EEE110 - Computer Programming
Week 11: Introduction to Classes

ADANA ALPARSLAN TÜRKEŞ
SCIENCE AND TECHNOLOGY UNIVERSITY

Dr Kasım Zor

Department of Electrical and Electronic Engineering

Spring 2020

Chapter 13:

Introduction to Classes

13.1

- Procedural and Object-Oriented Programming

Procedural and Object-Oriented Programming

- Procedural programming focuses on the process/actions that occur in a program
- Object-Oriented programming is based on the data and the functions that operate on it. Objects are instances of ADTs that represent the data and its functions

Limitations of Procedural Programming

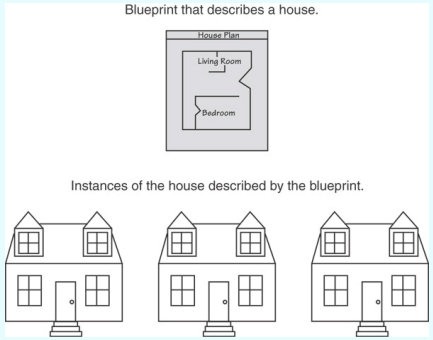
- If the data structures change, many functions must also be changed
- Programs that are based on complex function hierarchies are:
  - difficult to understand and maintain
  - difficult to modify and extend
  - easy to break

Object-Oriented Programming Terminology

- class: like a `struct` (allows bundling of related variables), but variables and functions in the class can have different properties than in a `struct`
- object: an instance of a `class`, in the same way that a variable can be an instance of a `struct`

Classes and Objects

- A Class is like a blueprint and objects are like houses built from the blueprint



Object-Oriented Programming Terminology

- attributes: members of a class
- methods or behaviors: member functions of a class

More on Objects

- data hiding: restricting access to certain members of an object
- public interface: members of an object that are available outside of the object. This allows the object to provide access to some data and functions without sharing its internal details and design, and provides some protection from data corruption

13.2

- Introduction to Classes

Introduction to Classes

- Objects are created from a `class`
- Format:

```
class ClassName
{
        declaration;
        declaration;
};
```

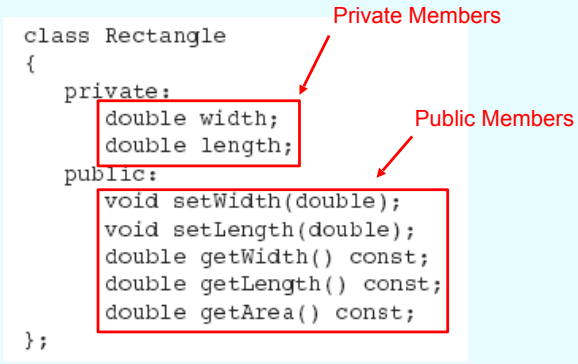

Class Example

```
class Rectangle
{
   private:
      double width;
      double length;
   public:
      void setWidth(double);
      void setLength(double);
      double getWidth() const;
      double getLength() const;
      double getArea() const;
};
```

Access Specifiers

- Used to control access to members of the class
- `public:` can be accessed by functions outside of the class
- `private:` can only be called by or accessed by functions that are members of the class

Class Example



More on Access Specifiers

- Can be listed in any order in a class
- Can appear multiple times in a class
- If not specified, the default is `private`

Using `const` With Member Functions

- `const` appearing after the parentheses in a member function declaration specifies that the function will not change any data in the calling object.

```
double getWidth() const;
double getLength() const;
double getArea() const;
```

Defining a Member Function

- When defining a member function:
  - Put prototype in class declaration
  - Define function using class name and scope resolution operator `(::)`

```
int Rectangle::setWidth(double w)
{
     width = w;
}
```


Accessors and Mutators

- Mutator: a member function that stores a value in a private member variable, or changes its value in some way
- Accessor: function that retrieves a value from a private member variable. Accessors do not change an object's data, so they should be marked `const`.

