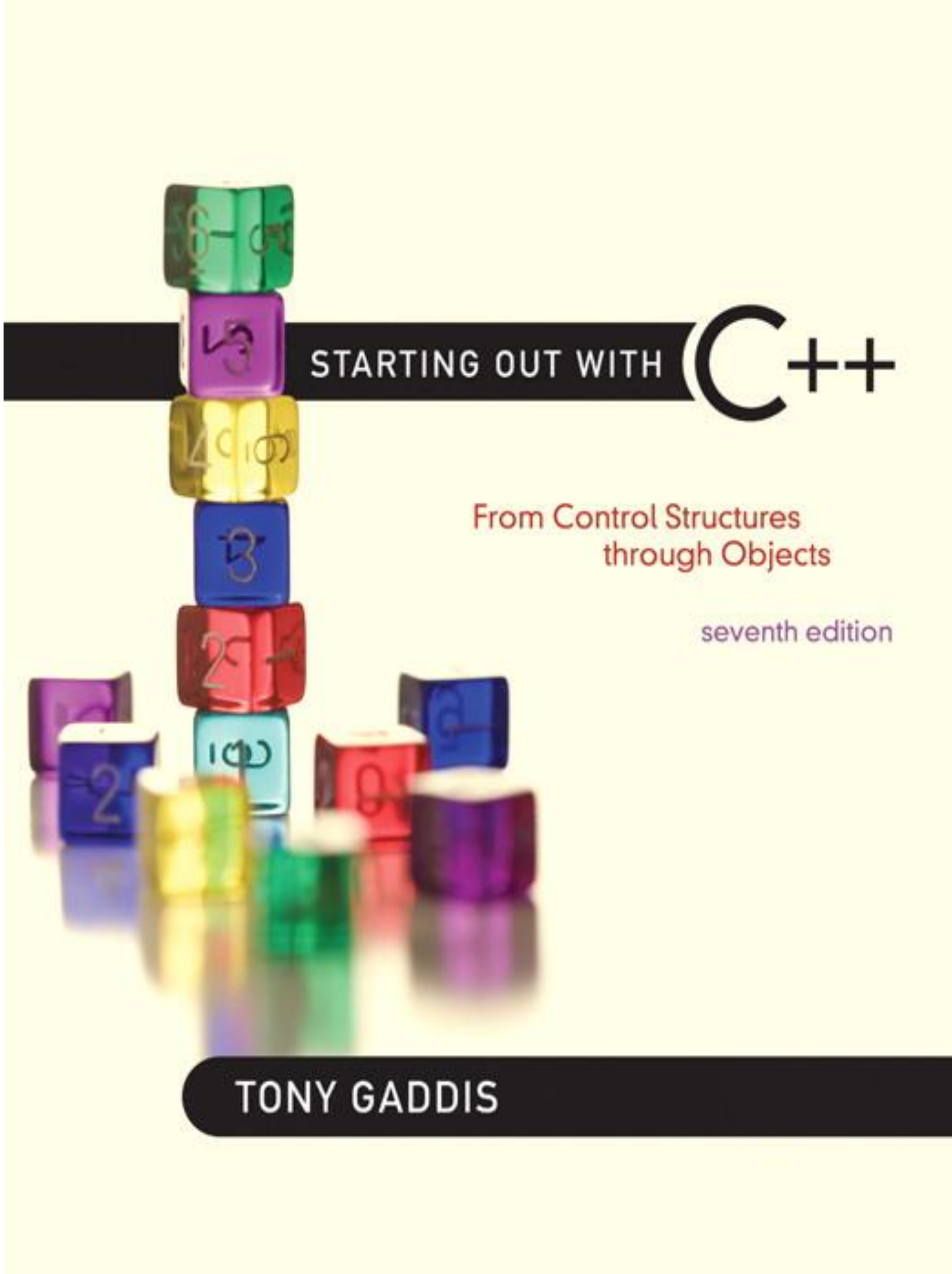
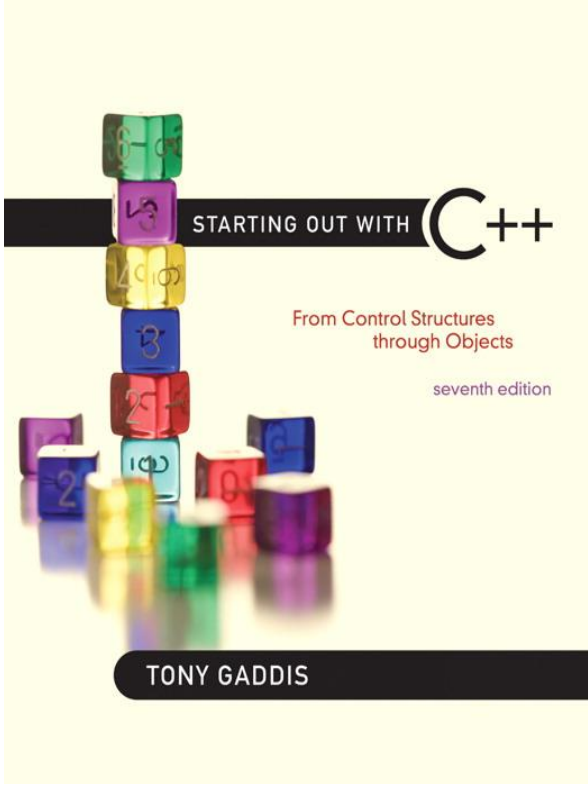
Chapter 13:

Introduction to Classes

Addison-Wesley
is an imprint of
PEARSON



CHAPTER 13   Topics



13.1

Procedural and Object-Oriented Programming

Copyright © 2012 Pearson Education, Inc.

Procedural and Object-Oriented Programming

- Procedural programming
  - focuses on the process/actions that occur in a program

- Object-Oriented programming
  - 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

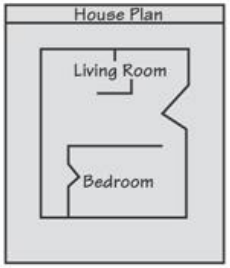
Object-Oriented Programming Terminology

- 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

Blueprint that describes a house.



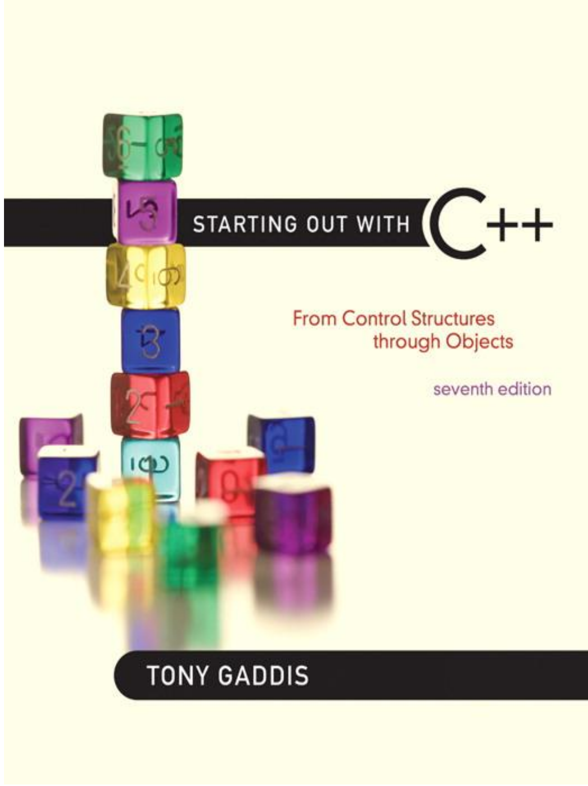
Instances of the house described by the blueprint.

Object-Oriented Programming Terminology

- attributes
  - Data members of a class
- methods or behaviors
  - Function members of a class
- Object “state”

More on Objects

- data hiding
  - restricting access to certain members of an object
  - private and protected members
- public interface
  - members of an object that are available outside of the object.



13.2

Introduction to Classes

Copyright © 2012 Pearson Education, Inc.

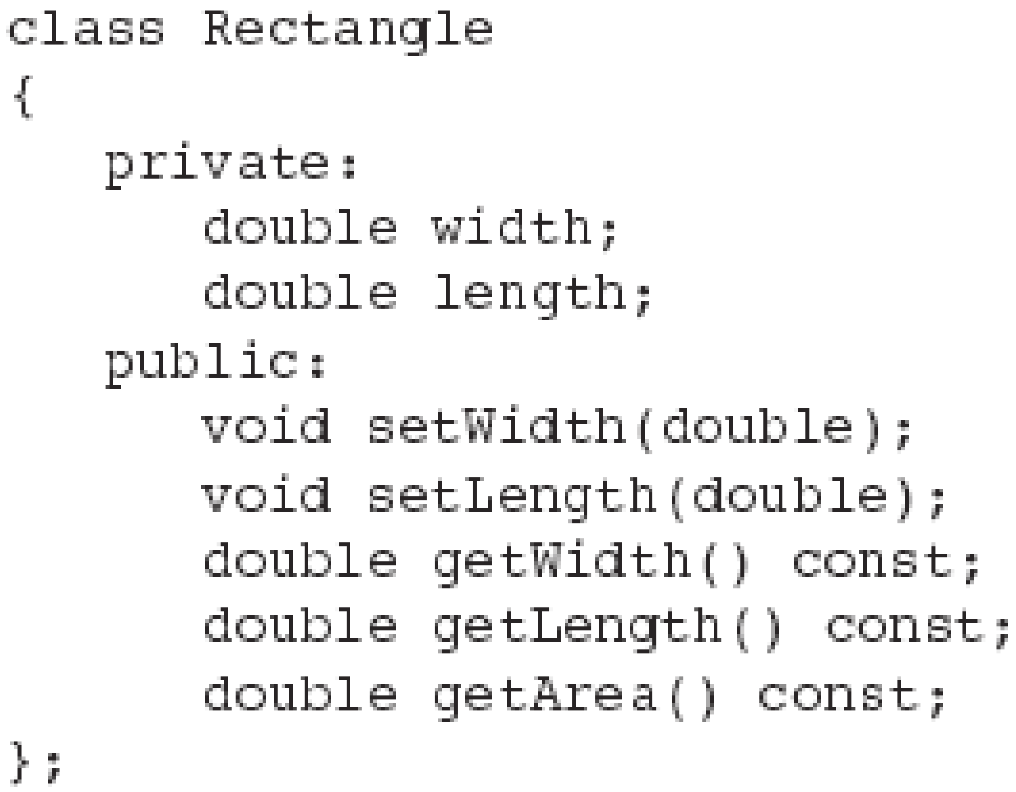
Introduction to Classes

- Objects are created from a `class`
  - `instantion`

- Class Declaration:

```
class ClassName
{
    <access specifiers>
    <member declarations/definitions>
    [global variables, functions]
};
```

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

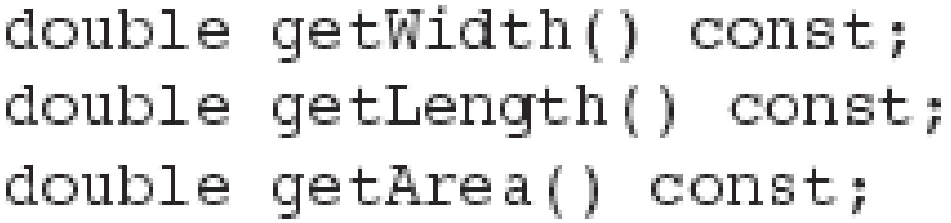
- Used to control access to members of the class
- `public:`
  - can be accessed by functions outside of the class
- `private:`
  - can only be called by or accessed by functions that are members of the class

