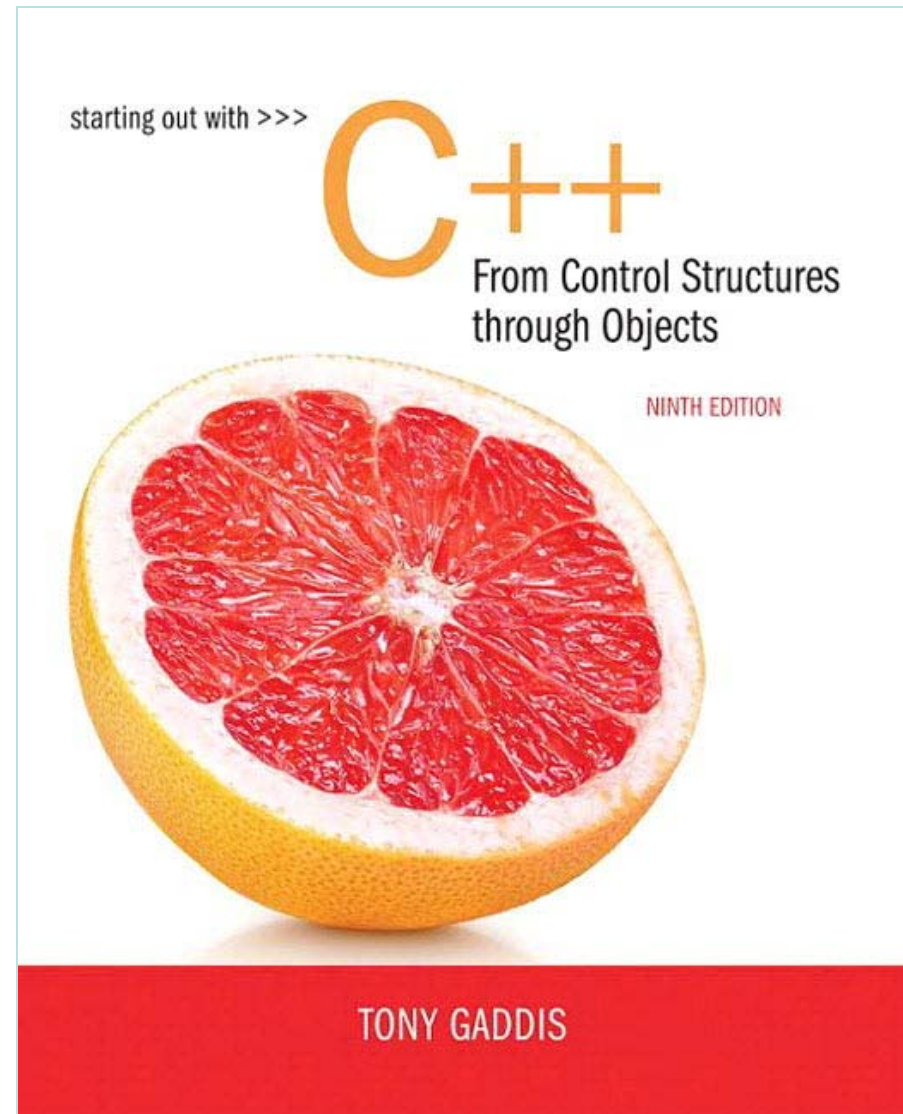


Chapter 14:

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
13         { return objectCount; }
14 };
15
16 // Definition of the static member variable, written
17 // outside the class.
18 int Tree::objectCount = 0;
```

Static member declared here.

Static member defined here.

Program 14-1

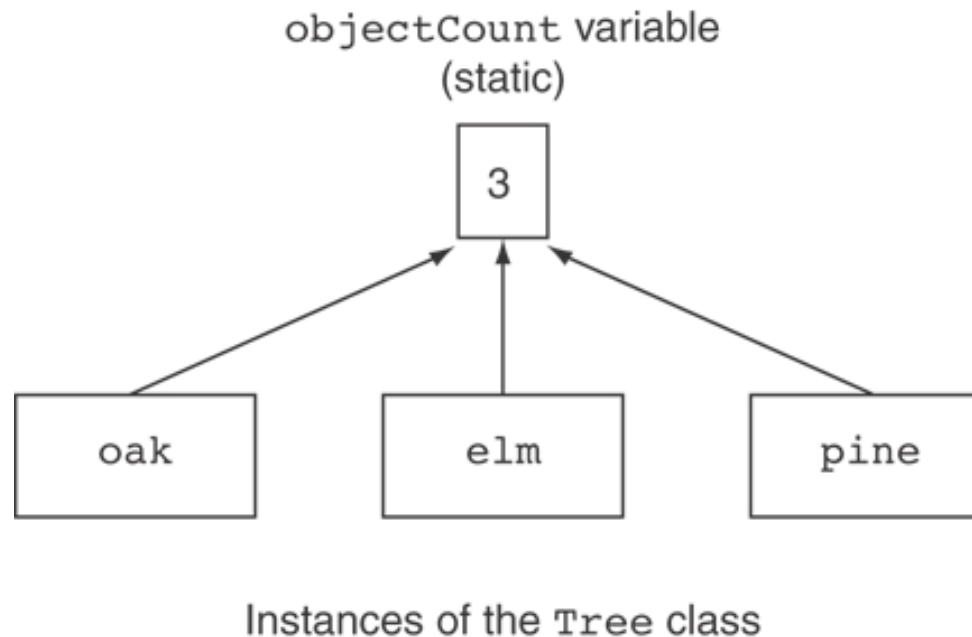
```
1  // This program demonstrates a static member variable.
2  #include <iostream>
3  #include "Tree.h"
4  using namespace std;
5
6  int main()
7  {
8      // Define three Tree objects.
9      Tree oak;
10     Tree elm;
11     Tree pine;
12
13     // Display the number of Tree objects we have.
14     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:

```
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";
```




