CSCE 121
Introduction to Program Design & Concepts
Classes and the Rule of 3

Dr. Tim McGuire

Grateful acknowledgment to Dr. Philip Ritchey and Dr. Michael Moore for some of the material on which these slides are based.

The Rule of 3

Classes have three special member functions that are commonly implemented together:

- **Destructor:** A destructor is a class member function that is automatically called when an object of the class is destroyed, such as when the object goes out of scope or is explicitly destroyed as in **`delete someObject;`**.
- **Copy constructor:** A copy constructor is another version of a constructor that can be called with a single pass by reference argument. The copy constructor is automatically called when an object is passed by value to a function, such as for the function
- **Copy assignment operator:** The assignment operator "=" can be overloaded for a class via a member function, known as the copy assignment operator, that overloads the built-in function "operator=", the member function having a reference parameter of the class type and returning a reference to the class type.

The Rule of Three

- The ***rule of three*** describes a practice that if a programmer explicitly defines any one of those three special member functions (destructor, copy constructor, copy assignment operator), then the programmer should explicitly define all three.
- For this reason, those three special member functions are sometimes called ***the big three***.

Destructors

- Member function automatically called when an object is destroyed
- Destructor name is ~classname, *e.g.*, **`~Rectangle`**
- Has no return type; takes no arguments
- Only one destructor per class, *i.e.*, it cannot be overloaded
- If constructor allocates dynamic memory, destructor should release it

- See destructor.cpp

Constructors, Destructors, and Dynamically Allocated Objects

- When an object is dynamically allocated with the **new** operator, its constructor executes:

```
Rectangle *r = new Rectangle(10, 20);
```

- When the object is destroyed, its destructor executes:

```
delete r;
```

- See ContactInfo.h and ContactInfo.cpp

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`

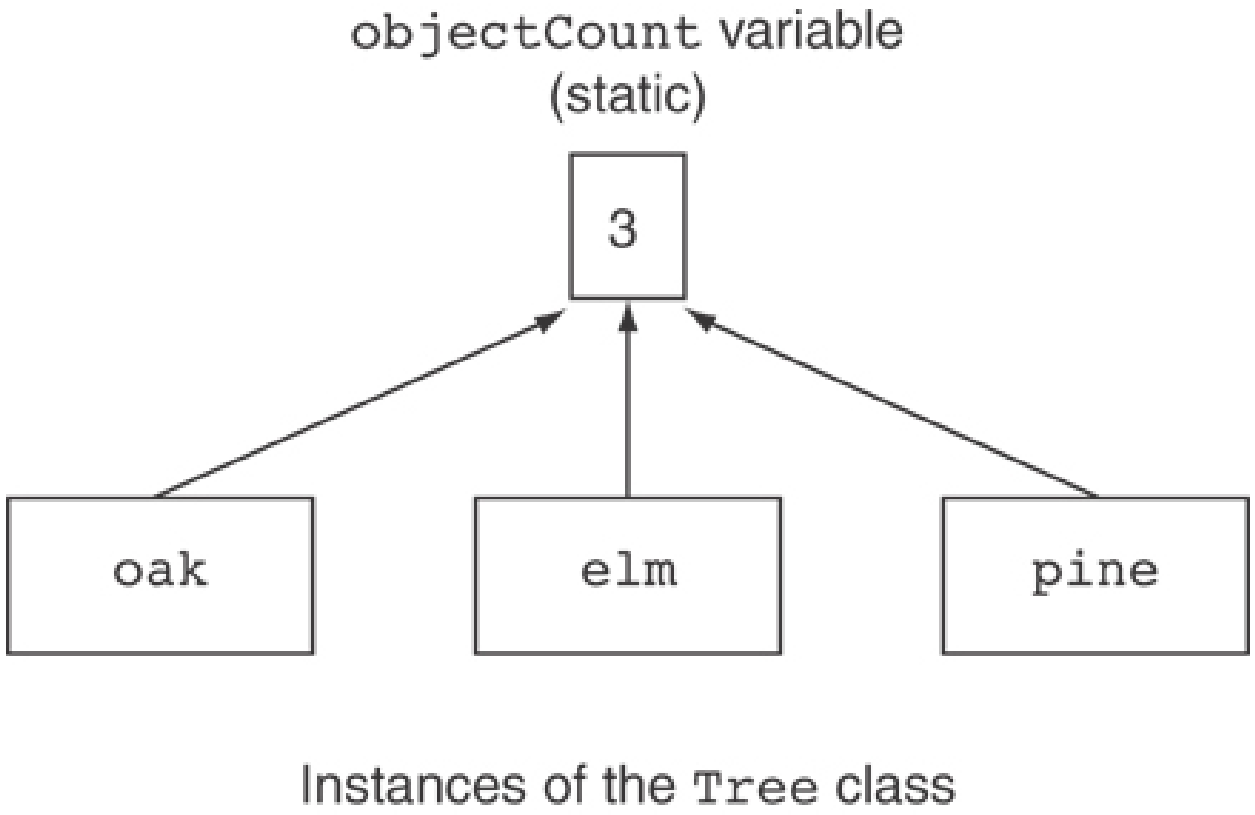
Static member declared here.

```
 1  // Tree class
 2  class Tree
 3  {
 4  private:
 5     static int objectCount = 0;     // 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;
```