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;
```

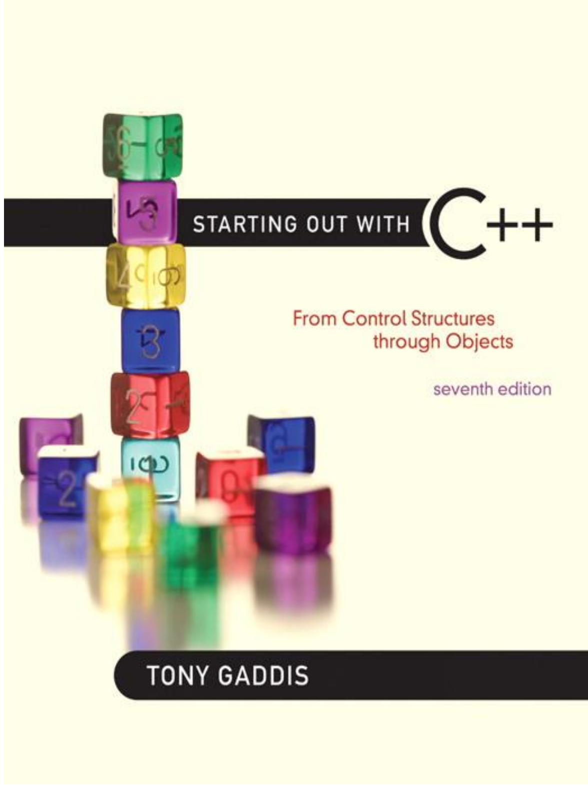
Defining a Member Function

- When defining a member function:
  - Put prototypes in class declaration
  - Define functions externally 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

Copyright © 2012 Pearson Education, Inc.

Defining an Instance of a Class

- An object is an instance of a class

- Defined like structure variables:

```
Rectangle r; //use default constructor
```

- Access members using dot operator:

```
r.setWidth(5.2);     //mutator
cout << r.getWidth();//accessor
```

- Attempting to access a private member using the dot operator - syntax

Programs 13-1, 13-2, 13-3

Avoiding “Stale” Data

- Some data is the result of a calculation.
- To avoid stale data, it is best to calculate the value of that data within a member function rather than store it in an instance variable.

Pointer to an Object

- Can define a pointer to an object:

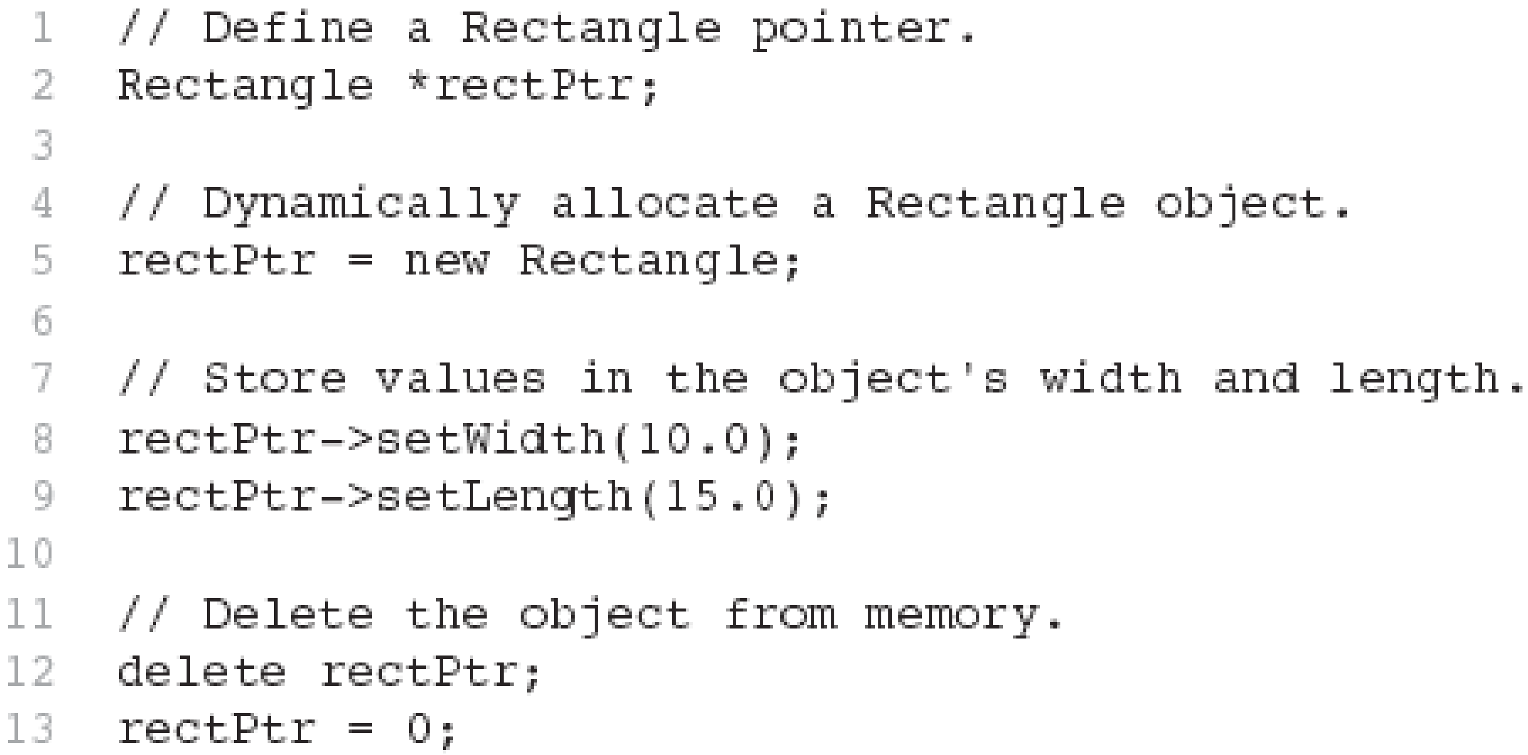
```
Rectangle *rPtr;
```

- Can access public members via pointer:

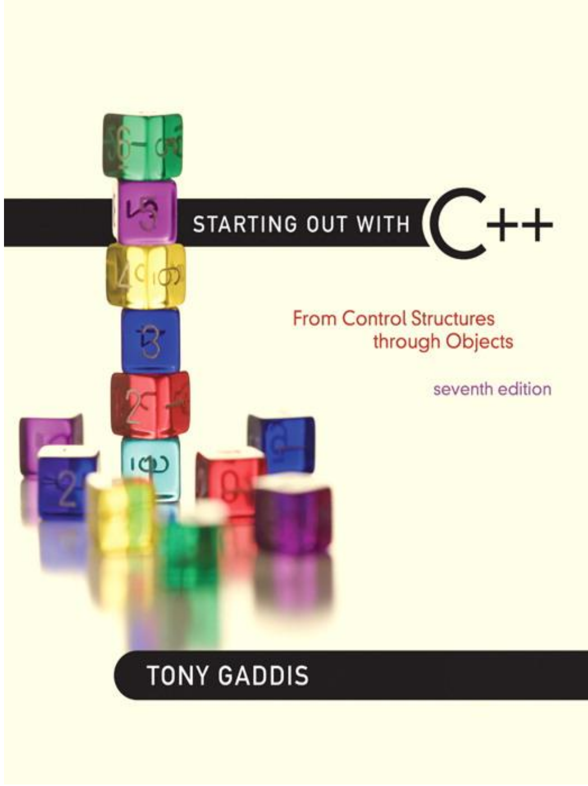
```
rPtr = &otherRectangle;
rPtr -> setLength(12.5);
cout << rPtr -> getLength() << endl;

(*rPtr).getLength()
```

Dynamically Allocating an Object [new]

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



```
// Define a Rectangle pointer.
Rectangle *rectPtr;

// Dynamically allocate a Rectangle object.
rectPtr = new Rectangle;

// Store values in the object's width and length.
rectPtr->setWidth(10.0);
rectPtr->setLength(15.0);

// Delete the object from memory.
delete rectPtr;
rectPtr = 0;
```



13.4

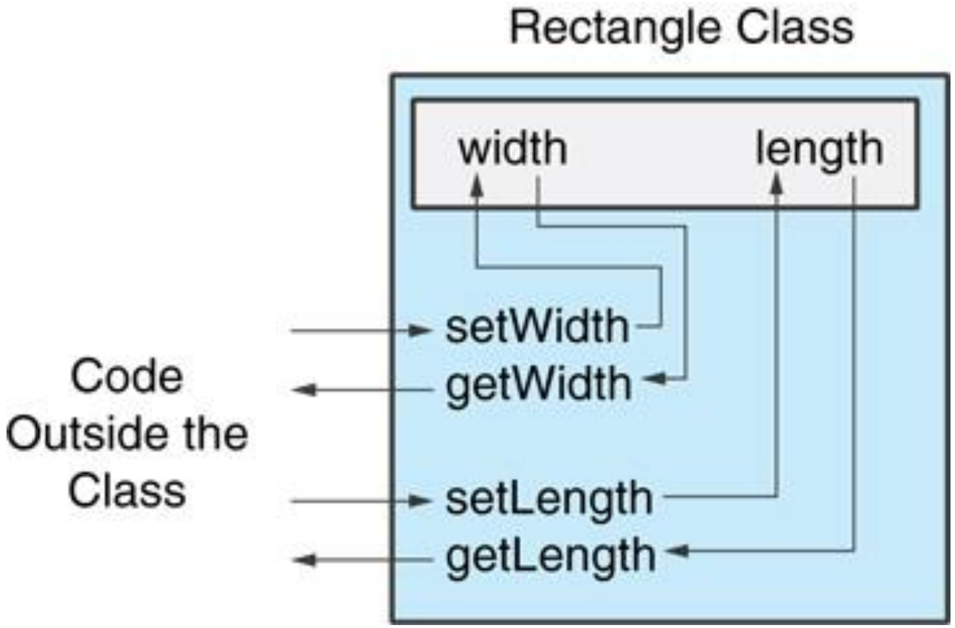
Private Members

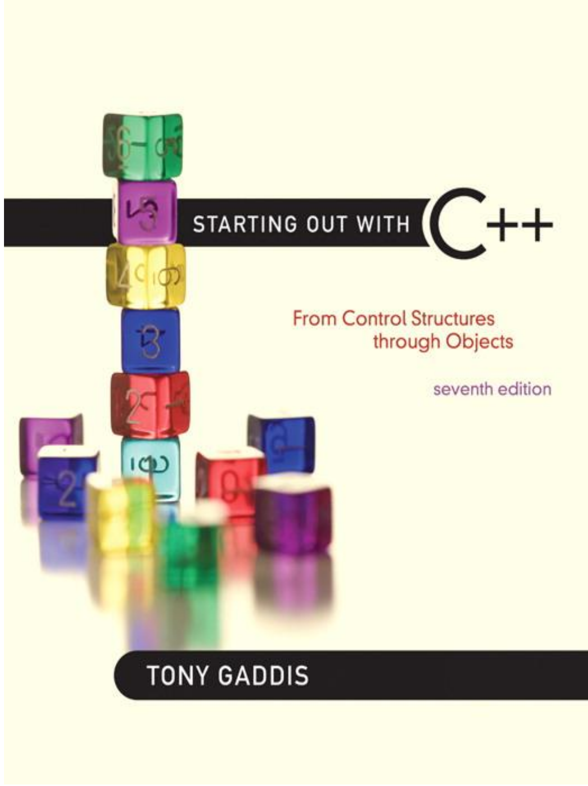
Copyright © 2012 Pearson Education, Inc.

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

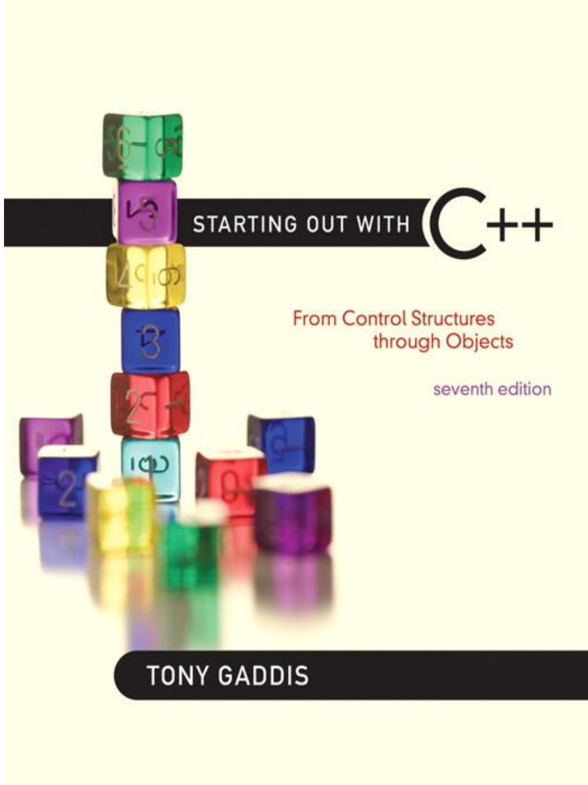
Copyright © 2012 Pearson Education, Inc.

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
- `#ifndef, #define, #endif (used in header files)`
- `Program 13-4`



13.6

Inline Member Functions

Copyright © 2012 Pearson Education, Inc.

Inline Member Functions

- Member functions can be defined
  - inline: in class declaration
  - after the class declaration (or in a different file)

- Inline appropriate for short function bodies:

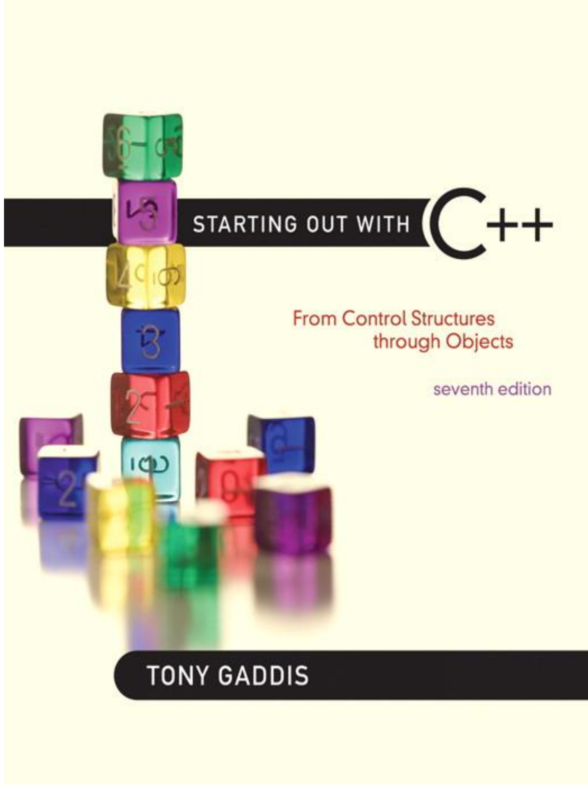
```
int getWidth() const
    { return width; }
```

- implicit/explicit use of “inline”

Rectangle Class with Inline Member Functions

rectangle.h (version 2)

