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
static member variable

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  // not a static member
13    { return objectCount; }
14  };
15
16  // Definition of the static member variable, written
17  // outside the class.
18  int Tree::objectCount = 0;  // Static member defined here.
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:

friend void setAVal(intVal&, int);
// declares setAVal function to be
// a friend of this class

• Member function of another class:

friend void SomeClass::setNum(int num)
// setNum function from SomeClass
// class is a friend of this class
class FriendClass
{

...;
};
class NewClass
{

public:

friend class FriendClass; // declares
// entire class FriendClass as a friend
// of this class

...;

};
Programs 14-2 to 14-4

- **Budget Version 1** 14-2
  - corporate budget
  - division budget

- **Budget Version 2** 14-3
  - adds main office

- **Budget Version 3** 14-4
  - adds friend function
  - adds AuxiliarryOffice 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.*,
  
  ```
  instance2 = instance1;
  ```

  means:
  
  copy all member values from `instance1` and assign to the corresponding member variables of `instance2`

- Use at *initialization*:
  
  ```
  Rectangle r2 = r1;
  ```
// 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 << 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 14-5 (continued)

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:

//shallow copy

SomeClass object1(5);
SomeClass object2 = object1;
object2.setVal(13);
cout << object1.getVal(); // also 13
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
```
Programmer-Defined Copy Constructor

- Since copy constructor has a reference to the object it is copying from,
  
  ```cpp
  SomeClass::SomeClass(SomeClass &obj)
  ```

  it can modify that object.

- To prevent this from happening, make the object parameter const:
  
  ```cpp
  SomeClass::SomeClass(const SomeClass &obj)
  ```
StudentTestScores.h Version 1

• Problem:
  – one of its members has a pointer and the class does not have a copy constructor
  – only does a "shallow copy"
  – this header file is not part of the presentation
#ifndef STUDENTTESTSCORES_H
#define STUDENTTESTSCORES_H
#include <string>
using namespace std;

const double DEFAULT_SCORE = 0.0;

class StudentTestScores
{
    private:
        string studentName;  // The student's name
        double *testScores;  // Points to array of test scores
        int numTestScores;   // Number of test scores

        // Private member function to create an
        // array of test scores.
        void createTestScoresArray(int size)
        {
            numTestScores = size;
            testScores = new double[size];
            for (int i = 0; i < size; i++)
                testScores[i] = DEFAULT_SCORE;
        }

    public:
        // Constructor
        StudentTestScores(string name, int numScores)
        {
            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;
    }

Copyright © 2012 Pearson Education, Inc.
// Get the number of test scores.
int getNumTestScores() const
{ return numTestScores; }

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

#endif
14.5

Operator Overloading
Types of Overloaded Operators

- Overloaded operators may be:
  - member functions (1 argument)
    - ( ), [ ], -> (required)
    - any assignment operator (required)
  - global functions (2 arguments)
    - stream operators >> <<
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:

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

- Operator is called via object on left side

- Parameter for object on right side of operator
Invoking an Overloaded Operator

• Operator can be invoked as a member function:
  
  \texttt{object1.operator\texttt{\=}}(\texttt{object2});

• It can also be used in more conventional manner:
  
  \texttt{object1 = object2};
StudentTestScores Version 3

- uses an overloaded $=$ operator
- Program 14-6
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
```
Assignment Operator Returning a Value

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

• Function declared as follows:
  
  \texttt{const SomeClass \texttt{operator=}=(const someClass \&rval)}

• In function, include as last statement:
  
  \texttt{return *this; //copy of the object}
StudentTestScores Version 4

- overloaded = operator returns a constant StudentTestScores object
- uses *this to refer to the object being returned ( returns itself )

- Program 14-7
- More on this pointer follows on next slide
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  //implicit parameter
- Can be used to access members that may be hidden by parameters with same name (shadowing)
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

- Overloaded mathematical operators should return a numeric value
- `++`, `--` operators overloaded differently for prefix vs. postfix notation
- Overloaded relational operators should return a `bool` value
- Overloaded stream operators `>>, <<` must return a reference to `istream / ostream` objects and take `istream / ostream` objects as parameters
• FeetInches.h Version 1
  – overloads + and – operators using member functions
  – The operators return a new FeetInches object
  – Program 14-8
Overloading Pre-Test ++ Operator

• Using FeetInches class

FeetInches FeetInches::operator++()
{
    ++inches;
    simplify();
    return *this
}
Overloading Post-Test ++ Operator

