Chapter 14:

More About Classes

Copy Constructors

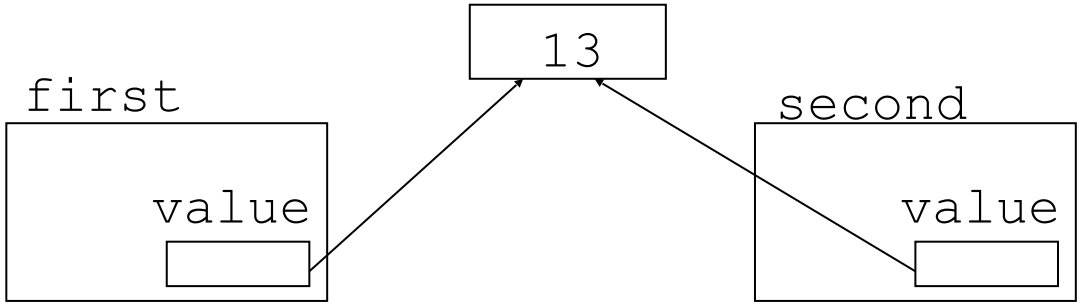
Problem: what if object contains a pointer?

```
class SomeClass
{
private:
     int *value;

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

Copy Constructors

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




```
SomeClass first(5);
// the value instance variable of both
// objects have the same address
SomeClass second = first;
// if we now change the content of the
// value variable by the second object
second.setVal(13);
// that will also impact the content of the
// value variable of the first object
cout << first.getVal(); // also 13
```

Programmer-Defined Copy Constructor

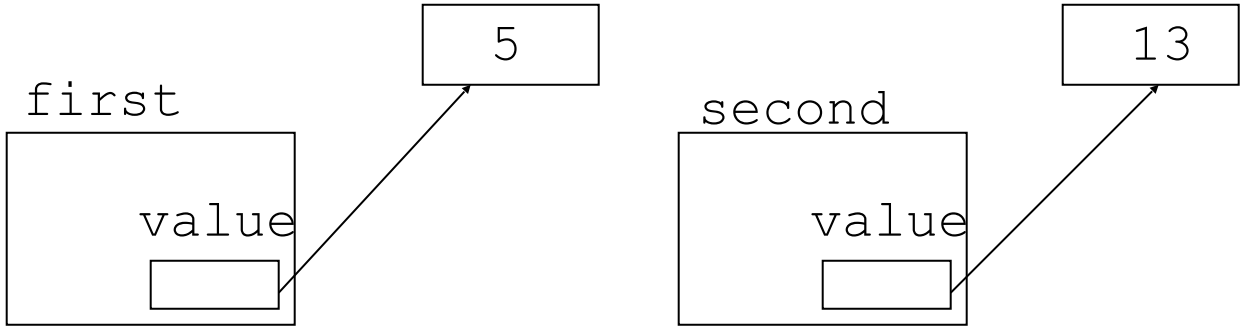
- Allows us to solve problem with objects containing pointers:

```
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 first(5);
// the value instance variable of both
// objects have the same address
SomeClass second = first;
// if we now change the content of the
// value variable by the second object
second.setVal(13);
// that will also impact the content of the
// value variable of the first object
cout << first.getVal(); // still 5
```

Programmer-Defined Copy Constructor

- Since copy constructor has a reference to the object it is copying from, it can modify that object. For example,

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

- Therefore, to prevent this from happening, we make the object parameter `const`:

```
SomeClass::SomeClass(const SomeClass &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 private:
11    string studentName;  // The student's name
12    double *testScores;  // Points to array of test scores
13    int numTestScores;   // Number of test scores
14
15     // Private member function to create an
16     // array of test scores.
17    void createTestScoresArray(int size)
18    { numTestScores = size;
19      testScores = new double[size];
20      for (int i = 0; i < size; i++)
21         testScores[i] = DEFAULT_SCORE; }
22
23 public:
24     // Constructor
25    StudentTestScores(string name, int numScores)
26    { studentName = name;
```

```
27      createTestScoresArray(numScores); }
28
29    // Copy constructor
30    StudentTestScores(const StudentTestScores &obj)
31    { studentName = obj.studentName;
32      numTestScores = obj.numTestScores;
33      testScores = new double[numTestScores];
34      for (int i = 0; i < numTestScores; i++)
35         testScores[i] = obj.testScores[i]; }
36
37     // Destructor
38    ~StudentTestScores()
39    { delete [] testScores; }
40
41     // The setTestScore function sets a specific
42     // test score's value.
43    void setTestScore(double score, int index)
44    { testScores[index] = score; }
45
46     // Set the student's name.
47    void setStudentName(string name)
48    { studentName = name; }
49
50     // Get the student's name.
51    string getStudentName() const
52    { return studentName; }
```

```
53
54     // Get the number of test scores.
55     int getNumTestScores() const
56     { return numTestScores; }
57
58      // Get a specific test score.
59     double getTestScore(int index) const
60     { return testScores[index]; }
61 };
62 #endif
```

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

```
class Point{
     private:
          int x, y;
     public:
          Point(int xp, int yp)
          {
               x = xp;
               y = yp;
          }
          void display()
          {
               cout<< this->x << ":" << this->y <<endl;

          }
};

int main() {
     Point pt_right (50, 30), pt_left (10, 50);
     pt_left.display();
     pt_right.display();

     return 0;
}
```

The `this` Pointer

- Example, `pt_left` and `pt_right` are both `Point` objects.
- The following statement causes the `display` member function to operate on `pt_left`:

```
pt_left.display();
```

- When `display` is operating on `pt_left`, the `this` pointer is pointing to `pt_left`.

The `this` Pointer

- Likewise, the following statement causes the `display` member function to operate on `pt_right`:

```
pt_right.display();
```

- When `display` is operating on `pt_right`, the `this` pointer is pointing to `pt_right`.

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

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

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:

  scope operator ::
  sizeof
  member selector .
  member pointer selector *
  ternary operator ?:

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);
```

- It can also be used in more conventional manner:

```
object1 = object2;
```

Invoking an Overloaded Operator

- Review the attached example, over_asgn.cpp