Static member defined here.

- See tree.cpp

Three Instances of the Tree Class, But Only One `objectCount` Variable



`static` member function

- Can declare static functions as well as static variables
- Declared with `static` before return type:

```
static int getObjectCount() const
{ return objectCount; }
```

- Static member functions can only access static member data
- Can be called independent of objects:

```
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";
```

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

Why friends?

- Class operations are typically implemented as member functions

- Some operations are better implemented as ordinary (nonmember) functions

Example: DayMonth Class

```
Class DayMonth {
private:
        int month;
        int day;
public:
        int getMonth();
        int getDay();
        setMonth();
        setDay();
…. etc….
}
```

Program Example: An Equality Function

- The DayMonth class can be enhanced to include an equality function
  - An equality function tests two objects of type DayMonth to see if their values represent the same date
  - Two dates are equal if they represent the same day and month
- We want the equality function to return a value of type bool that is true if the dates are the same
- The equality function requires a parameter for each of the two dates to compare
- The declaration is

  bool equal(DayMonth date1, DayMonth date2);

  - Notice that equal is not a member of the class DayMonth

Defining Function equal

- The function equal, is not a member function
  - It must use public accessor functions to obtain the day and month from a DayMonth object
- equal can be defined in this way:

```
bool equal(DayMonth date1, DayMonth date2)
 {
   return ( date1.getMonth( ) == date2.getMonth( )
              &&
            date1.getDay( ) == date2.getDay( ) );
 }
```

Using The Function equal

- The equal function can be used to compare dates in this manner

```
if ( equal( today,  bach_birthday) )
    cout << "It's Bach's birthday!";
```

Is equal Efficient?

- Function equal could be made more efficient
  - Equal uses member function calls to obtain the private data values
  - Direct access of the member variables would be more efficient (faster)

A More Efficient equal

- As defined here, equal is more efficient, but not legal

```
bool equal(DayMonth date1, DayMonth date2)
{
        return (date1.month = = date2.month
                &&
                date1.day = = date2.day );
}
```

  - The code is simpler and more efficient
  - Direct access of private member variables is not legal!

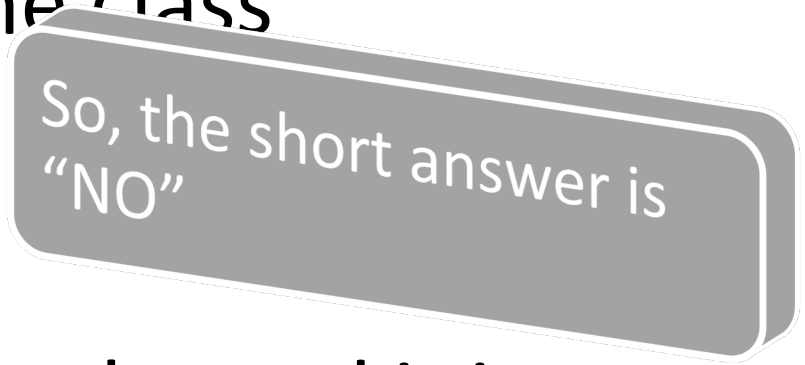
Declaring A Friend

- The function equal is declared a friend in the abbreviated class definition here

```
class DayMonth
 {
     public:
        friend bool equal(DayMonth date1, DayMonth date2);
        // The rest of the public members
    private:
        //  the private members
};
```

Are Friends Needed?

