Introduction to Classes

Tony Gaddis

Ch 13

Procedural and Object-Oriented Programming

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

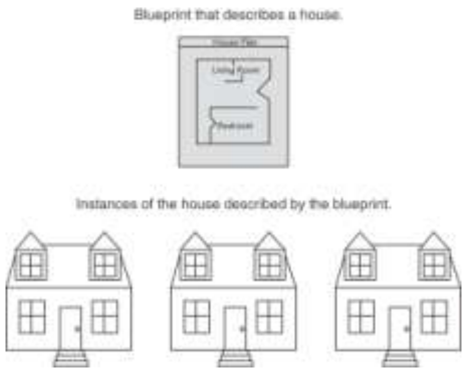
- Object-Oriented programming is based on the data and the functions that operate on it. Objects are instances of Classes that represent the data and its functions

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.
House Plan
Living Room
Bedroom
Instances of the house described by the blueprint.

Introduction to Classes

CONCEPT: In C++, the class is the construct primarily used to create objects.

A *class is similar to a structure. It is a data type defined by the programmer, consisting of* variables and functions.

Here is the general format of a class declaration:

```
class ClassName
{
      declaration;
     // ... more declarations
     // may follow...
};
```

Class Example

the following code declares a class named Rectangle with two member variables: width and length.

```
class Rectangle
{
    double width;
    double length;
};                                    // Don't forget the semicolon.
```

Access Specifiers

C++ provides the key words private and public which you may use in class declarations.

These key words are known as *access specifiers* because they specify how class members may be accessed.

```
class ClassName
{
   private:
         // Declarations of private
         // members appear here.
   public:
         // Declarations of public
         // members appear here.
};
```

Notice that the access specifiers are followed by a colon (**:**), and then followed by one or more member declarations.

Public Member Functions

To allow access to a class's private member variables width and length , you create public member functions that work with the private member variables.

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

←

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

In this declaration, the member variables width and length are declared as private, which means they can be accessed only by the class's member functions. The member functions, however, are declared as public, which means they can be called from statements outside the class.

←

Using const with Member Functions

Notice that the key word const appears in the declarations of the getWidth, getLength, and getArea member functions, as shown here:

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

When the key word const appears after the parentheses in a member function declaration, it specifies that the function will not change any data stored in the calling object. If you inadvertently write code in the function that changes the calling object's data, the compiler will generate an error.

Placement of public and private Members

There is no rule requiring you to declare private members before public members. The Rectangle class could be declared as follows:

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

Defining Member Functions

The Rectangle class declaration contains declarations or prototypes for five member functions: setWidth, setLength, getWidth, getLength, and getArea. The definitions of these functions are usually written outside the class declaration:

In each function definition, the following precedes the name of each function:

Rectangle::

The two colons are called the *scope resolution operator. When Rectangle:: appears* before the name of a function in a function header, it identifies the function as a member of the Rectangle class.

Here is the general format of the function header of any member function defined outside the declaration of a class:

```
ReturnType ClassName::functionName(ParameterList)
```

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. *
//**************************************************

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;
}
```

```

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

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

```

Accessors and Mutators

A member function that gets a value from a class's member variable but does not change it is known as an *accessor.*

A member function that stores a value in member variable or changes the value of member variable in some other way is known as a *mutator.*

In the Rectangle class, the member functions

getLength , *getWidth* and *getArea* are accessors or *getters*,

setLength and *setWidth* are mutators or *setters*.

Defining an Instance of a Class

CONCEPT: Class objects must be defined after the class is declared.

Like structure variables, class objects are not created in memory until they are defined.

This is because a class declaration by itself does not create an object, but is merely the description of an object. We can use it to create one or more objects, which are instances of the class.

Here is the general format of a simple object definition statement:

```
ClassName objectName;
```

For example, the following statement defines box as an object of the Rectangle class:

```
Rectangle box;
```

Defining a class object is called the *instantiation* of a class. In this statement, box is an *instance* of the Rectangle class.

Accessing an Object’s Members

The box object that we previously defined is an instance of the Rectangle class. Suppose we want to change the value in the box object’s width variable. To do so, we must use the box object to call the setWidth member function, as shown here:

```
box.setWidth(12.7);
```

Just as you use the dot operator to access a structure’s members, you use the dot operator to call a class’s member functions. Here are other examples of statements that use the box object to call member functions:

```
box.setLength(4.8);          // Set box's length to 4.8.
double x = box.getWidth();   // Assign box's width to x.
cout << box.getLength();     // Display box's length.
cout << box.getArea();       // Display box's area.
```

Program 13-1

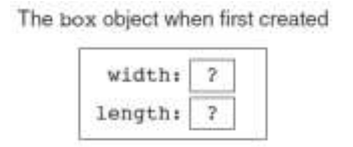
```
// This program demonstrates a simple class.
#include <iostream>
using namespace std;

// 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;
}
```

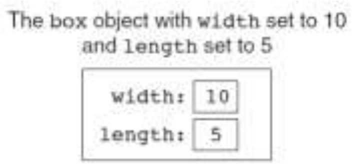
```

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

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;
}
```

```

//*****************************************************
// 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);

   // Display the rectang
   cout << "Here is the r
   cout << "Width: " << b
   cout << "Length: " << 
   cout << "Area: " << bo
   return 0;
}
```

```
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
```

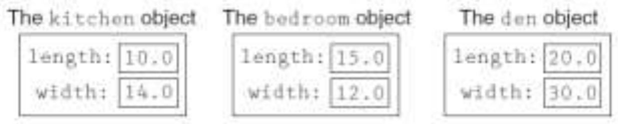


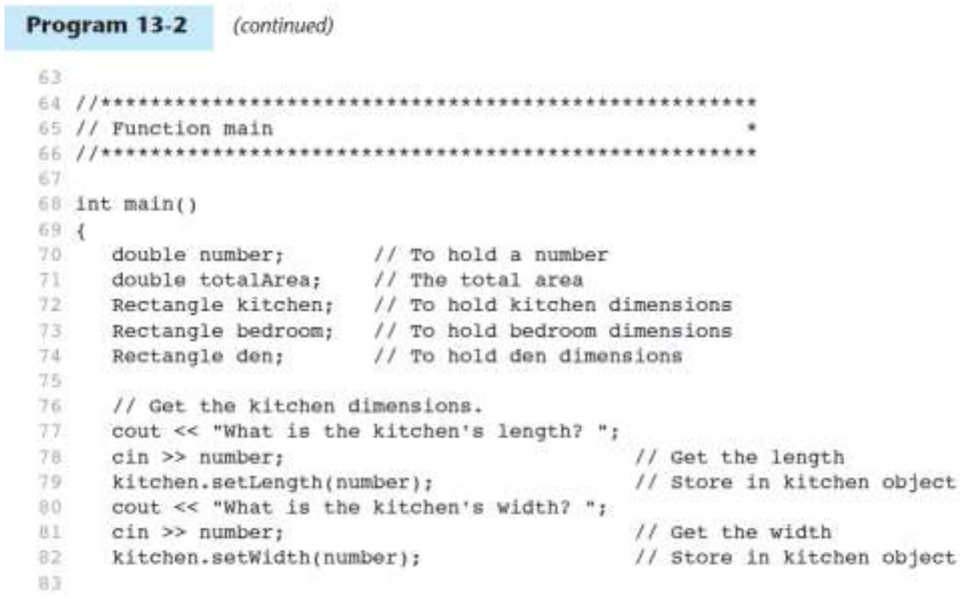


Modify the program to create three instances of the Rectangle class.