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
```

friend Class Declarations

● Class as a friend of a class:

```
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.*,
`instance2 = instance1;` means:
copy all member values from `instance1` and assign to the corresponding member variables of `instance2`
- Use at initialization:
`Rectangle r2 = r1;`

Program 14-5

```
1 // This program demonstrates memberwise assignment.
2 #include <iostream>
3 #include "Rectangle.h"
4 using namespace std;
5
6 int main()
7 {
8     // Define two Rectangle objects.
9     Rectangle box1(10.0, 10.0);    // width = 10.0, length = 10.0
10    Rectangle box2 (20.0, 20.0);    // width = 20.0, length = 20.0
11
12    // Display each object's width and length.
13    cout << "box1's width and length: " << box1.getWidth()
14         << " " << box1.getLength() << endl;
15    cout << "box2's width and length: " << box2.getWidth()
16         << " " << box2.getLength() << endl << endl;
17
18    // Assign the members of box1 to box2.
19    box2 = box1;
20
21    // Display each object's width and length again.
22    cout << "box1's width and length: " << box1.getWidth()
23         << " " << box1.getLength() << endl;
24    cout << "box2's width and length: " << box2.getWidth()
25         << " " << box2.getLength() << endl;
26
27    return 0;
28 }
```



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

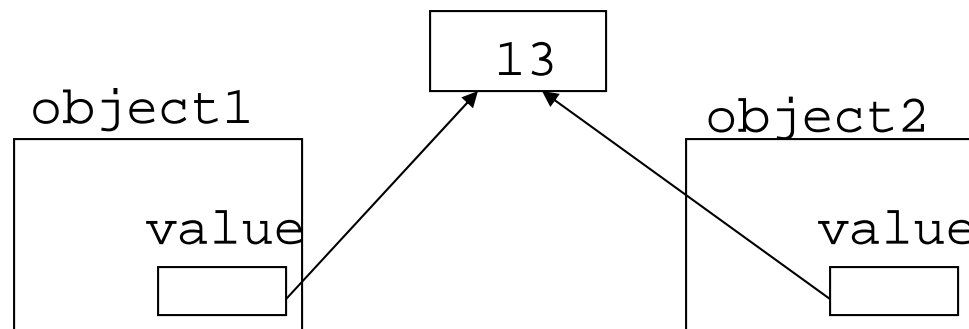
Problem: what if object contains a pointer?

```
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:

```
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:

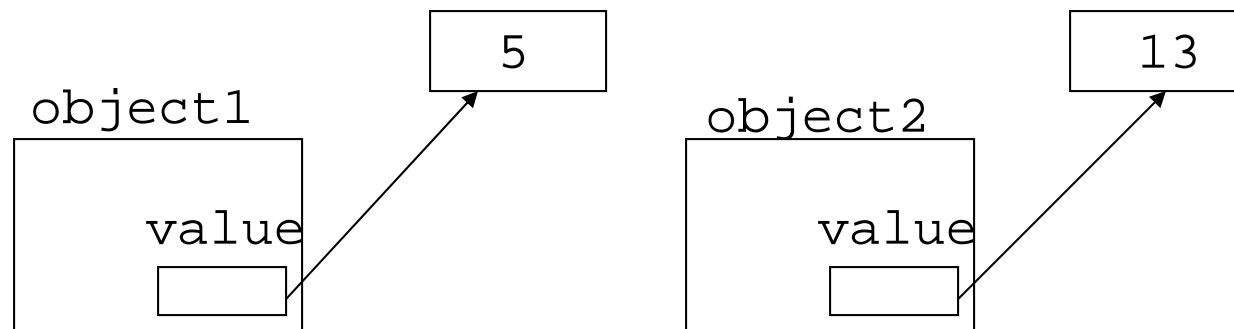
```
SomeClass::SomeClass(const SomeClass &obj)
{
    value = new int;
    *value = obj.value;
}
```