- Friend functions can be written as non-friend functions using the normal accessor and mutator functions that should be part of the class



- The code of a friend function is simpler and it is more efficient

Choosing Friends

- How do you know when a function should be a friend or a member function?
  - In general, use a member function if the task performed by the function involves only one object
  - In general, use a nonmember function if the task performed by the function involves more than one object
    - Choosing to make the nonmember function a friend is a decision of efficiency and personal taste

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

Why Copy Constructors?

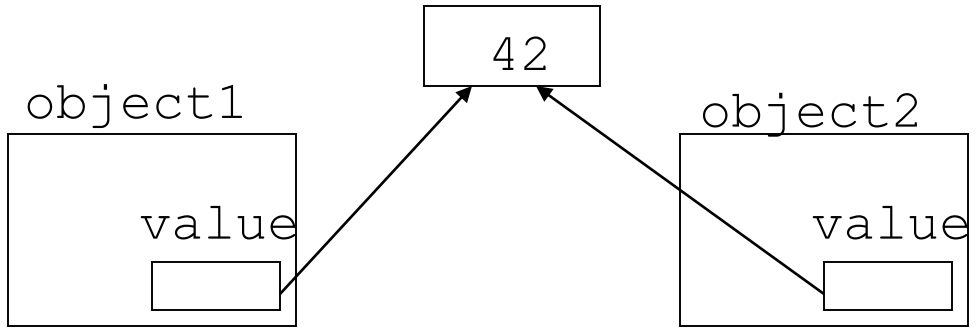
Problem: what if an object contains a pointer?