13.3

Defining an Instance of a Class

Defining an Instance of a Class

- An object is an instance of a class
- Defined like structure variables:

```
Rectangle r;
```

- Access members using dot operator:

```
r.setWidth(5.2);
cout << r.getWidth();
```

- Compiler error if attempt to access `private` member using dot operator

Program 13-1

```
// This program demonstrates a simple class.
#include <iostream>
using namespace std;

// Rectangle class declaration.
class Rectangle
{
   private:
      double width;
      double length;
   public:
      void setWidth(double);
      void setLength(double);
      double getWidth() const;
      double getLength() const;
      double getArea() const;
};

//***************************************************
// setWidth assigns a value to the width member.   *
//***************************************************

void Rectangle::setWidth(double w)
{
   width = w;
}

//***************************************************
// setLength assigns a value to the length member. *
//***************************************************

```


Program 13-1 (Continued)

```
void Rectangle::setLength(double len)
{
   length = len;
}

//***************************************************
// getWidth returns the value in the width member. *
//***************************************************

double Rectangle::getWidth() const
{
   return width;
}

//***************************************************
// getLength returns the value in the length member. *
//***************************************************

double Rectangle::getLength() const
{
   return length;
}

```


Program 13-1 (Continued)

```
//***************************************************
// getArea returns the product of width times length. *
//***************************************************

double Rectangle::getArea() const
{
   return width * length;
}

//***************************************************
// Function main                                    *
//***************************************************

int main()
{
   Rectangle box;     // Define an instance of the Rectangle class
   double rectWidth;  // Local variable for width
   double rectLength; // Local variable for length

   // Get the rectangle's width and length from the user.
   cout << "This program will calculate the area of a\n";
   cout << "rectangle. What is the width? ";
   cin >> rectWidth;
   cout << "What is the length? ";
   cin >> rectLength;

   // Store the width and length of the rectangle
   // in the box object.
   box.setWidth(rectWidth);
   box.setLength(rectLength);
```


Program 13-1 (Continued)

```

   // Display the rectangle's data.
   cout << "Here is the rectangle's data:\n";
   cout << "Width: " << box.getWidth() << endl;
   cout << "Length: " << box.getLength() << endl;
   cout << "Area: " << box.getArea() << endl;
   return 0;
}
```

Program Output

```
This program will calculate the area of a
rectangle. What is the width? 10 [Enter]
What is the length? 5 [Enter]
Here is the rectangle's data:
Width: 10
Length: 5
Area: 50
```

Avoiding Stale Data

- Some data is the result of a calculation.
- In the `Rectangle` class the area of a rectangle is calculated.
  - length x width
- If we were to use an `area` variable here in the `Rectangle` class, its value would be dependent on the length and the width.
- If we change `length` or `width` without updating `area`, then `area` would become *stale*.
- To avoid stale data, it is best to calculate the value of that data within a member function rather than store it in a variable.

Pointer to an Object

- Can define a pointer to an object:

```
Rectangle *rPtr;
```

- Can access public members via pointer:

```
rPtr = &otherRectangle;
rPtr->setLength(12.5);
cout << rPtr->getLenght() << endl;
```


Dynamically Allocating an Object

- We can also use a pointer to dynamically allocate an object.

```
1  // Define a Rectangle pointer.
2  Rectangle *rectPtr;
3
4  // Dynamically allocate a Rectangle object.
5  rectPtr = new Rectangle;
6
7  // Store values in the object's width and length.
8  rectPtr->setWidth(10.0);
9  rectPtr->setLength(15.0);
10
11 // Delete the object from memory.
12 delete rectPtr;
13 rectPtr = 0;
```

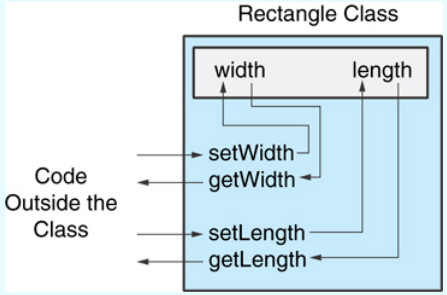

13.4

Why Have Private Members?

Why Have Private Members?

- Making data members `private` provides data protection
- Data can be accessed only through `public` functions
- Public functions define the class's public interface

Code outside the class must use the class's public member functions to interact with the object.