- 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 = 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,

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

it can modify that object.
- To prevent this from happening, 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;
```



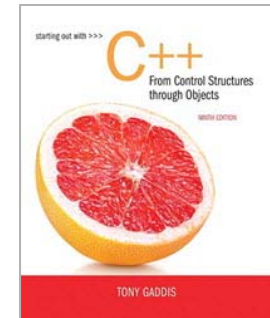

```

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
```



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

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`:

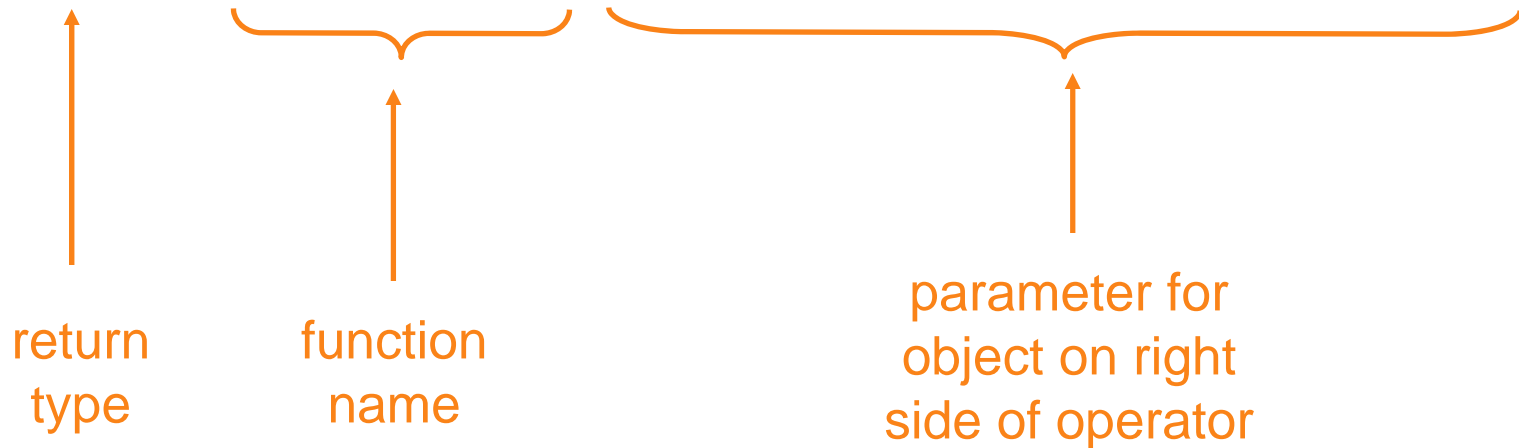
```
cout << student2.getStudentName() << endl;
```

- 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);
```

- It can also be used in more conventional manner:

```
object1 = object2;
```

Returning a Value

Overloaded operator can return a value

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

    ...
public:
    double operator-(const point2d &right)
    { return sqrt(pow((x-right.x),2)
                  + pow((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
```

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;
```

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:

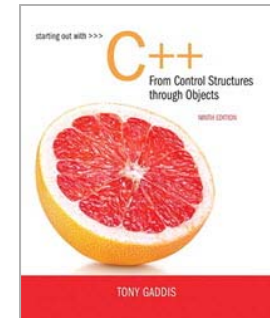
`? : . .* :: sizeof`

Overloading Types of Operators

- `++`, `--` operators overloaded differently for prefix vs. postfix notation
- 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;