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

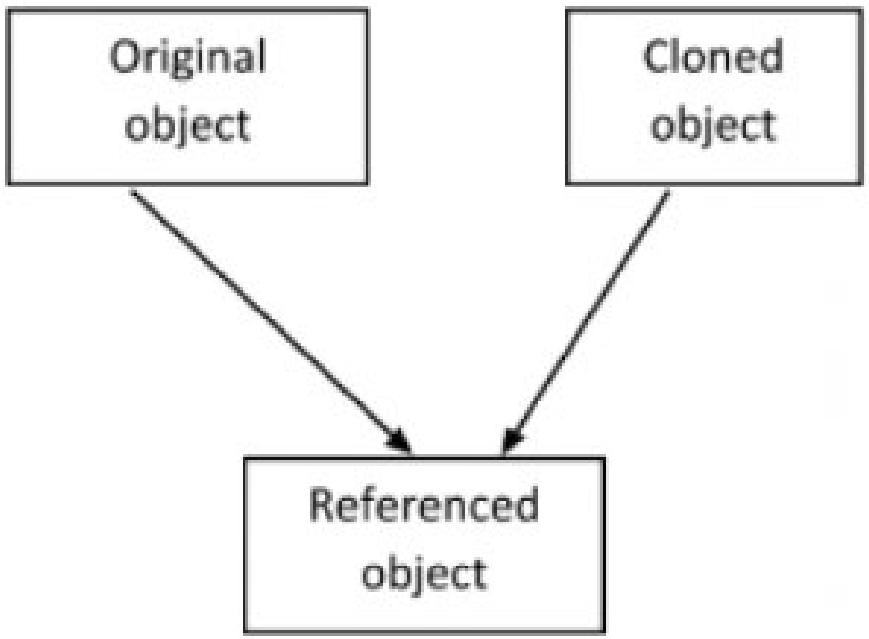
Copy Constructors

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

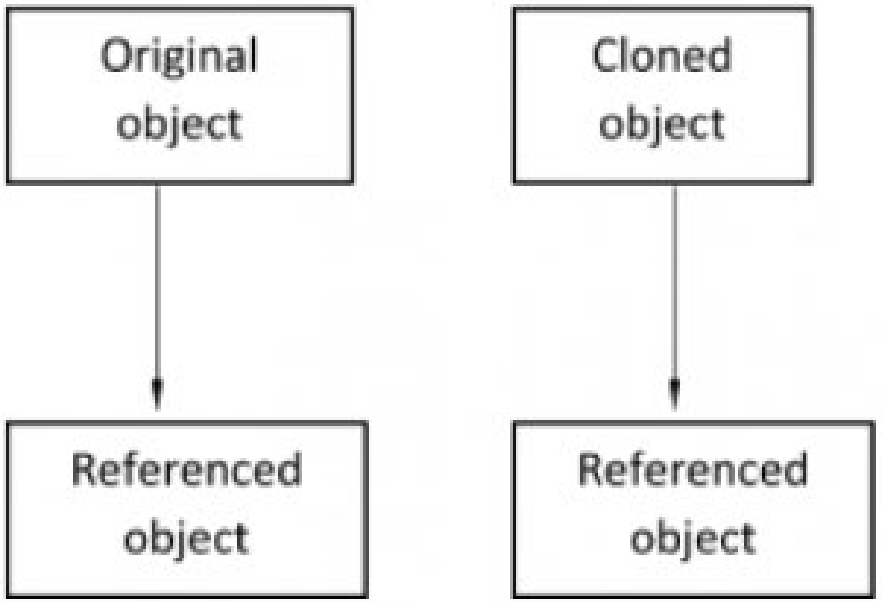
```
SomeClass object1(5);
SomeClass object2 = object1;
object2.setVal(42);
cout << object1.getVal(); // also 42
```



Shallow copy



Deep copy



Copy Constructors

- The problem with using call-by-value parameters with pointer variables is solved by the copy constructor.
- A copy constructor is a constructor with one parameter of the same type as the class
  - The parameter is a call-by-reference parameter
  - The parameter is usually a constant parameter
  - The constructor creates a complete, independent copy of its argument

Programmer-Defined Copy Constructor

- Allows us to solve problem with objects containing pointers:

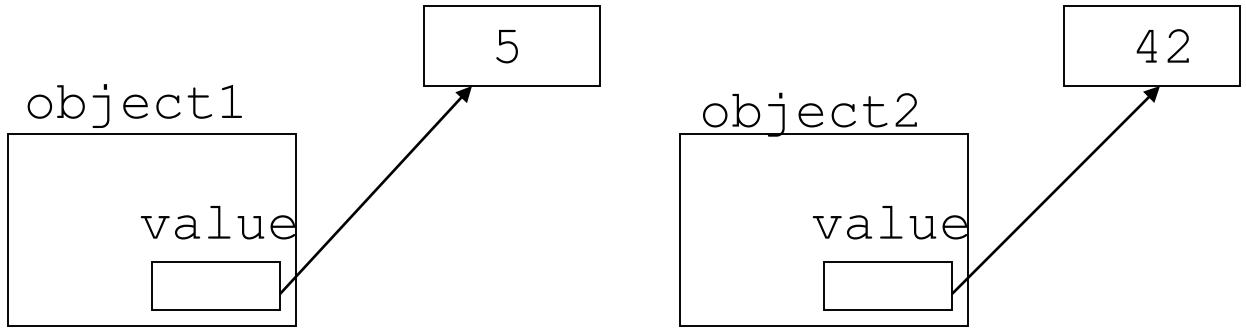
```
SomeClass::SomeClass(const SomeClass &obj)
{
    value = new int;
    *value = obj.value;
}
```

- Copy constructor takes a reference parameter to an object of the class

Programmer-Defined Copy Constructor

- Each object now points to separate dynamic memory:

```
SomeClass object1(5);
SomeClass object2 = new SomeClass(object1);
object2.setVal(42);
cout << object1.getVal(); // still 5
```



Programmer-Defined Copy Constructor

- Since copy constructor has a reference to the object it is copying from,

```
SomeClass::SomeClass(SomeClass &obj)
```

  it can modify that object.

- To prevent this from happening, make the object parameter `const`:

```
SomeClass::SomeClass(const SomeClass &obj)
```

When To Include a Copy Constructor

- When a class definition involves pointers and dynamically allocated memory using "new", include a copy constructor
- Classes that do not involve pointers and dynamically allocated memory do not need copy constructors

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

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

The **`this`** Pointer

- Example, `student1` and `student2` are both `StudentTestScores` objects.

- The following statement causes the `getStudentName` member function to operate on `student1`:

```
cout << student1.getStudentName() << endl;
```

- When `getStudentName` is operating on `student1`, the `this` pointer is pointing to `student1`.