13.5

Separating Specification from Implementation

Separating Specification from Implementation

– Place class declaration in a header file that serves as the class specification file. Name the file *`ClassName`*`.h`, for example, `Rectangle.h`
– Place member function definitions in *`ClassName`*`.cpp`, for example, `Rectangle.cpp` File should `#include` the class specification file
– Programs that use the class must `#include` the class specification file, and be compiled and linked with the member function definitions

13.6

Inline Member Functions

Inline Member Functions

- Member functions can be defined
  - inline: in class declaration
  - after the class declaration
- Inline appropriate for short function bodies:

```
int getWidth() const
  { return width; }
```


Rectangle Class with Inline Member Functions

```
 1  // Specification file for the Rectangle class
 2  // This version uses some inline member functions.
 3  #ifndef RECTANGLE_H
 4  #define RECTANGLE_H
 5
 6  class Rectangle
 7  {
 8     private:
 9        double width;
10        double length;
11     public:
12        void setWidth(double);
13        void setLength(double);
14
15        double getWidth() const
16           { return width; }
17
18        double getLength() const
19           { return length; }
20
21        double getArea() const
22           { return width * length; }
23  };
24  #endif
```


Tradeoffs – Inline vs. Regular Member Functions

- Regular functions – when called, compiler stores return address of call, allocates memory for local variables, etc.
- Code for an inline function is copied into program in place of call – larger executable program, but no function call overhead, hence faster execution

13.7

Constructors

Constructors

- Member function that is automatically called when an object is created
- Purpose is to construct an object
- Constructor function name is class name
- Has no return type

Contents of `Rectangle.h` (Version 3)

```
1  // Specification file for the Rectangle class
2  // This version has a constructor.
3  #ifndef RECTANGLE_H
4  #define RECTANGLE_H
5
6  class Rectangle
7  {
8     private:
9        double width;
10       double length;
11    public:
12       Rectangle();              // Constructor
13       void setWidth(double);
14       void setLength(double);
15
16       double getWidth() const
17          { return width; }
18
19       double getLength() const
20          { return length; }
21
22       double getArea() const
23          { return width * length; }
24 };
25 #endif
```


Contents of `Rectangle.cpp` (Version 3)

```
1  // Implementation file for the Rectangle class.
2  // This version has a constructor.
3  #include "Rectangle.h"    // Needed for the Rectangle class
4  #include <iostream>       // Needed for cout
5  #include <cstdlib>        // Needed for the exit function
6  using namespace std;
7
8  //***********************************************************
9  // The constructor initializes width and length to 0.0.     *
10 //***********************************************************
11
12 Rectangle::Rectangle()
13 {
14    width = 0.0;
15    length = 0.0;
16 }
```

Continues...

Contents of `Rectangle.ccp` Version3 (continued)

```
17
18  //***************************************************************
19  // setWidth sets the value of the member variable width.    *
20  //***************************************************************
21
22  void Rectangle::setWidth(double w)
23  {
24     if (w >= 0)
25        width = w;
26     else
27     {
28        cout << "Invalid width\n";
29        exit(EXIT_FAILURE);
30     }
31  }
32
33  //***************************************************************
34  // setLength sets the value of the member variable length.  *
35  //***************************************************************
36
37  void Rectangle::setLength(double len)
38  {
39     if (len >= 0)
40        length = len;
41     else
42     {
43        cout << "Invalid length\n";
44        exit(EXIT_FAILURE);
45     }
46  }
```


Program 13-6

```
1  // This program uses the Rectangle class's constructor.
2  #include <iostream>
3  #include "Rectangle.h"  // Needed for Rectangle class
4  using namespace std;
5
6  int main()
7  {
8     Rectangle box;     // Define an instance of the Rectangle class
9
10    // Display the rectangle's data.
11    cout << "Here is the rectangle's data:\n";
12    cout << "Width: " << box.getWidth() << endl;
13    cout << "Length: " << box.getLength() << endl;
14    cout << "Area: " << box.getArea() << endl;
15    return 0;
16 }
```

Program 13-6 *(continued)*

Program Output

```
Here is the rectangle's data:
Width: 0
Length: 0
Area: 0
```


Default Constructors

- A default constructor is a constructor that takes no arguments.
- If you write a class with no constructor at all, C++ will write a default constructor for you, one that does nothing.
- A simple instantiation of a class (with no arguments) calls the default constructor:

```
Rectangle r;
```


13.8

Passing Arguments to Constructors

Passing Arguments to Constructors

- To create a constructor that takes arguments:
  - indicate parameters in prototype:

```
Rectangle(double, double);
```

