declared as follows:
  
  const SomeClass operator=(const SomeClass &rval)

• In function, include as last statement:
  
  return *this;
The **this** Pointer

- **this**: predefined pointer available to a class’s member functions
- Always points to the instance (object) of the class whose function is being called
- Is passed as a hidden argument to all non-static member functions
- Can be used to access members that may be hidden by parameters with same name
this Pointer Example

class SomeClass
{
    private:
        int num;
    public:
        void setNum(int num)
        {
            this->num = num;
        }

    ...
};
Notes on Overloaded Operators

• Can change meaning of an operator
• Cannot change the number of operands of the operator
• Only certain operators can be overloaded. Cannot overload the following operators:
  
  ?:  .  .*  ::  sizeof
Overloading Types of Operators

• `++`, `--` operators overloaded differently for prefix notation (in right-to-left) vs. postfix notation (in left-to-right)
• Overloaded relational operators should return a `bool` value
• Overloaded stream operators `>>`, `<<` must return reference to `istream`, `ostream` objects and take `istream`, `ostream` objects as parameters
Overloaded \[\] Operator

- Can create classes that behave like arrays, provide bounds-checking on subscripts
- Must consider constructor, destructor
- Overloaded \[\] returns a reference to object, not an object itself
14.6

Object Conversion
Object Conversion

• Type of an object can be converted to another type
• Automatically done for built-in data types
• Must write an operator function to perform conversion
• To convert an `FeetInches` object to an `int`:
  ```
  FeetInches::operator int() 
  {return feet;}
  ```
• Assuming distance is a `FeetInches` object, allows statements like:
  ```
  int d = distance;
  ```
14.7

Aggregation
Aggregation

• **Aggregation**: a class is a member of a class

• Supports the modeling of ‘has a’ relationship between classes – enclosing class ‘has a’ enclosed class

• Same notation as for structures within structures
Aggregation

class StudentInfo
{
    private:
        string firstName, LastName;
        string address, city, state, zip;
    ...
};
class Student
{
    private:
        StudentInfo personalData;
    ...
};
See the Instructor, TextBook, and Course classes in Chapter 14.