```
// Specification file for the Rectangle class
// This version uses some inline member functions.
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle
{
   private:
      double width;
      double length;
   public:
      void setWidth(double);
      void setLength(double);

      double getWidth() const
         { return width; }

      double getLength() const
         { return length; }

      double getArea() const
         { return width * length; }
};
#endif
```



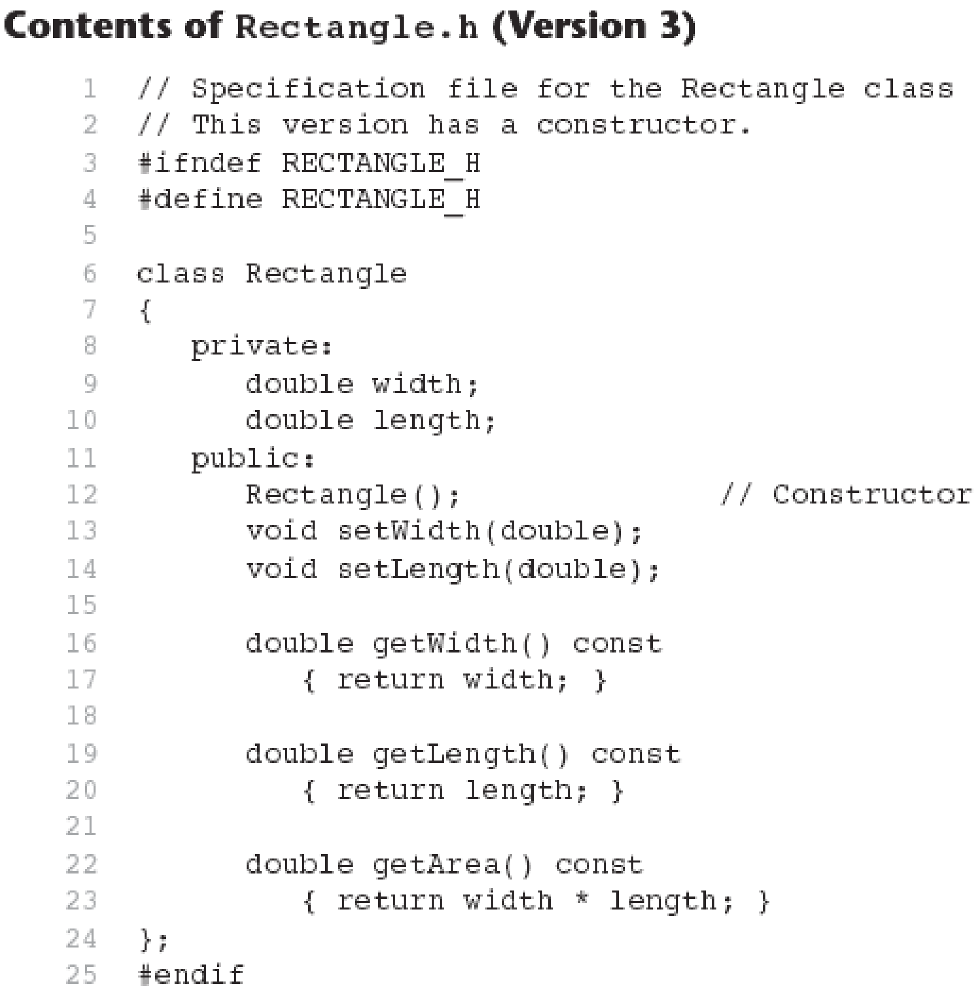
13.7

Constructors

Copyright © 2012 Pearson Education, Inc.

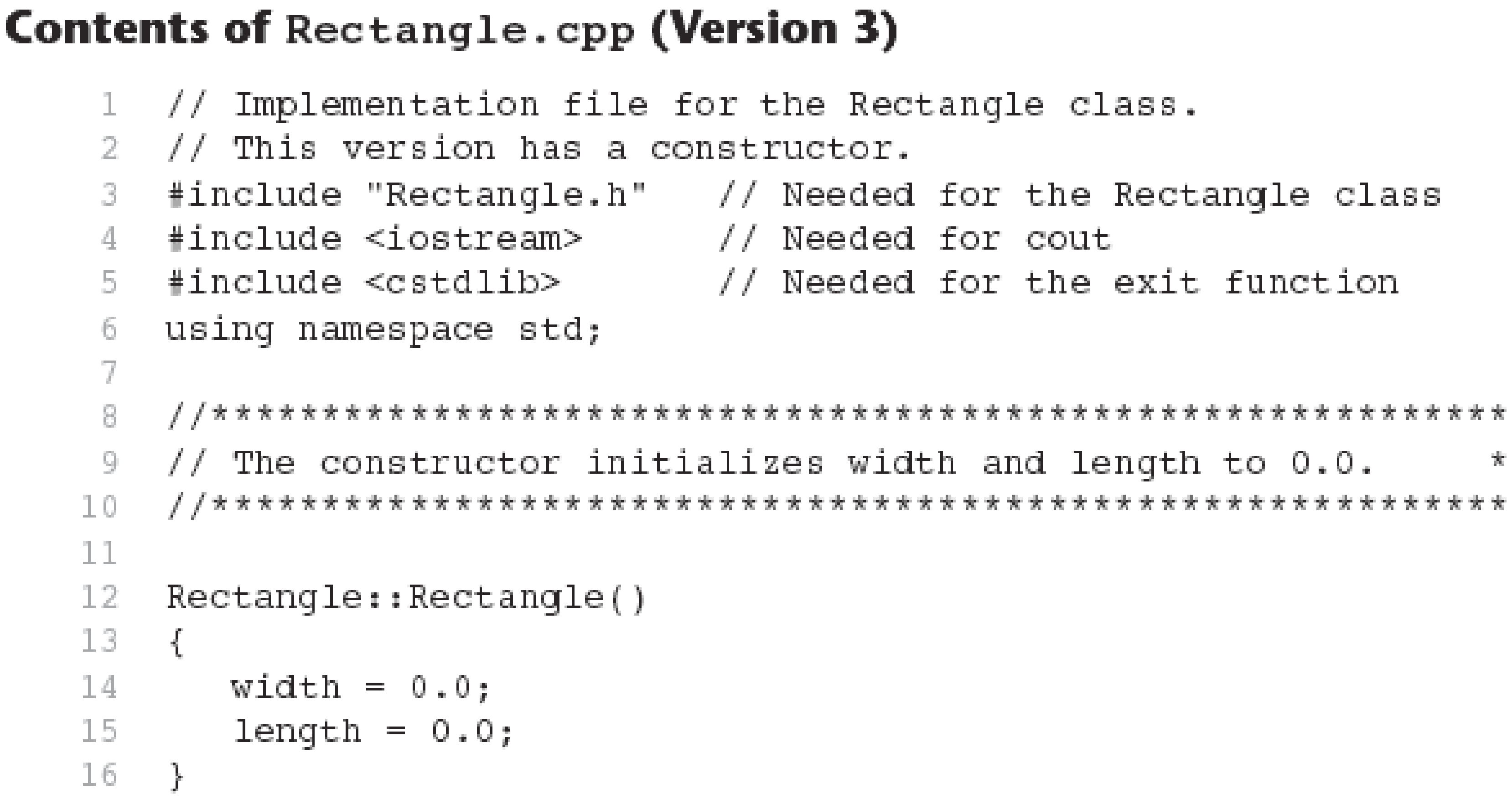
Constructors

- 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)

```
// Specification file for the Rectangle class
// This version has a constructor.
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle
{
   private:
      double width;
      double length;
   public:
      Rectangle();               // Constructor
      void setWidth(double);
      void setLength(double);