  - Use parameters in the definition:

```
Rectangle::Rectangle(double w, double len)
{
   width = w;
   length = len;
}
```


Passing Arguments to Constructors

- You can pass arguments to the constructor when you create an object:

```
Rectangle r(10, 5);
```


More About Default Constructors

- If all of a constructor's parameters have default arguments, then it is a default constructor. For example:

```
Rectangle(double = 0, double = 0);
```

- Creating an object and passing no arguments will cause this constructor to execute:

```
Rectangle r;
```


Classes with No Default Constructor

- When all of a class's constructors require arguments, then the class has NO default constructor.
- When this is the case, you must pass the required arguments to the constructor when creating an object.

13.9

Destructors

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

Contents of `InventoryItem.h` (Version 1)

```
1  // Specification file for the InventoryItem class.
2  #ifndef INVENTORYITEM_H
3  #define INVENTORYITEM_H
4  #include <cstring>   // Needed for strlen and strcpy
5
6  // InventoryItem class declaration.
7  class InventoryItem
8  {
9  private:
10    char *description;  // The item description
11    double cost;        // The item cost
12    int units;          // Number of units on hand
```


Contents of `InventoryItem.h` Version1 (Continued)

```
13 public:
14    // Constructor
15    InventoryItem(char *desc, double c, int u)
16       { // Allocate just enough memory for the description.
17         description = new char [strlen(desc) + 1];
18
19         // Copy the description to the allocated memory.
20         strcpy(description, desc);
21
22         // Assign values to cost and units.
23         cost = c;
24         units = u;}
25
26    // Destructor
27    ~InventoryItem()
28       { delete [] description; }
29
30    const char *getDescription() const
31       { return description; }
32
33    double getCost() const
34       { return cost; }
35
36    int getUnits() const
37       { return units; }
38 };
39 #endif
```

Program 13-11

```
1  // This program demonstrates a class with a destructor.
2  #include <iostream>
3  #include <iomanip>
4  #include "InventoryItem.h"
5  using namespace std;
6
7  int main()
8  {
9     // Define an InventoryItem object with the following data:
10    // Description: Wrench  Cost: 8.75   Units on hand: 20
11    InventoryItem stock("Wrench", 8.75, 20);
12
13    // Set numeric output formatting.
14    cout << setprecision(2) << fixed << showpoint;
15
```


Program 13-11 *(continued)*

```
16    // Display the object's data.
17    cout << "Item Description: " << stock.getDescription() << endl;
18    cout << "Cost: $" << stock.getCost() << endl;
19    cout << "Units on hand: " << stock.getUnits() << endl;
20    return 0;
21 }
```

Program Output

```
Item Description: Wrench
Cost: $8.75
Units on hand: 20
```


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;
```


13.10

Overloading Constructors

Overloading Constructors

- A class can have more than one constructor
- Overloaded constructors in a class must have different parameter lists:

```
Rectangle();
Rectangle(double);
Rectangle(double, double);
```



```
// This class has overloaded constructors.
#ifndef INVENTORYITEM_H
#define INVENTORYITEM_H
#include <string>
using namespace std;

class InventoryItem
{
private:
   string description; // The item description
   double cost;        // The item cost
   int units;          // Number of units on hand
public:
   // Constructor #1
   InventoryItem()
      { // Initialize description, cost, and units.
        description = "";
        cost = 0.0;
        units = 0; }

   // Constructor #2
   InventoryItem(string desc)
      { // Assign the value to description.
        description = desc;

        // Initialize cost and units.
        cost = 0.0;
        units = 0; }
```

Continues...