```
Rectangle kitchen;     // To hold kitchen dimensions
Rectangle bedroom;     // To hold bedroom dimensions
Rectangle den;         // To hold den dimensions
```

```
What is the kitchen's length? 10 [Enter]
What is the kitchen's width? 14 [Enter]
What is the bedroom's length? 15 [Enter]
What is the bedroom's width? 12 [Enter]
What is the den's length? 20 [Enter]
What is the den's width? 30 [Enter]
The total area of the three rooms is 920
```





Program 13-2 *(continued)*

```

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

int main()
{
   double number;       // To hold a number
   double totalArea;    // The total area
   Rectangle kitchen;   // To hold kitchen dimensions
   Rectangle bedroom;   // To hold bedroom dimensions
   Rectangle den;       // To hold den dimensions

   // Get the kitchen dimensions.
   cout << "What is the kitchen's length? ";
   cin >> number;                            // Get the length
   kitchen.setLength(number);                // Store in kitchen object
   cout << "What is the kitchen's width? ";
   cin >> number;                            // Get the width
   kitchen.setWidth(number);                 // Store in kitchen object

```

Rest of the code, omitted.

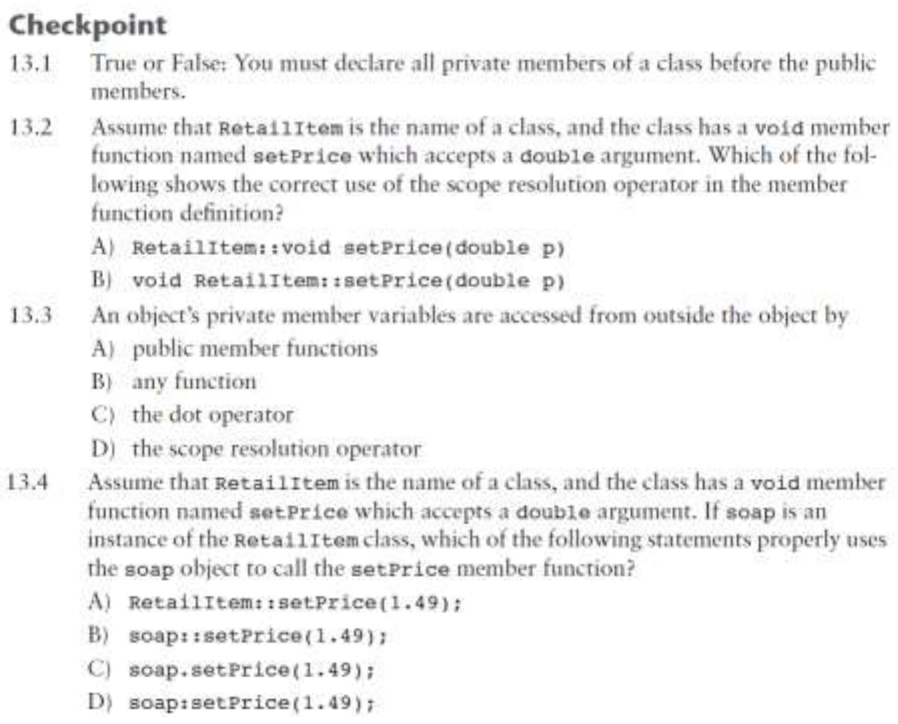
Pointers to Objects

Later
next 7 slides hidden

You can also define pointers to class objects. For example, the following statement defines a pointer variable named rectPtr:

```
Rectangle *rectPtr;
```

The rectPtr variable is not an object, but it can hold the address of a Rectangle object.



Checkpoint

13.1 True or False: You must declare all private members of a class before the public members.

13.2 Assume that `RetailItem` is the name of a class, and the class has a `void` member function named `setPrice` which accepts a `double` argument. Which of the following shows the correct use of the scope resolution operator in the member function definition?

A) `RetailItem::void setPrice(double p)`

B) `void RetailItem::setPrice(double p)`

13.3 An object's private member variables are accessed from outside the object by

A) public member functions

B) any function

C) the dot operator

D) the scope resolution operator

13.4 Assume that `RetailItem` is the name of a class, and the class has a `void` member function named `setPrice` which accepts a `double` argument. If `soap` is an instance of the `RetailItem` class, which of the following statements properly uses the `soap` object to call the `setPrice` member function?

A) `RetailItem::setPrice(1.49);`

B) `soap::setPrice(1.49);`

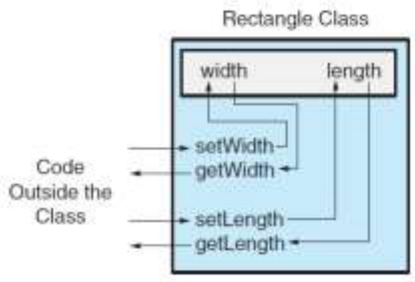
C) `soap.setPrice(1.49);`

D) `soap:setPrice(1.49);`

Why Have Private Members?

CONCEPT: In object-oriented programming, an object should protect its important data by making it private and providing a public interface to access that data.

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



13.5 Focus on Software Engineering: Separating Class Specification from Implementation

CONCEPT: Usually class declarations are stored in their own header .h files.
Member function definitions are stored in their own .cpp files.

Typically, program components are stored in the following fashion:

- Class declarations are stored in their own header files. A header file that contains a class declaration is called a *class specification file.* The name of the class specification file is usually the same as the name of the class, with a .h extension. For example, the Rectangle class would be declared in the file Rectangle.h.

- The member function definitions for a class are stored in a separate .cpp file called the *class implementation file.* The file usually has the same name as the class, with the .cpp extension. For example, the Rectangle class's member functions would be defined in the file Rectangle.cpp.

- Any program that uses the class should #include the class's header file. The class's .cpp file (that which contains the member function definitions) should be compiled and linked with the main program. This process can be automated with a project or make utility.

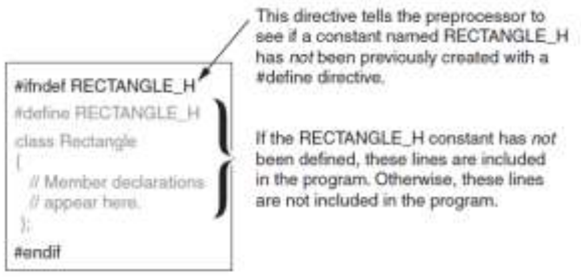
Contents of `Rectangle.h` (Version 1)

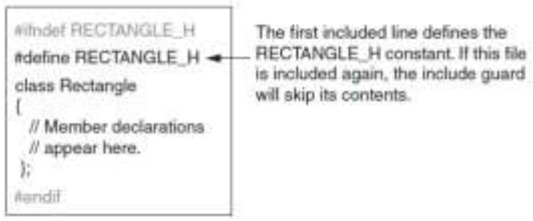
```
// Specification file for the Rectangle class.
#ifndef RECTANGLE_H
#define RECTANGLE_H

// 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;
};