      double getWidth() const
         { return width; }

      double getLength() const
         { return length; }

      double getArea() const
         { return width * length; }
};
#endif
```

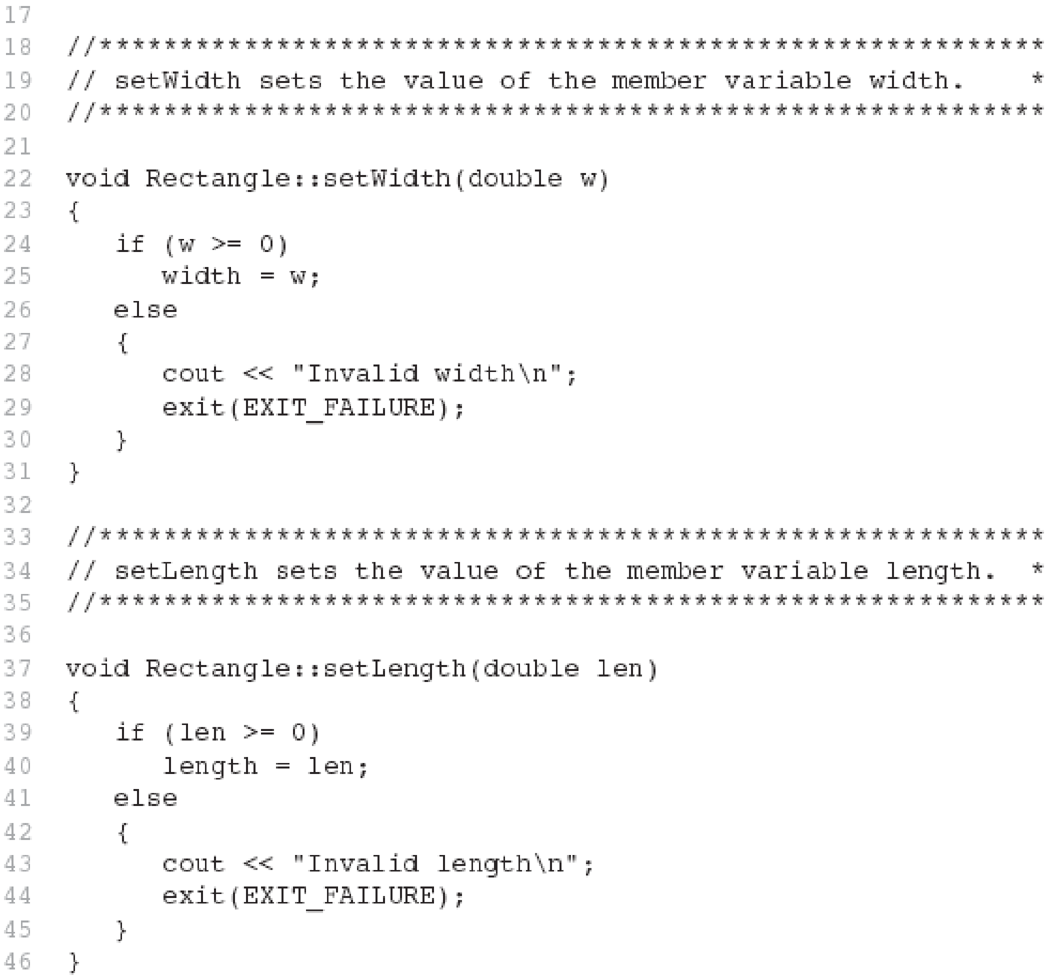


Contents of `Rectangle.cpp` (Version 3)

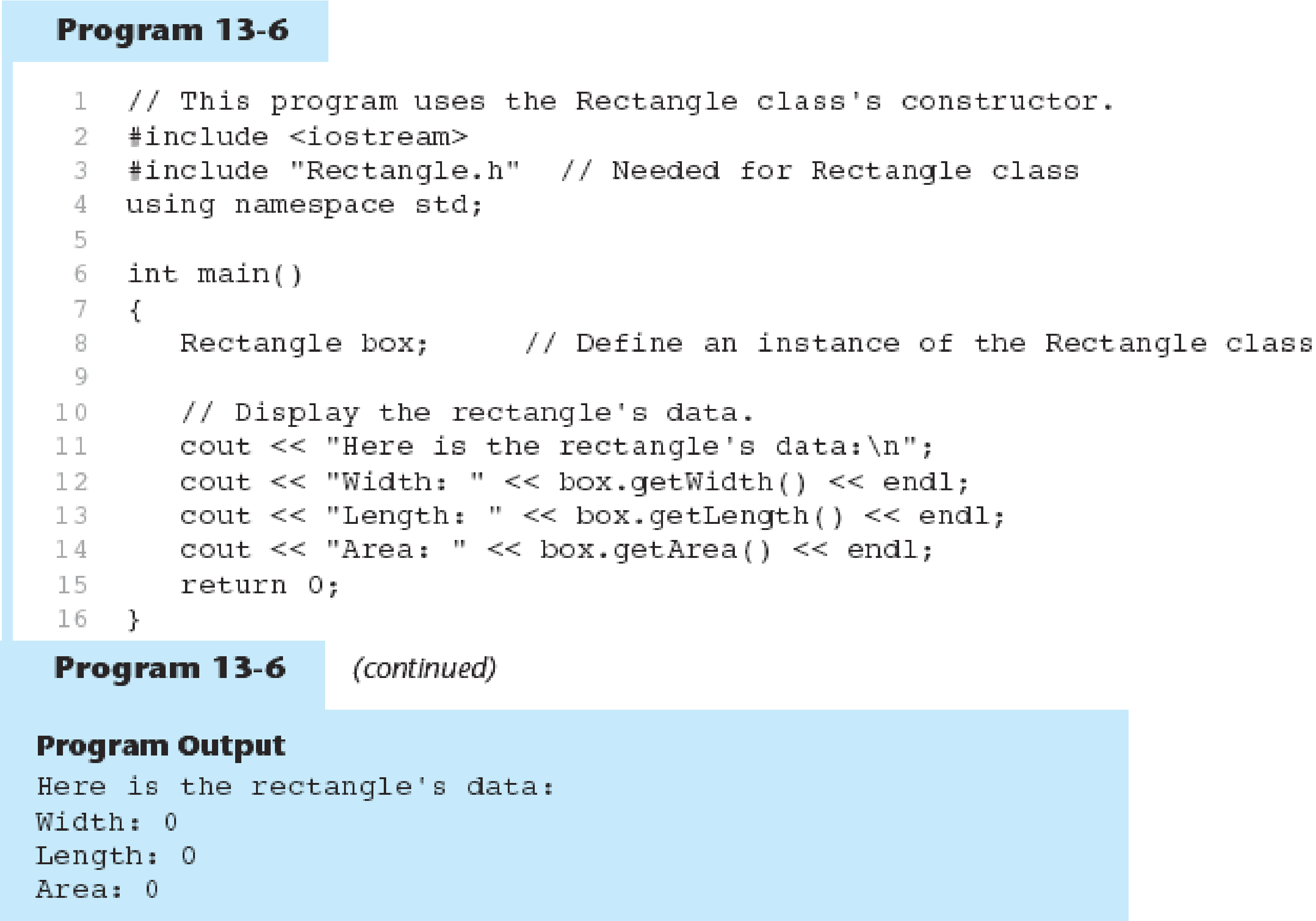
```
// Implementation file for the Rectangle class.
// This version has a constructor.
#include "Rectangle.h"    // Needed for the Rectangle class
#include <iostream>       // Needed for cout
#include <cstdlib>        // Needed for the exit function
using namespace std;

//************************************************************
// The constructor initializes width and length to 0.0.     *
//************************************************************

Rectangle::Rectangle()
{
   width = 0.0;
   length = 0.0;
}
```

Continues...

Contents of `Rectangle.cpp` Version3 (continued)



```
//***********************************************************
// setWidth sets the value of the member variable width.    *
//***********************************************************

void Rectangle::setWidth(double w)
{
   if (w >= 0)
      width = w;
   else
   {
      cout << "Invalid width\n";
      exit(EXIT_FAILURE);
   }
}

//***********************************************************
// setLength sets the value of the member variable length.  *
//***********************************************************

void Rectangle::setLength(double len)
{
   if (len >= 0)
      length = len;
   else
   {
      cout << "Invalid length\n";
      exit(EXIT_FAILURE);
   }
}
```



Program 13-6

```
// This program uses the Rectangle class's constructor.
#include <iostream>
#include "Rectangle.h"  // Needed for Rectangle class
using namespace std;

int main()
{
   Rectangle box;     // Define an instance of the Rectangle class

   // 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 13-6 *(continued)*

Program Output

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

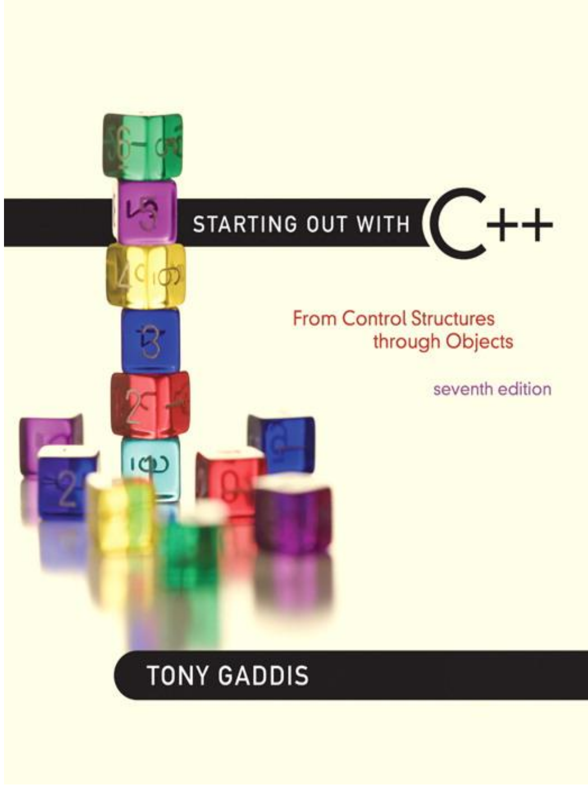
Default Constructors

- 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

Copyright © 2012 Pearson Education, Inc.

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);
```

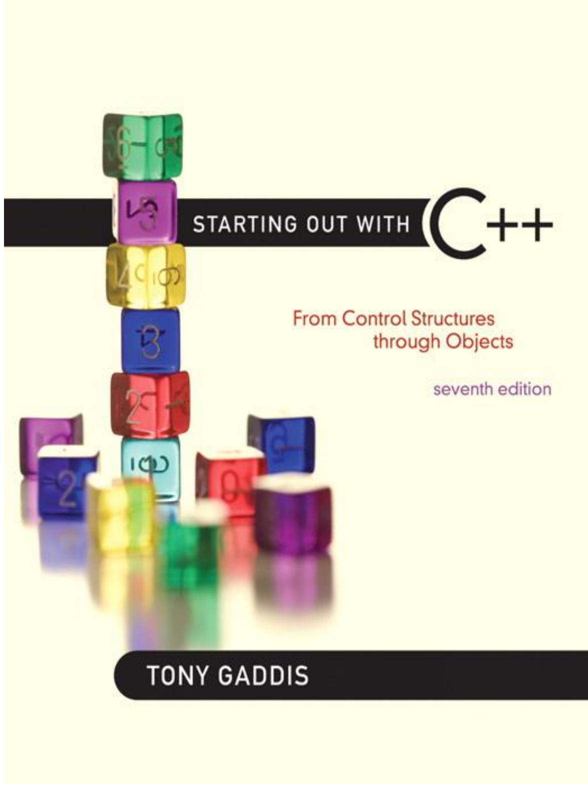
- Creating an object and passing no arguments will cause this constructor to execute:

```
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 or create a default constructor



13.9

Destructors

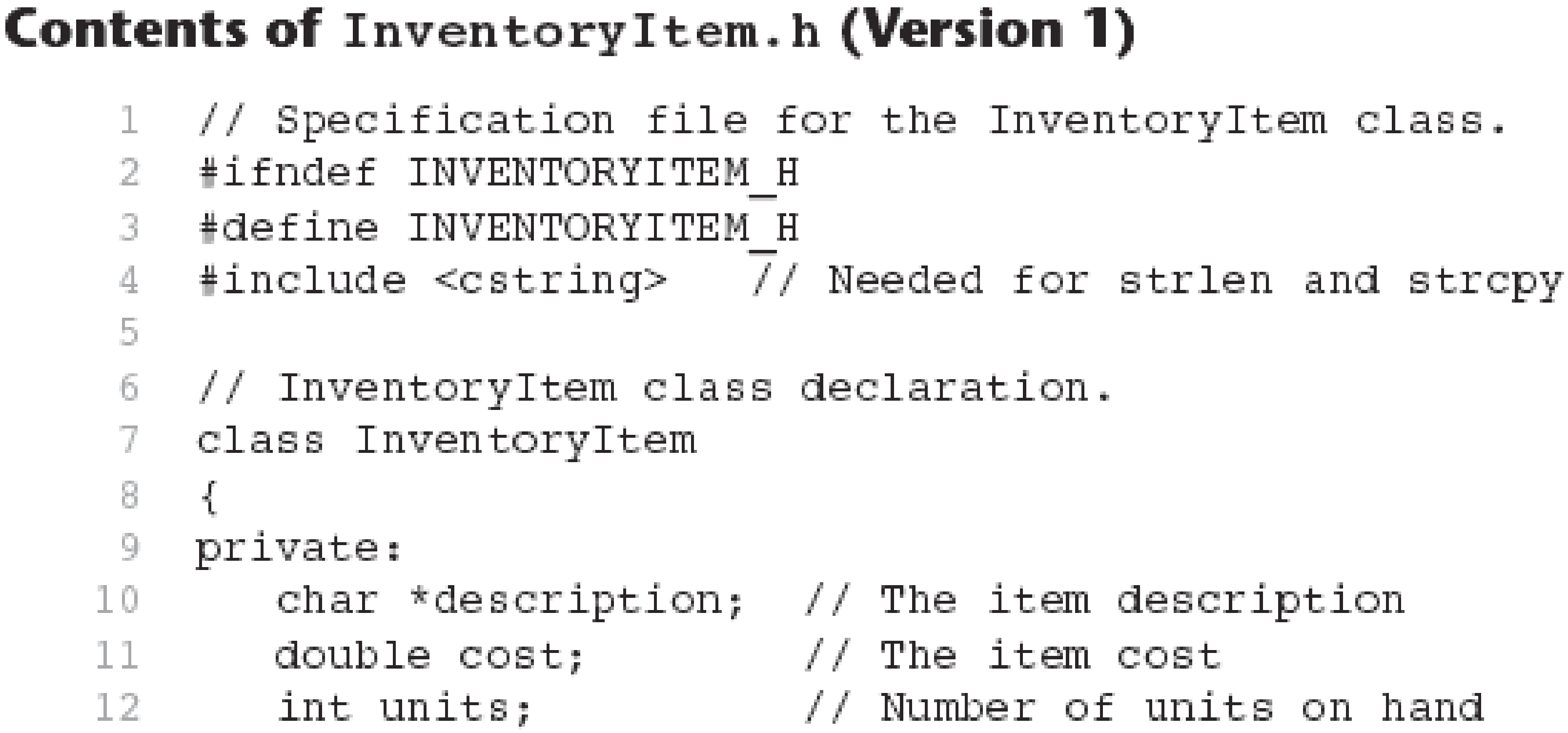
Copyright © 2012 Pearson Education, Inc.