```

   // Constructor #3
   InventoryItem(string desc, double c, int u)
     { // Assign values to description, cost, and units.
       description = desc;
       cost = c;
       units = u; }

   // Mutator functions
   void setDescription(string d)
      { description = d; }

   void setCost(double c)
      { cost = c; }

   void setUnits(int u)
      { units = u; }

   // Accessor functions
   string getDescription() const
      { return description; }

   double getCost() const
      { return cost; }

   int getUnits() const
      { return units; }
};
#endif
```


Only One Default Constructor and One Destructor

- Do not provide more than one default constructor for a class: one that takes no arguments and one that has default arguments for all parameters

```
Square();
Square(int = 0);  // will not compile
```

- Since a destructor takes no arguments, there can only be one destructor for a class

Member Function Overloading

- Non-constructor member functions can also be overloaded:

```
void setCost(double);
void setCost(char *);
```

- Must have unique parameter lists as for constructors

3.11

Using Private Member Functions

Using Private Member Functions

- A `private` member function can only be called by another member function
- It is used for internal processing by the class, not for use outside of the class
- See the `createDescription` function in **`ContactInfo.h`** (Version 2)

13.12

Arrays of Objects

Arrays of Objects

- Objects can be the elements of an array:

```
InventoryItem inventory[40];
```

- Default constructor for object is used when array is defined

Arrays of Objects

- Must use initializer list to invoke constructor that takes arguments:

```
InventoryItem inventory[3] =
  { "Hammer", "Wrench", "Pliers" };
```


Arrays of Objects

- If the constructor requires more than one argument, the initializer must take the form of a function call:

```
InventoryItem inventory[3] = { InventoryItem("Hammer", 6.95, 12),
                               InventoryItem("Wrench", 8.75, 20),
                               InventoryItem("Pliers", 3.75, 10) };
```


Arrays of Objects

- It isn't necessary to call the same constructor for each object in an array:

```
InventoryItem inventory[3] = { "Hammer",
                               InventoryItem("Wrench", 8.75, 20),
                               "Pliers" };
```

Accessing Objects in an Array

- Objects in an array are referenced using subscripts
- Member functions are referenced using dot notation:

```
inventory[2].setUnits(30);
cout << inventory[2].getUnits();
```


Program 13-13

```
1  // This program demonstrates an array of class objects.
2  #include <iostream>
3  #include <iomanip>
4  #include "InventoryItem.h"
5  using namespace std;
6
7  int main()
8  {
9     const int NUM_ITEMS = 5;
10    InventoryItem inventory[NUM_ITEMS] = {
11                  InventoryItem("Hammer", 6.95, 12),
12                  InventoryItem("Wrench", 8.75, 20),
13                  InventoryItem("Pliers", 3.75, 10),
14                  InventoryItem("Ratchet", 7.95, 14),
15                  InventoryItem("Screwdriver", 2.50, 22) };
16
17    cout << setw(14) <<"Inventory Item"
18         << setw(8) << "Cost" << setw(8)
19         << setw(16) << "Units On Hand\n";
20    cout << "-------------------------------------\n";
```


Program 13-3 (Continued)

```
21
22     for (int i = 0; i < NUM_ITEMS; i++)
23     {
24        cout << setw(14) << inventory[i].getDescription();
25        cout << setw(8) << inventory[i].getCost();
26        cout << setw(7) << inventory[i].getUnits() << endl;
27     }
28
29     return 0;
30  }
```

Program Output

```
Inventory Item    Cost  Units On Hand
-------------------------------------
       Hammer    6.95     12
       Wrench    8.75     20
       Pliers    3.75     10
      Ratchet    7.95     14
   Screwdriver    2.5      22
```


13.15

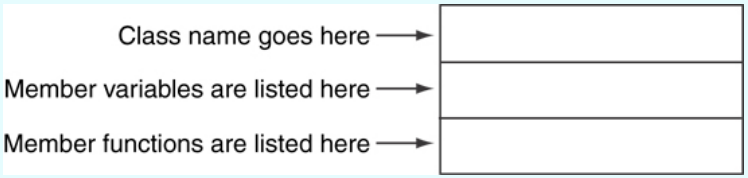
The Unified Modeling Language

The Unified Modeling Language

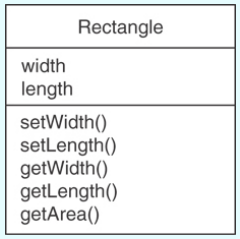
- *UML* stands for *Unified Modeling Language*.
- The UML provides a set of standard diagrams for graphically depicting object-oriented systems

UML Class Diagram

- A UML diagram for a class has three main sections.



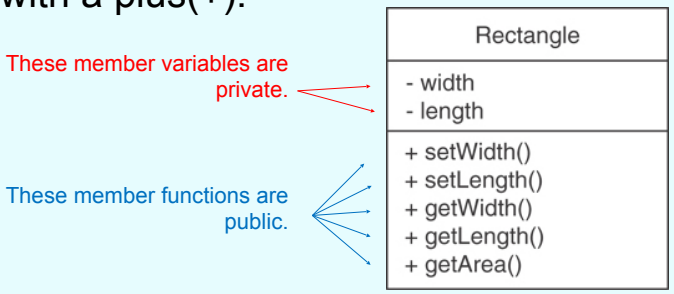
Example: A Rectangle Class