#endif
```

This file also introduces two new preprocessor directives: #ifndef and #endif. The #ifndef directive that appears in line 2 is called an *include guard.*

It prevents the header file from accidentally being included more than once. When your main program file has an #include directive for a header file, there is always the possibility that the header file will have an #include directive for a second header file. If your main program file also has an #include directive for the second header file, then the preprocessor will include the second header file twice.

Unless an include guard has been written into the second header file, an error will occur because the compiler will process the declarations in the second header file twice.


#ifndef RECTANGLE_H
#define RECTANGLE_H
class Rectangle
{
// Member declarations
// appear here.
};
#endif
This directive tells the preprocessor to see if a constant named RECTANGLE_H has *not* been previously created with a #define directive.
If the RECTANGLE_H constant has *not* been defined, these lines are included in the program. Otherwise, these lines are not included in the program.

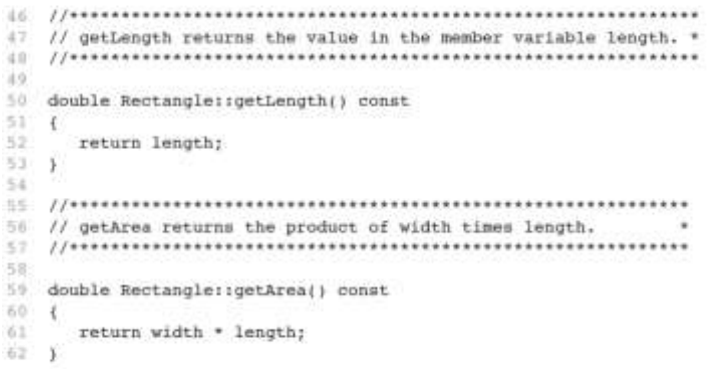

#ifndef RECTANGLE_H
#define RECTANGLE_H
class Rectangle
{
// Member declarations
// appear here.
};
#endif
The first included line defines the RECTANGLE_H constant. If this file is included again, the include guard will skip its contents.

The implementation file for the Rectangle class is Rectangle.cpp.

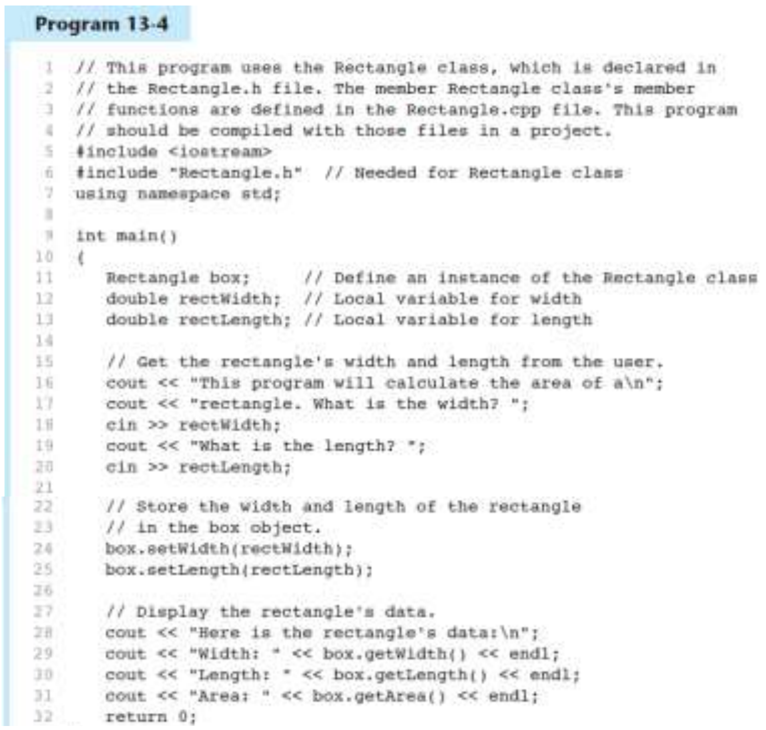
Contents of `Rectangle.cpp` (Version 1)

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

//************************************************************
// 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);
   }
}

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

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

```



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

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

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

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



Program 13-4

```
// This program uses the Rectangle class, which is declared in
// the Rectangle.h file. The member Rectangle class's member
// functions are defined in the Rectangle.cpp file. This program
// should be compiled with those files in a project.
#include <iostream>
#include "Rectangle.h"  // Needed for Rectangle class
using namespace std;

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);

   // 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;
```

To create an executable program from this file, the following steps must be taken:

- The implementation file, Rectangle.cpp, must be compiled. Rectangle.cpp is not a complete program, so you cannot create an executable file from it alone. Instead, you compile Rectangle.cpp to an object file which contains the compiled code for the Rectangle class. This file would typically be named Rectangle.obj. g++ -o Rectangle.cpp

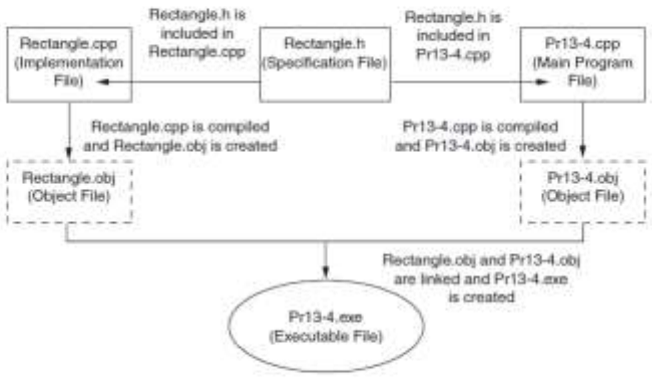
- The main program file, Pr13-4.cpp, must be compiled. This file is not a complete program either, because it does not contain any of the implementation code for the Rectangle class. So, you compile this file to an object file such as Pr13-4.obj. g++ -0 Pr13-4.cpp

- The object files, Pr13-4.obj and Rectangle.obj, are linked together to create an executable file, which would be named something like Pr13-4.exe. g++ -o Pr13-4 Rectangle.o Pr13-4.o

```
g++ -o myProgram thing.cpp main.cpp
```

This command compiles and links (since -c not used) the code files "thing.cpp" and "main.cpp" together into the executable program called "myProgram".