Destructors

- Member function automatically called when an object is destroyed
- Destructor name is ~classname, *e.g.*,

```
~Rectangle
```

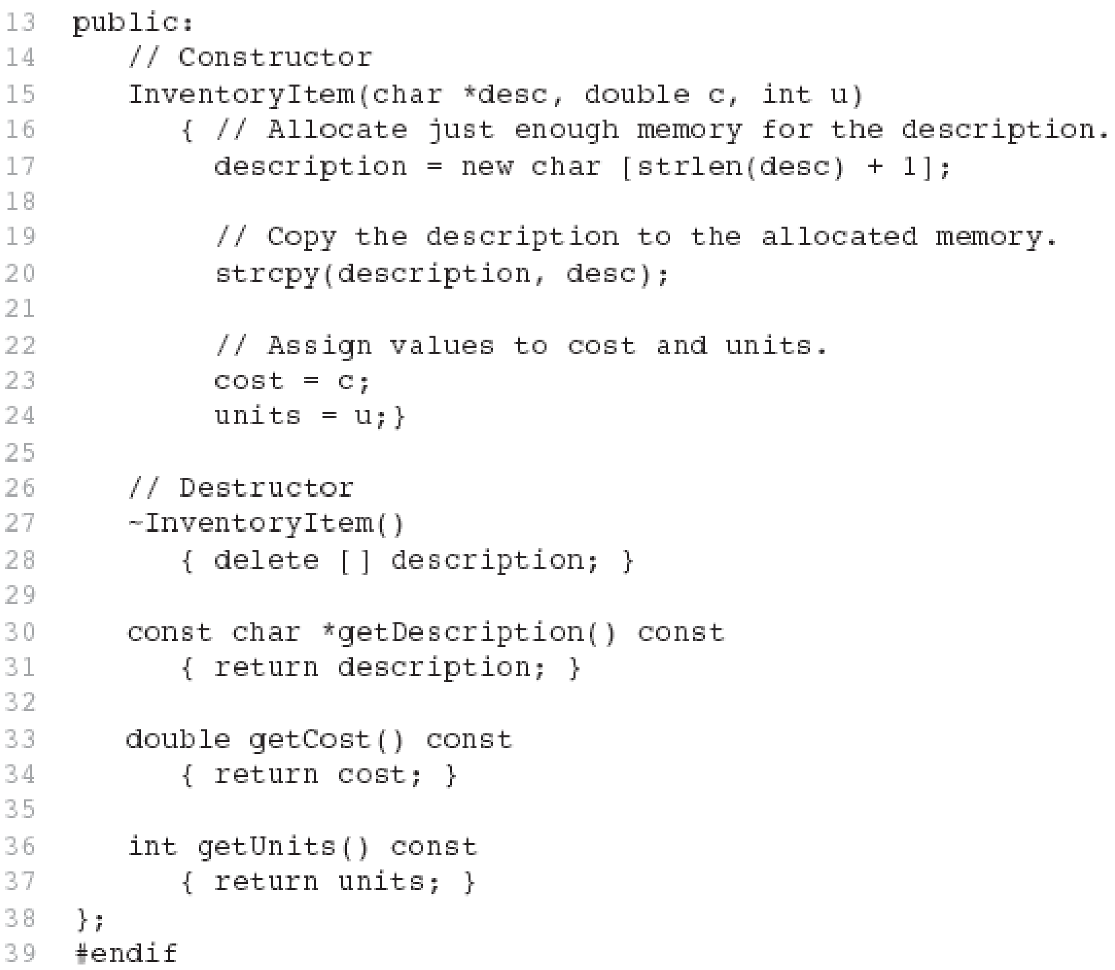
- Has no return type; takes no arguments
- Only one destructor per class, *i.e.*, it cannot be overloaded
- If constructor allocates dynamic memory, destructor should release it



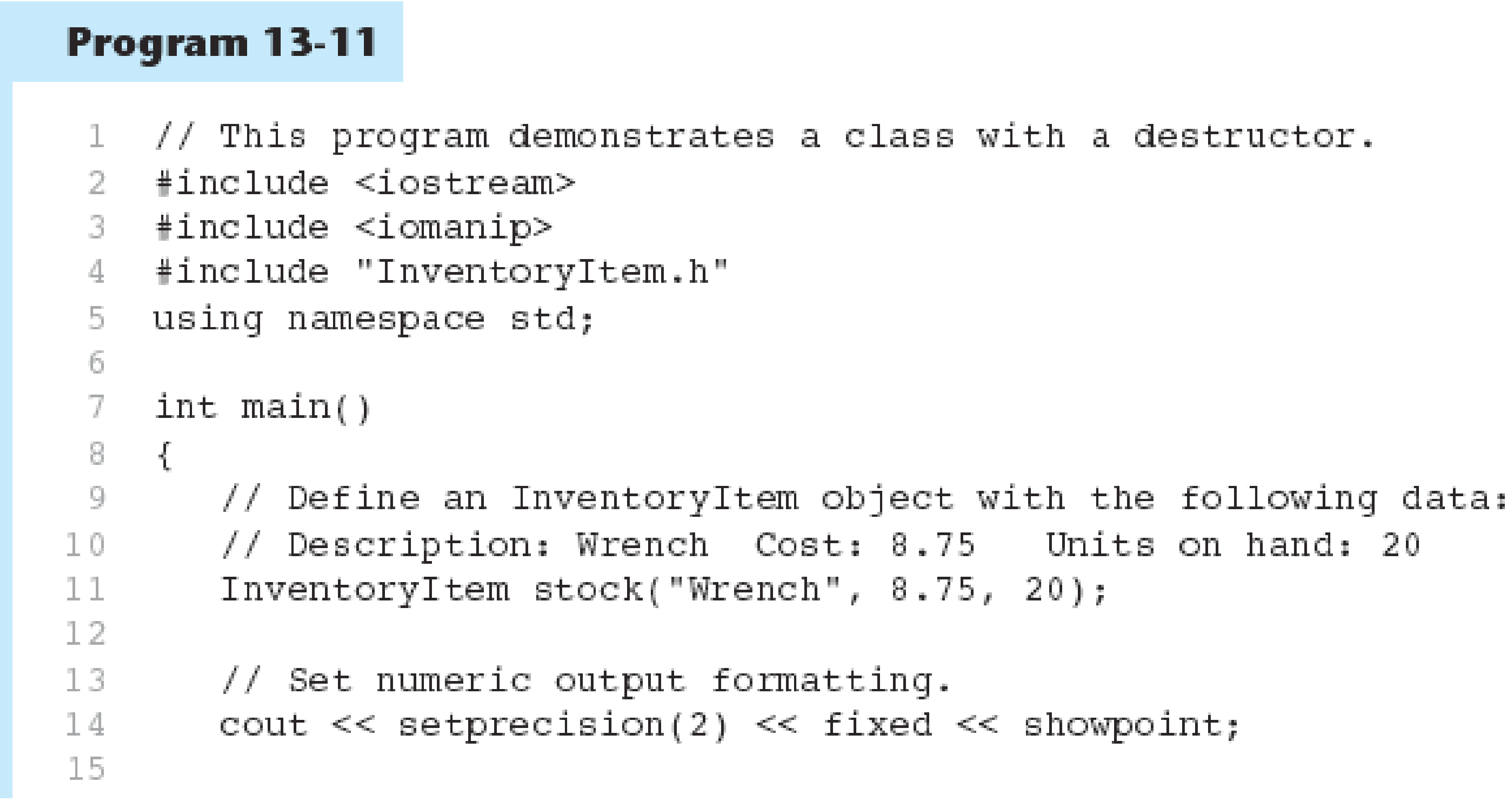
Contents of InventoryItem.h (Version 1)

```
// Specification file for the InventoryItem class.
#ifndef INVENTORYITEM_H
#define INVENTORYITEM_H
#include <cstring>   // Needed for strlen and strcpy

// InventoryItem class declaration.
class InventoryItem
{
private:
   char *description;  // The item description
   double cost;        // The item cost
   int units;          // Number of units on hand
```

Contents of InventoryItem.h Version1 (Continued)

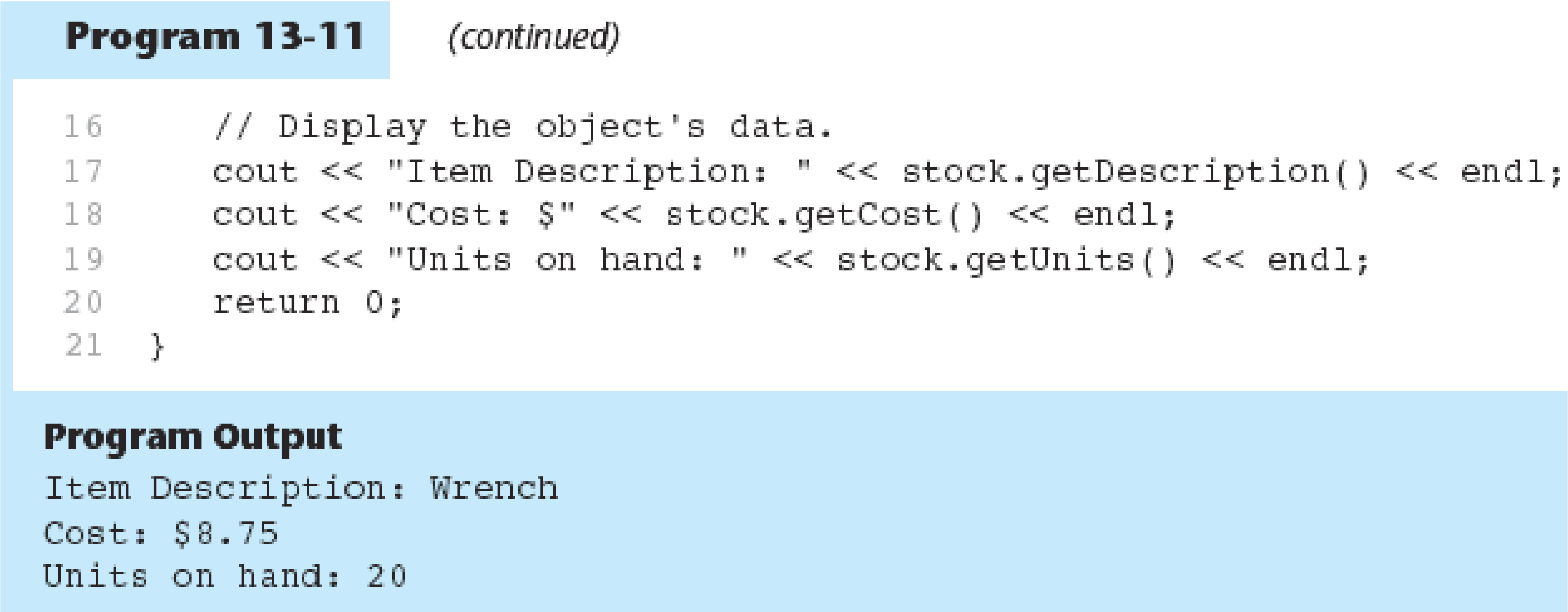


```
public:
   // Constructor
   InventoryItem(char *desc, double c, int u)
      { // Allocate just enough memory for the description.
        description = new char [strlen(desc) + 1];

        // Copy the description to the allocated memory.
        strcpy(description, desc);

        // Assign values to cost and units.
        cost = c;
        units = u;}

   // Destructor
   ~InventoryItem()
      { delete [] description; }

   const char *getDescription() const
      { return description; }

   double getCost() const
      { return cost; }

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



Program 13-11