```
class Rectangle
{
   private:
      double width;
      double length;
   public:
      bool setWidth(double);
      bool setLength(double);
      double getWidth() const;
      double getLength() const;
      double getArea() const;
};
```


UML Access Specification Notation

- In UML you indicate a private member with a minus (-) and a public member with a plus(+).



UML Data Type Notation

- To indicate the data type of a member variable, place a colon followed by the name of the data type after the name of the variable.

  - width : double
  - length : double

UML Parameter Type Notation

- To indicate the data type of a function's parameter variable, place a colon followed by the name of the data type after the name of the variable.

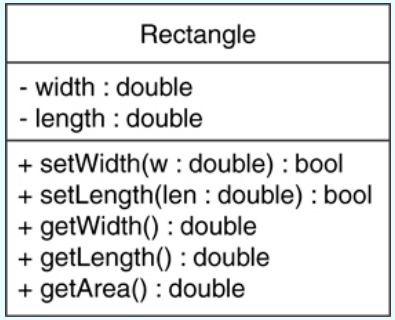
  + setWidth(w : double)

UML Function Return Type Notation

- To indicate the data type of a function's return value, place a colon followed by the name of the data type after the function's parameter list.

  + setWidth(w : double) : void

The Rectangle Class

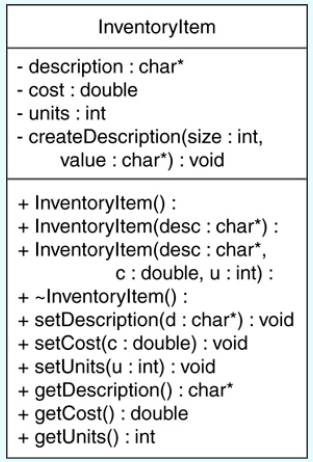


Showing Constructors and Destructors

No return type listed for constructors or destructors

Constructors

Destructor



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.

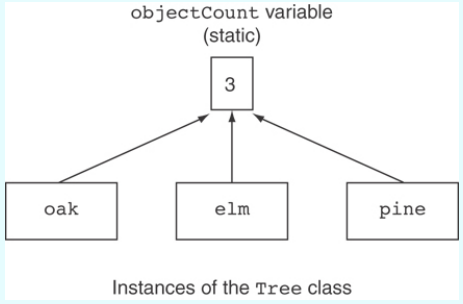
Program 14-1

```
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

```
// 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

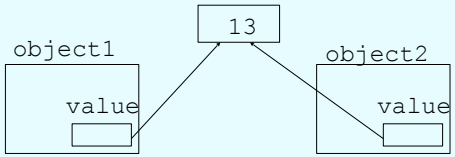
Problem: what if object contains a pointer?

```
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:

```
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:

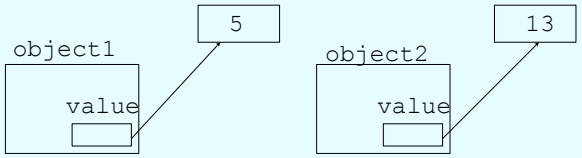
```
SomeClass::SomeClass(const SomeClass
&obj)
{
        value = new int;
        *value = obj.value;
}
```

- 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

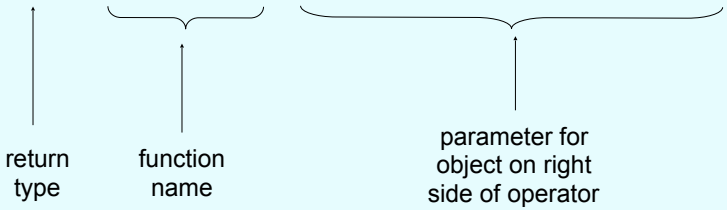
Operator Overloading

- 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

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
{
   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 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.