The **`this`** Pointer

- Likewise, the following statement causes the `getStudentName` member function to operate on `student2`:

```
cout << student2.getStudentName() << endl;
```

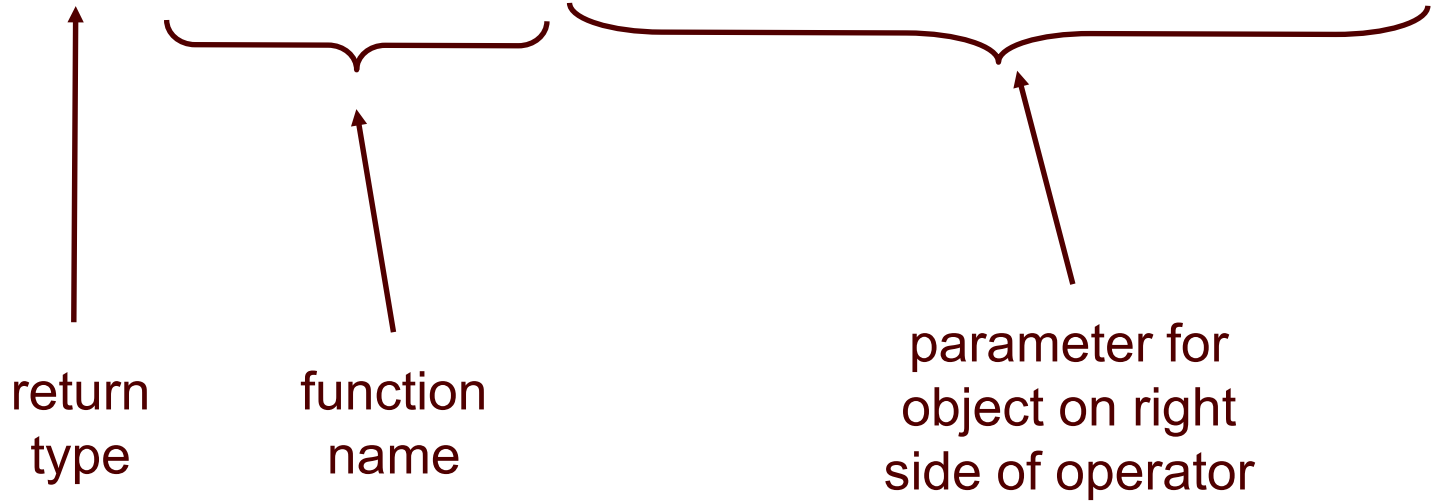
- When `getStudentName` is operating on `student2`, the `this` pointer is pointing to `student2`.

- The `this` pointer always points to the object that is being used to call the member function.

Operator Overloading

- Prototype:

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



- Operator is called via object on left side

Invoking an Overloaded Operator

- Operator can be invoked as a member function:

```
object1.operator=(object2);
```

- But if we did it that way, we might as well have just called it “assign” and invoke it this way:

```
object1.assign(object2);
```

- Instead, we can can use it this way:

```
object1 = object2;
```

Returning a Value

- An overloaded operator can return a value

```
class Point2d
{
private:
    int x, y;

...
public:
  double operator-(const point2d &right)
  { return sqrt(pow((this.x-right.x),2) + pow((this.y-right.y),2)); }
};
Point2d point1(2,2), point2(4,4);
// Compute and display distance between 2 points.
cout << point2 – point1 << endl; // displays 2.82843
```

We could simply say **`x-right.x`** here, but the use of **`this`** reminds us that **`x`** belongs to the calling object.

The Big Three

- So, now we have seen
  - Destructors
  - Copy Constructors
  - Assignment Operators
- If you need to define one, you need to define all