```
// This program demonstrates a class with a destructor.
#include <iostream>
#include <iomanip>
#include "InventoryItem.h"
using namespace std;

int main()
{
   // Define an InventoryItem object with the following data:
   // Description: Wrench  Cost: 8.75   Units on hand: 20
   InventoryItem stock("Wrench", 8.75, 20);

   // Set numeric output formatting.
   cout << setprecision(2) << fixed << showpoint;

```



Program 13-11 *(continued)*

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

Program Output

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

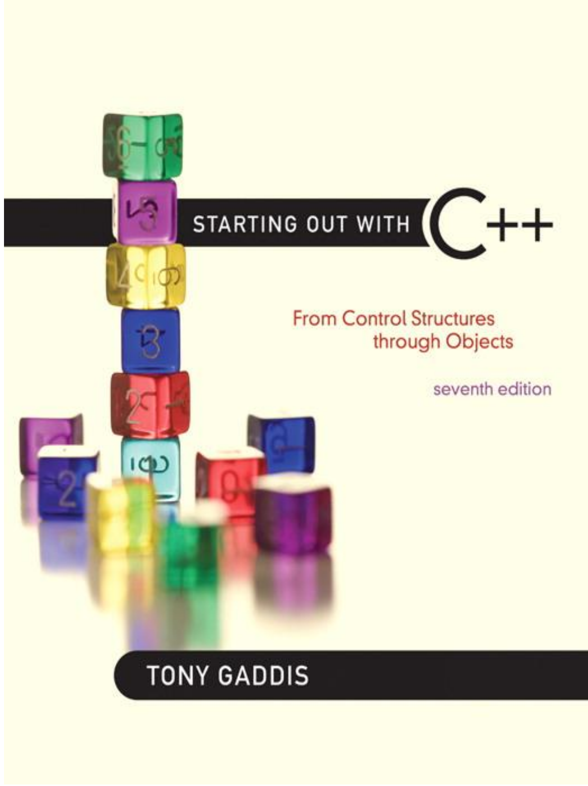
Constructors, Destructors, and Dynamically Allocated Objects

- 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

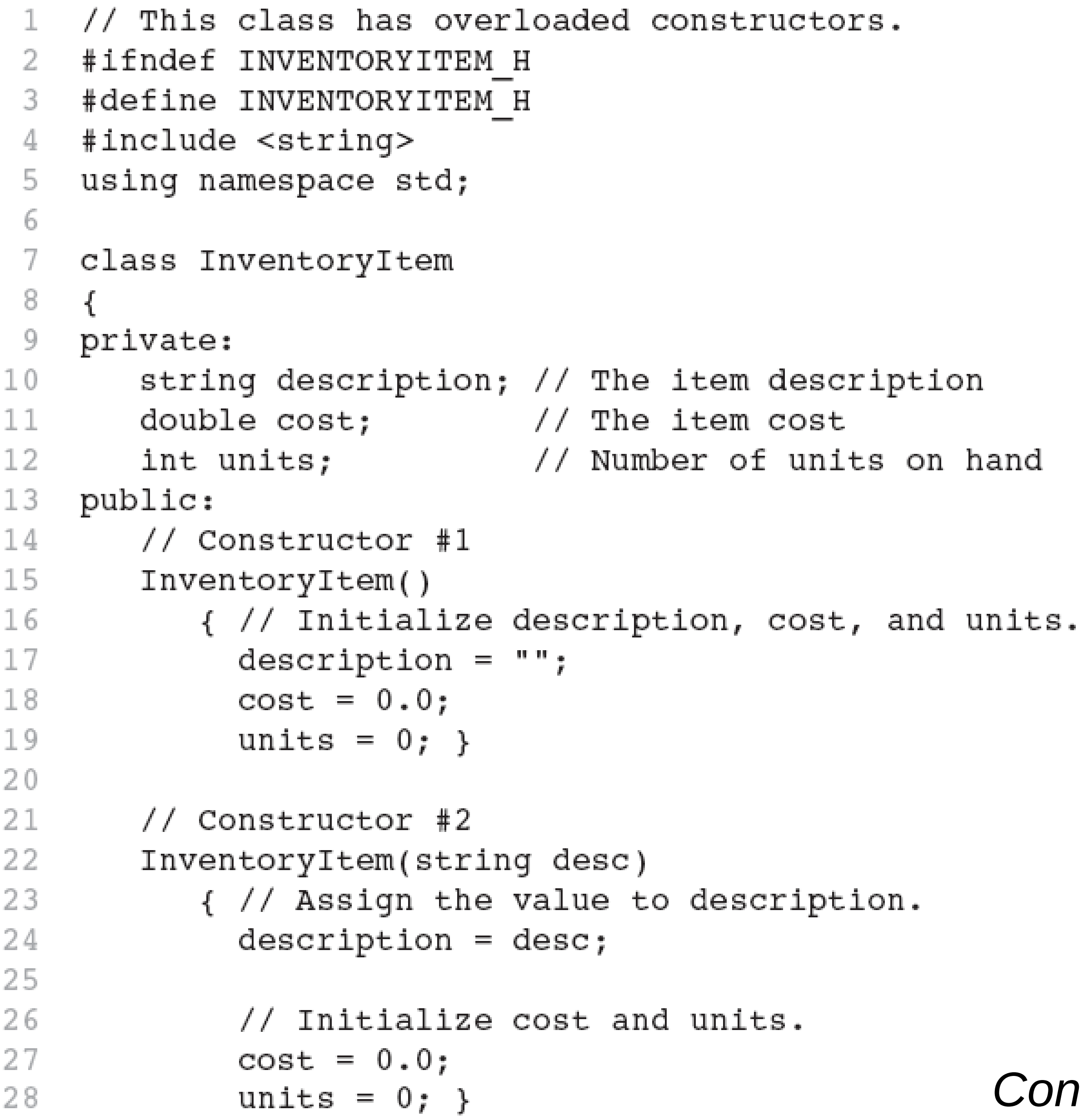
Copyright © 2012 Pearson Education, Inc.

Overloading Constructors

- A class can have more than one constructor

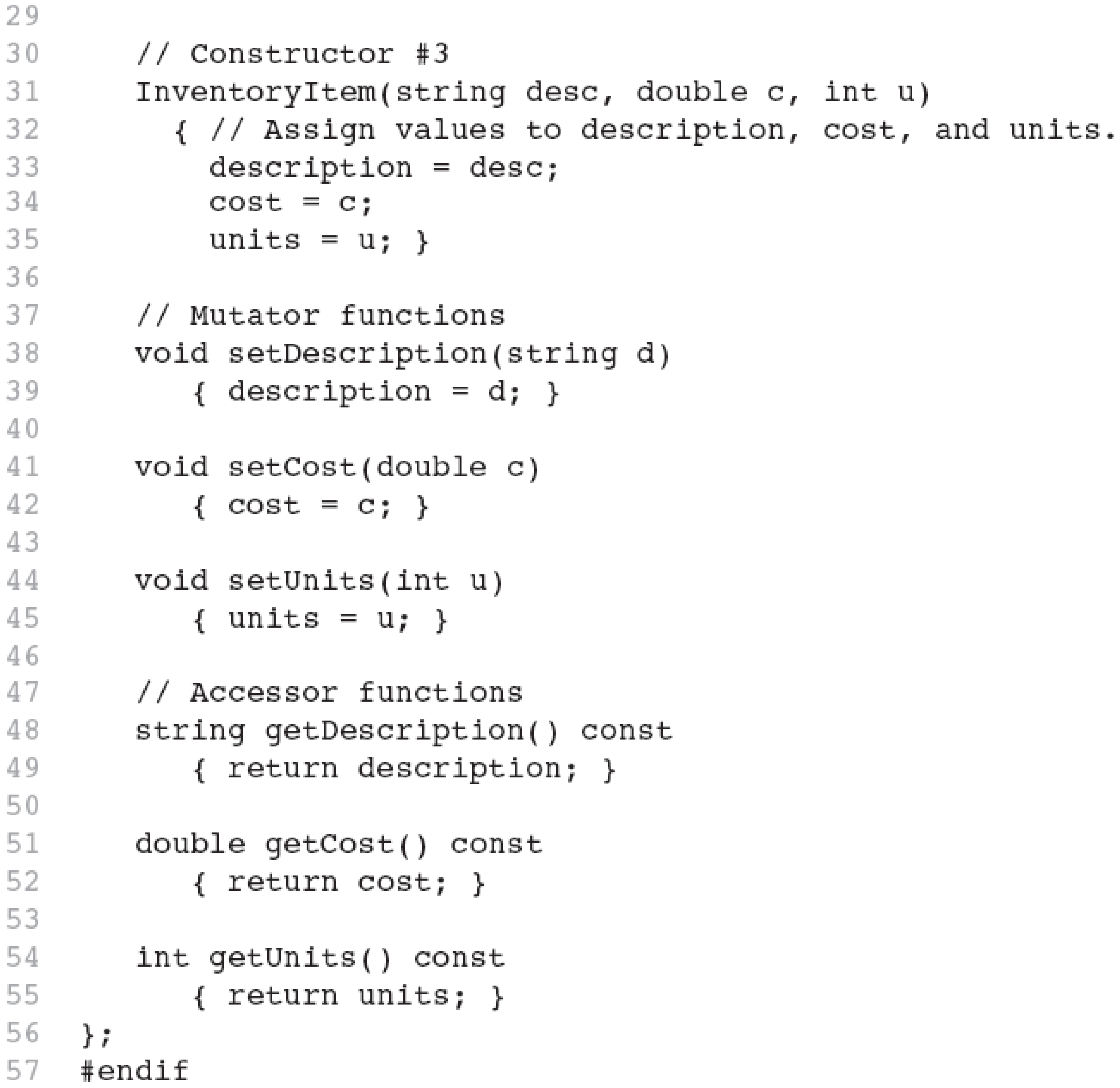
- Overloaded constructors in a class must have different parameter lists:

```
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
```

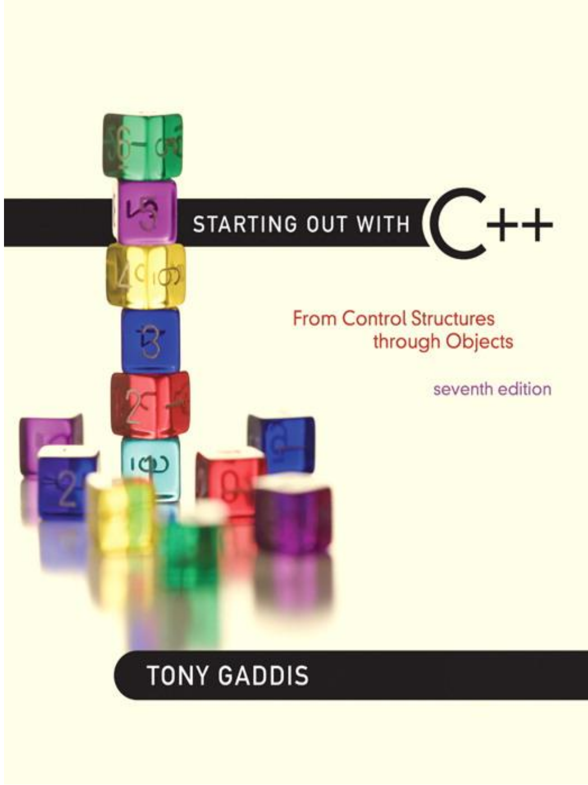
- Since a destructor takes no arguments, there can only be one destructor for a class

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

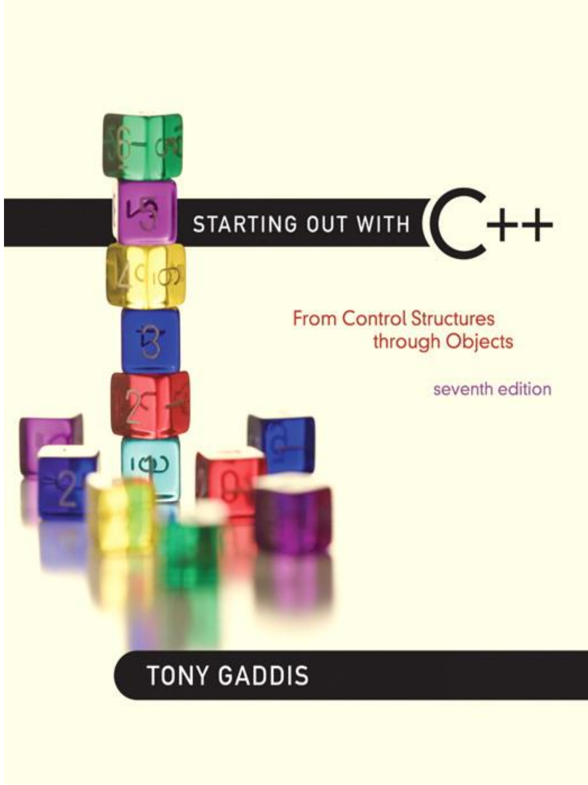
Copyright © 2012 Pearson Education, Inc.

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

Copyright © 2012 Pearson Education, Inc.

Arrays of Objects

- Objects can be the elements of an array:

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

- Default constructor for object is used when array is defined

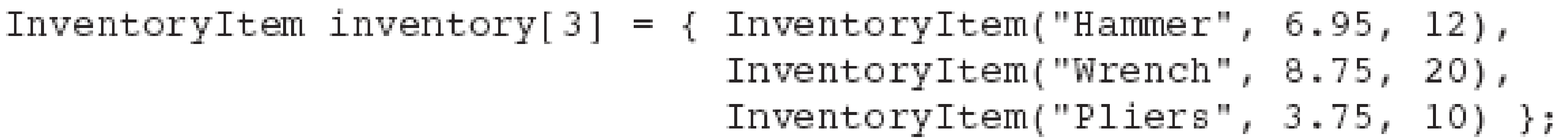
Arrays of Objects

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

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

Arrays of Objects

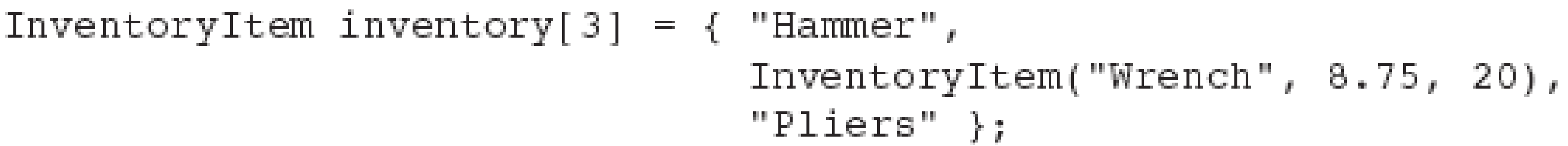
- If the constructor requires more than one argument, the initializer must take the form of a function call:



```
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:



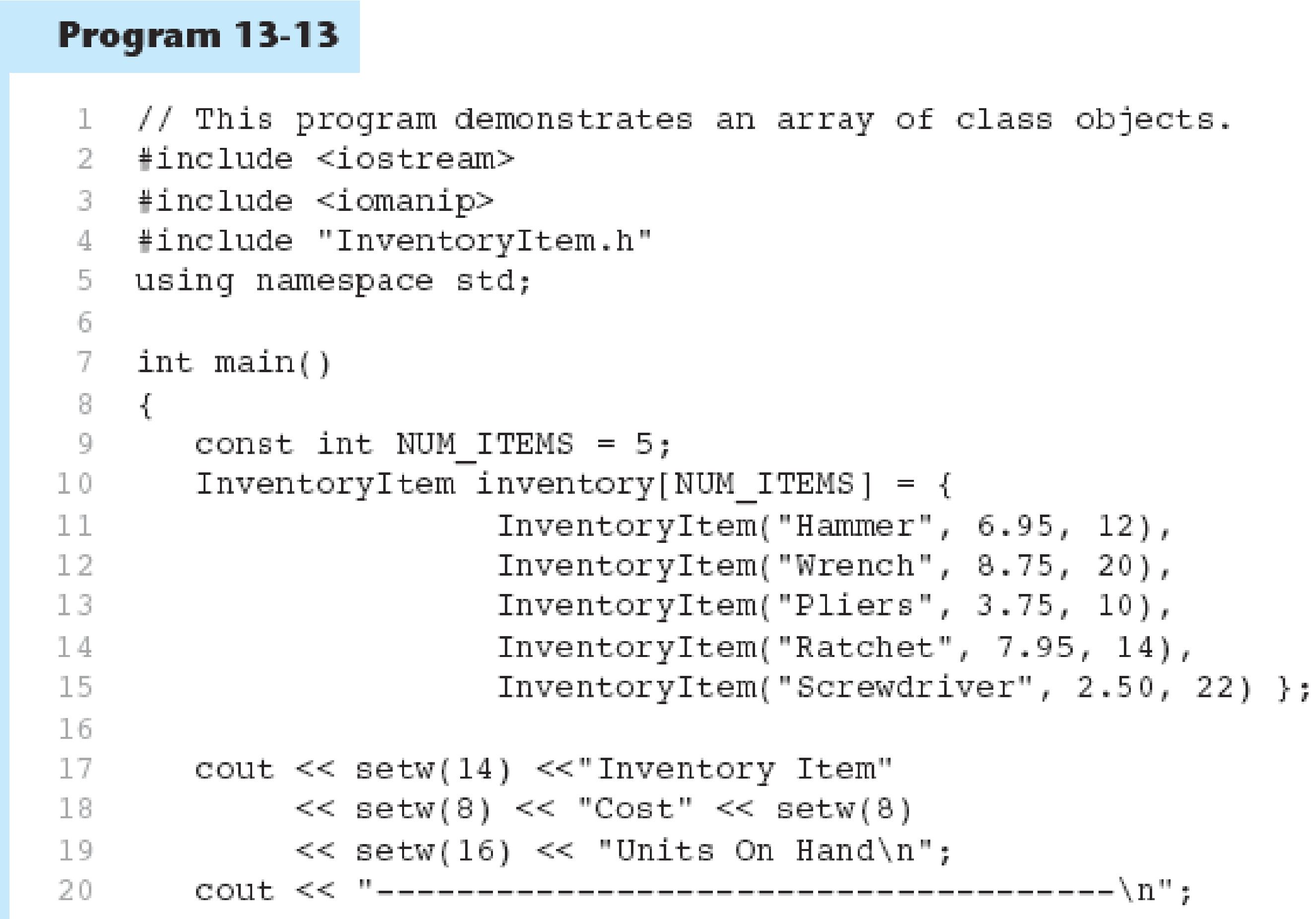
```
InventoryItem inventory[3] = { "Hammer",
                               InventoryItem("Wrench", 8.75, 20),
                               "Pliers" };
```

Accessing Objects in an Array

- Objects in an array are referenced using subscripts

- Member functions are referenced using dot notation:

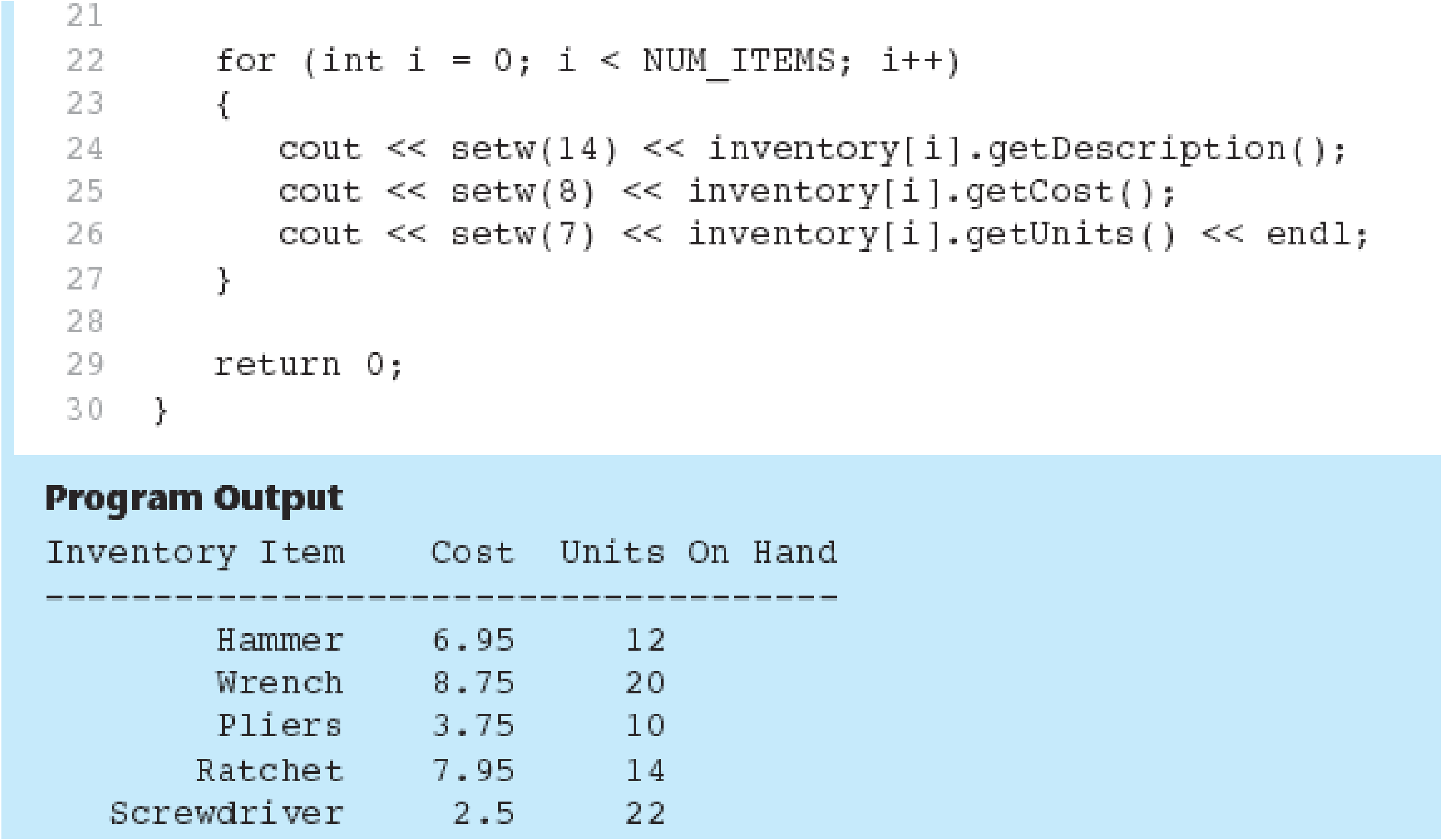
```
inventory[2].setUnits(30);
cout << inventory[2].getUnits();
```