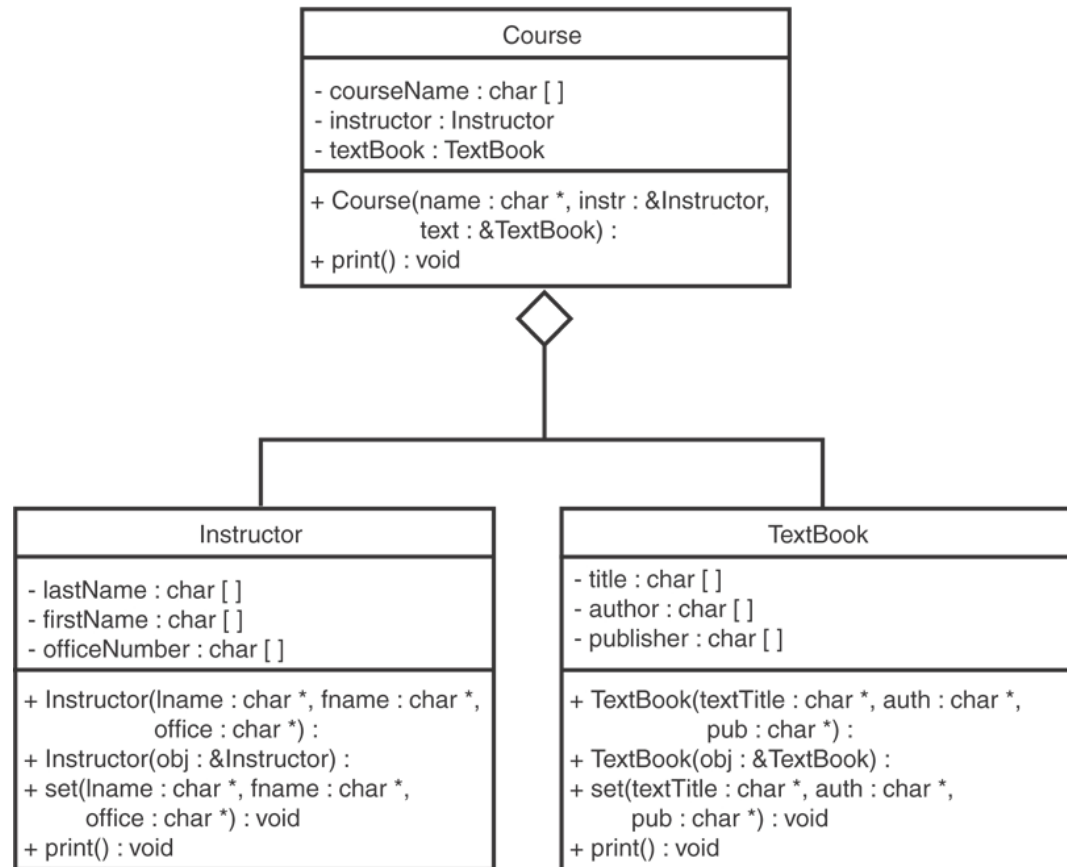
        ...
};

class Student
{
    private:
        StudentInfo personalData;

        ...
};
```



See the Instructor, TextBook, and Course classes in Chapter 14.





14.10

Rvalue References and Move Semantics

Temporary Values

- Consider this code:

```
int x;  
x = 2 * 6;
```

- When the expression $2 * 6$ is evaluated, the value 12 is stored in memory as a temporary value.
- The temporary value is then stored in the `x` variable.
- Then, the temporary value is discarded.

Temporary Values

- Consider this:

```
1 int square(int a)
2 {
3     return a * a;
4 }
5
6 int main()
7 {
8     int x = 0;
9
10    x = square(5);
11    cout << x << endl;
12    return 0;
13 }
```

- The square function is called, and the value 5 is passed as an argument.
- The square function calculates $5 * 5$ and stores the result, 25, as a temporary value.
- The temporary value is copied (assigned) to the variable x.
- The temporary value is no longer needed, so the system discards it.

Lvalues and Rvalues

- Two types of values stored in memory during the execution of a program:
 - Values that persist beyond the statement that created them, and have names that make them accessible to other statements in the program. In C++, these values are called *lvalues*.
 - Values that are temporary, and cannot be accessed beyond the statement that created them. In C++, these values are called *rvalues*.

Rvalue References

- Rvalue Reference: a reference variable that can refer only to temporary objects that would otherwise have no name.
- Rvalue references are used to write move constructors and move assignment operators (otherwise known as move semantics).
- Anytime you write a class with a pointer or reference to a piece of data outside the class, you should implement move semantics.
- Move semantics increase the performance of these types of classes.

Move Assignment vs. Copy Assignment

🍊 From the Person class, in Chapter 14:

```
// Move assignment operator
Person& operator=(Person&& right)
{ if (this != &right)
  {
    swap(name, right.name);
  }
  return *this;
}
```

```
// Copy assignment operator
Person & operator=(const Person &right)
{ if (this != &right)
  {
    name = new char[strlen(right.name) + 1];
    strcpy(name, right.name);
  }
  return *this;
}
```

Move Constructor vs. Copy Constructor

🍊 From the Person class, in Chapter 14:

```
// Move constructor
Person(Person&& temp)
{ // Steal the name pointer from temp.
  name = temp.name;

  // Nullify the temp object's name pointer.
  temp.name = nullptr;
}
```

```
// Copy constructor
Person(const Person &obj)
{ name = new char[strlen(obj.name) + 1];
  strcpy(name, obj.name); }
```