Notes on Overloaded Operators

- Can change meaning of an operator
- Cannot change the number of operands of the operator
- Overloaded relational operators should return a `bool` value
- Overloaded stream operators >>, << must return reference to `istream`, `ostream` objects

```
1  #ifndef FEETINCHES_H
2  #define FEETINCHES_H
3
4  // The FeetInches class holds distances or measurements
5  // expressed in feet and inches.
6
7  class FeetInches
8  {
9  private:
10    int feet;        // To hold a number of feet
11    int inches;      // To hold a number of inches
12    void simplify(); // Helper function efined in FeetInches.cpp
13 public:
14    // Constructor
15    FeetInches(int f = 0, int i = 0)
16       { feet = f; inches = i; simplify(); }
17
18    // Mutator functions
19    void setFeet(int f)
20       { feet = f; }
21
22    void setInches(int i)
23       { inches = i; simplify(); }
24
25    // Accessor functions
26    int getFeet() const
27       { return feet; }
28
29    int getInches() const
30       { return inches; }
31
32 };
33
34 #endif
```

Modified from an example by Tony Gaddis

```cpp
// Implementation file for the FeetInches class
#include <cstdlib>       // Needed for abs()
#include "FeetInches.h"

//*************************************************************************************
// Definition of member function simplify. This function hecks for values in the inches member greater than   *
// twelve or less than zero. If such a value is found, the numbers in feet and inches are adjusted to conform *
// to a standard feet & inches expression. For example,                                                        *
// 3 feet 14 inches would be adjusted to 4 feet 2 inches and                                                   *
// 5 feet -2 inches would be adjusted to 4 feet 10 inches.                                                     *
//*************************************************************************************

void FeetInches::simplify()
{
   if (inches >= 12)
   {
      feet += (inches / 12);
      inches = inches % 12;
   }
   else if (inches < 0)
   {
      feet -= ((abs(inches) / 12) + 1);
      inches = 12 - (abs(inches) % 12);
   }
}
```

```
int main()
{
   int feet, inches;  // To hold input for feet and inches

   // Create three FeetInches objects. The default arguments
   // for the constructor will be used.
   FeetInches first, second, third;

   // Get a distance from the user.
   cout << "Enter a distance in feet and inches: ";
   cin >> feet >> inches;

   // Store the distance in the first object.
   first.setFeet(feet);
   first.setInches(inches);

   // Get another distance from the user.
   cout << "Enter another distance in feet and inches: ";
   cin >> feet >> inches;

   // Assign first + second to third.
   third.setFeet(first.getFeet() + second.getFeet());
   third.setInches(first.getInches() + second.getInches());

   // Display the result.
   cout << "first + second = ";
   cout << third.getFeet() << " feet, ";
   cout << third.getInches() << " inches.\n";


   return 0;
}
```

```
7  class FeetInches
8  {
9  private:
10    int feet;        // To hold a number of feet
11    int inches;      // To hold a number of inches
12    void simplify(); // Defined in FeetInches.cpp
13 public:
14    // Constructor
15    FeetInches(int f = 0, int i = 0)
16       { feet = f; inches = i; simplify(); }
17
18    // Mutator functions
19    void setFeet(int f)
20       { feet = f; }
21
22    void setInches(int i)
23       { inches = i;   simplify(); }
24
25    // Accessor functions
26    int getFeet() c
27       { return fee
28
29    int getInches()
30       { return inc
31
32    // Overloaded operator functions
33    FeetInches operator + (const FeetInches &); // Overloaded +
34    FeetInches operator - (const FeetInches &); // Overloaded -
35 };
36
```

```
// Overloaded operator functions
FeetInches operator + (const FeetInches &); // Overloaded +
FeetInches operator - (const FeetInches &); // Overloaded -
```

```
28
29  //*************************************************
30  // Overloaded binary + operator.                  *
31  //*************************************************
32
33  FeetInches FeetInches::operator + (const FeetInches &right)
34  {
35     FeetInches temp;
36
37     temp.inches = inches + right.inches;
38     temp.feet = feet + right.feet;
39     temp.simplify();
40     return temp;
41  }
42
43     temp.inches = inches - right.inches;
44     temp.feet = feet - right.feet;
45     temp.simplify();
46     return temp;
47  }
```

```
31  // Assign first + second to third.
32  third = first + second;
33
34
35  // Assign first + second to third.
36  third = first.operator+(second);
37
38
39  // Assign first - second to third.
40  third = first - second;
41
42  // Display the result.
43  cout << "first - second = ";
44  cout << third.getFeet() << " feet, ";
45  cout << third.getInches() << " inches.\n";
46
```

```
// Assign first + second to third.
third.setFeet(first.getFeet() + second.getFeet());
third.setInches(first.getInches() + second.getInches());
```