Program 13-13

```cpp
// This program demonstrates an array of class objects.
#include <iostream>
#include <iomanip>
#include "InventoryItem.h"
using namespace std;

int main()
{
   const int NUM_ITEMS = 5;
   InventoryItem inventory[NUM_ITEMS] = {
                  InventoryItem("Hammer", 6.95, 12),
                  InventoryItem("Wrench", 8.75, 20),
                  InventoryItem("Pliers", 3.75, 10),
                  InventoryItem("Ratchet", 7.95, 14),
                  InventoryItem("Screwdriver", 2.50, 22) };

   cout << setw(14) <<"Inventory Item"
        << setw(8) << "Cost" << setw(8)
        << setw(16) << "Units On Hand\n";
   cout << "-------------------------------------\n";
```

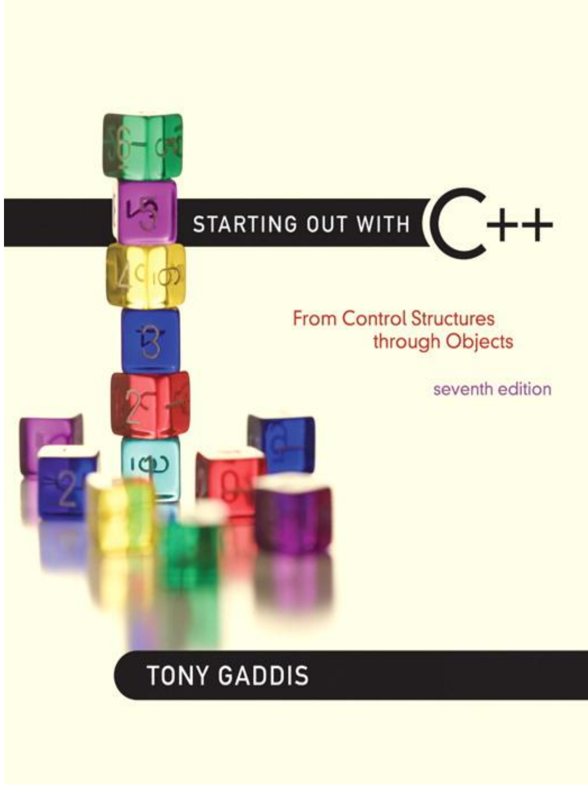
Program 13-3 (Continued)



```
   for (int i = 0; i < NUM_ITEMS; i++)
   {
      cout << setw(14) << inventory[i].getDescription();
      cout << setw(8) << inventory[i].getCost();
      cout << setw(7) << inventory[i].getUnits() << endl;
   }

   return 0;
}
```

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

Copyright © 2012 Pearson Education, Inc.

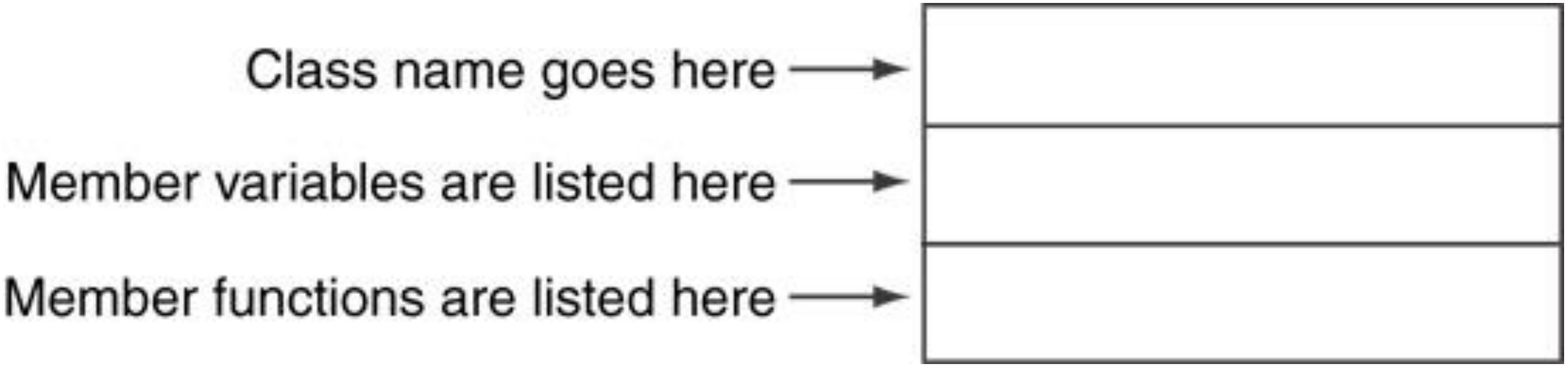
The Unified Modeling Language

- 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.

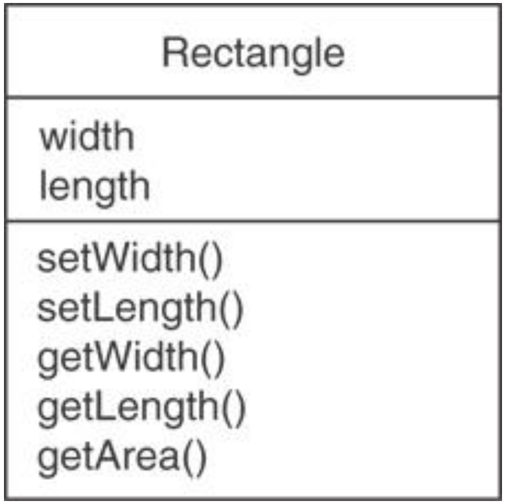


Class name goes here

Member variables are listed here

Member functions are listed here

Example: A Rectangle Class



Rectangle

width
length

setWidth()
setLength()
getWidth()
getLength()
getArea()

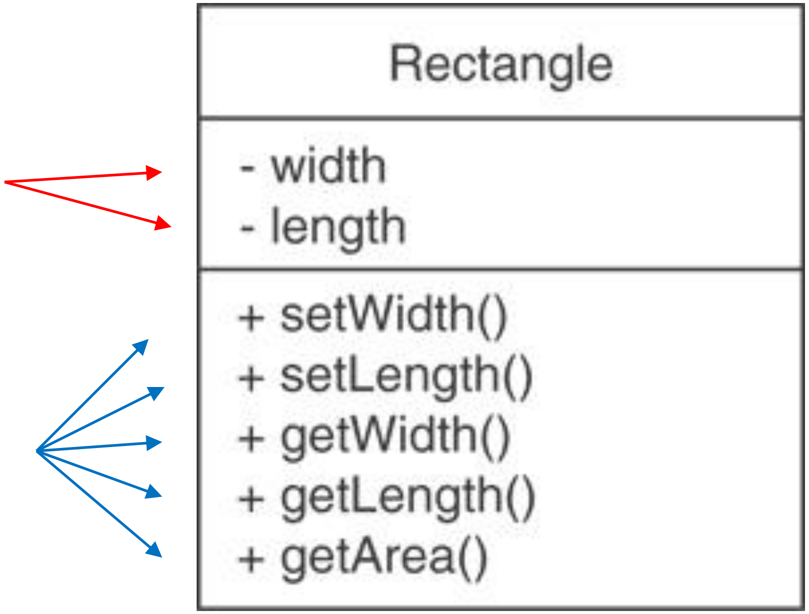
```
class Rectangle
{
   private:
      double width;
      double length;
   public:
      bool setWidth(double);
      bool setLength(double);
      double getWidth() const;
      double getLength() const;
      double getArea() const;
};
```

UML Access Specification Notation

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

These member variables are private.

These member functions are public.



Rectangle

- width
- length

+ setWidth()
+ setLength()
+ getWidth()
+ getLength()
+ getArea()

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.

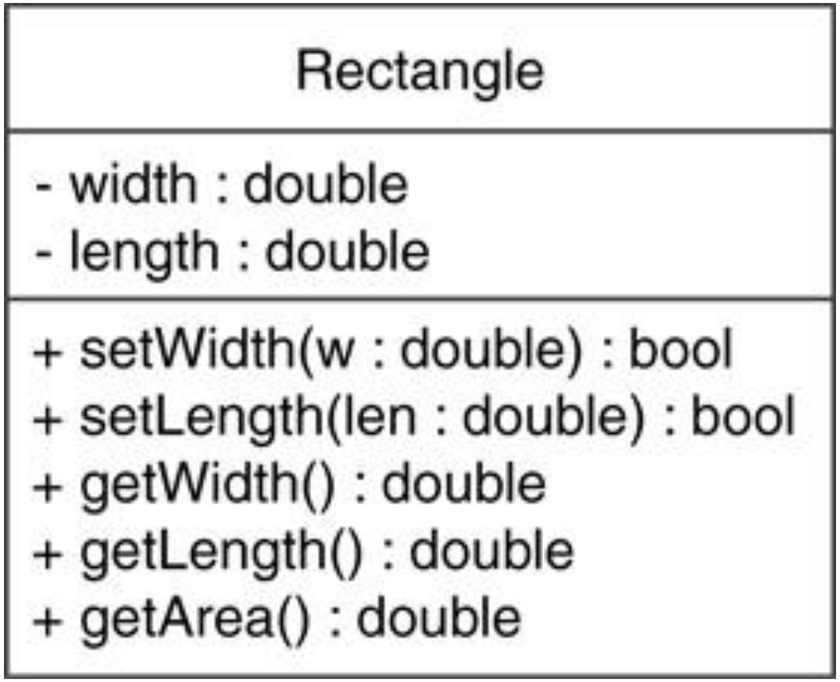
```
+ setWidth(w : double)
```

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



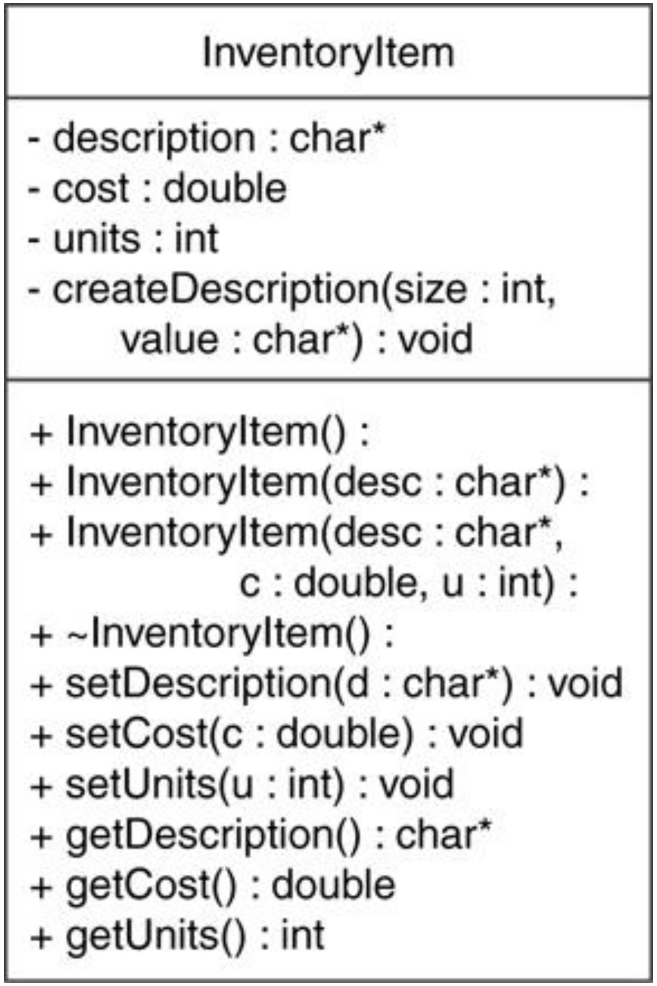
Rectangle

- width : double
- length : double

+ setWidth(w : double) : bool
+ setLength(len : double) : bool
+ getWidth() : double
+ getLength() : double
+ getArea() : double

Showing Constructors and Destructors

No return type listed for constructors or destructors



```
InventoryItem
- description : char*
- cost : double
- units : int
- createDescription(size : int,
      value : char*) : void
+ InventoryItem() :
+ InventoryItem(desc : char*) :
+ InventoryItem(desc : char*,
              c : double, u : int) :
+ ~InventoryItem() :
+ setDescription(d : char*) : void
+ setCost(c : double) : void
+ setUnits(u : int) : void
+ getDescription() : char*
+ getCost() : double
+ getUnits() : int
```

Constructors

Destructor