• Using **FeetInches** class

```cpp
FeetInches FeetInches::operator++(int)
{
    FeetInches temp(feet, inches)
    += inches; // or inches++
    simplify();
    return temp
}
```

nameless integer parameter
• **FeetInches.h Version 2**
  – overloads *pre-test* and *post-test* `++` operator
  – Program 14-9
  – The operators return a new `FeetInches` object
bool FeetInches::operator > (const FeetInches &right) 
{
    bool status;

    if (feet > right.feet)
        status = true;
    else
        if (feet == right.feet && inches > right.inches)
            status = true;
        else
            status = false;

    return status;
}
Overloading  operator <

bool FeetInches::operator < (const FeetInches &right)  
{
    bool status;

    if (feet < right.feet)
        status = true;
    else
        if (feet == right.feet && inches < right.inches)
            status = true;
        else
            status = false;

    return status;
}
Overloading operator ==

```cpp
bool FeetInches::operator == (const FeetInches &right) {
    bool status;

    if (feet == right.feet && inches == right.inches)
        status = true;
    else
        status = false;

    return status;
}
```
- FeetInches.h Version 3
  - overloads `<` `>` `==` operators
  - Program 14-10
  - The operators return a `bool` value
Overloading \texttt{operator >>}

Some compilers require you to prototype the stream operator functions outside the class. These declarations have been added for this reason.

```cpp
class FeetInches;  //forward declaration

//Function prototypes for overloaded stream operators
istream &operator >> (istream &, FeetInches &)

istream &operator >> (istream &stream, FeetInches &obj)
{
    //prompt the user for the feet
    cout << "Feet: ";
    stream >> obj.feet;

    //prompt the user for the inches
    cout << "Inches: ";
    stream >> obj.inches;

    obj.simplify();

    return stream;
}
```
Overloading \texttt{operator $\ll$}

Some compilers require you to prototype the stream operator functions outside the class. These declarations have been added for this reason.

\begin{verbatim}
class FeetInches;  //forward declaration

//Function prototypes for overloaded stream operators
ostream &operator $\ll$ (ostream &, const FeetInches &)

ostream &operator $\ll$ (ostream &stream, const FeetInches &obj)
{
    stream $\ll$ obj.feet
    $\ll$ " feet, "
    $\ll$ obj.inches
    $\ll$ " inches";

    return stream;
}
\end{verbatim}
• FeetInches.h Version 4
  - overloads <<   >> operators
  - Program 14-11
  - The operators return a reference to a stream object (istream / ostream)
  - The operators are defined outside of the class. friend declarations are used
Overloaded [ ] Operator

- Can create classes that behave like arrays, provide bounds-checking on subscripts
- Must consider constructor, destructor
- Overloaded [ ] returns a reference to an object, not an object itself
- [ ] dereference an array element
• **IntArray.h**
  - overloads \([\ ]\) operator
  - **Program 14-12**
  - **Program 14-13**
    - demonstrates use of an invalid subscript
  - The operator returns a *reference to an integer* (or exits if an invalid subscript is used)
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 a FeetInches object to an int:

```cpp
FeetInches::operator int()  //cast FeetInches object to an int
{return feet;}  //truncates remainder (inches)
```

Note: no return type is specified in the function header for a conversion function. Also a conversion function does not take any arguments.

• Assuming distance is a FeetInches object, allows statements like:

```cpp
int d = distance;
```
• FeetInches.h Version 5
  – provides conversion operators
    • int () //conversion to int
    • double () //conversion to double
  – Program 14-14
  – operators return a FeetInches object converted to an int or double
14.7

Aggregation
Aggregation (aka composition)

- **Aggregation:**
  - a class is a member of a different 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.
• instructor.h
• course.h
• textbook.h
• program 14-15

• Stock.h
• StockPurchase.h
• program 14-16
Class Responsibilities

• The things that the class is responsible for knowing (attributes)

• The things that the class is responsible for doing (behaviors)
CRC Cards

- C – class
- R – responsibilities
- C – collaborations

<table>
<thead>
<tr>
<th>StockPurchase</th>
</tr>
</thead>
<tbody>
<tr>
<td>know the stock to purchase</td>
</tr>
<tr>
<td>Stock class</td>
</tr>
<tr>
<td>know the number of shares to purchase</td>
</tr>
<tr>
<td>None</td>
</tr>
<tr>
<td>Calculate the cost of the purchase</td>
</tr>
<tr>
<td>Stock class</td>
</tr>
</tbody>
</table>