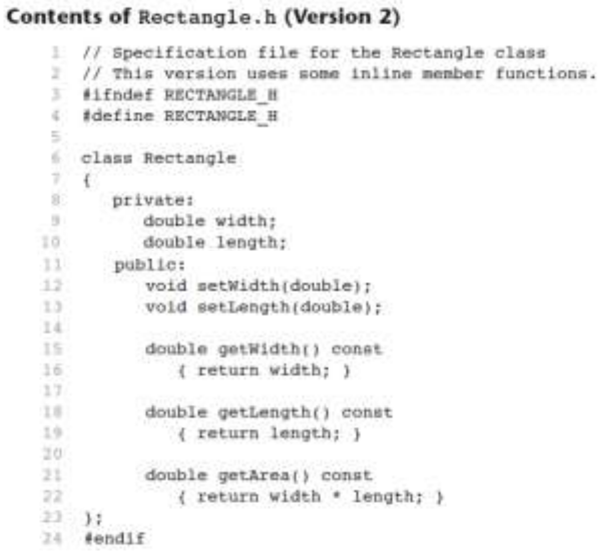
The exact details on how these steps take place are different for each C++ development system.


Rectangle.cpp (Implementation File)
Rectangle.h is included in Rectangle.cpp
Rectangle.h (Specification File)
Rectangle.h is included in Pr13-4.cpp
Pr13-4.cpp (Main Program File)
Rectangle.cpp is compiled and Rectangle.obj is created
Pr13-4.cpp is compiled and Pr13-4.obj is created
Rectangle.obj (Object File)
Pr13-4.obj (Object File)
Rectangle.obj and Pr13-4.obj are linked and Pr13-4.exe is created
Pr13-4.exe (Executable File)

Inline Member Functions

CONCEPT: When the body of a member function is written inside a class declaration, it is declared inline.

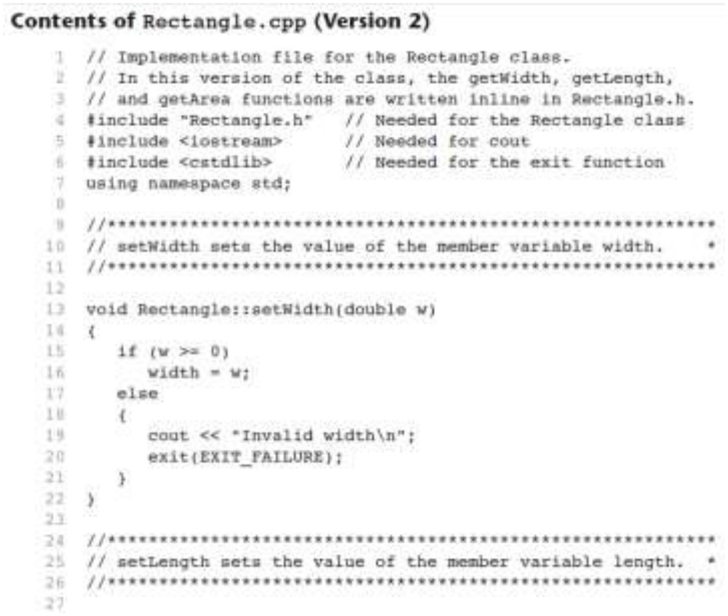
When the body of a member function is small, it is usually more convenient to place the function's definition, instead of its prototype, in the class declaration. For example, in the Rectangle class the member functions getWidth, getLength, and getArea each have only one statement. The Rectangle class could be revised as shown in the following listing.



Contents of Rectangle.h (Version 2)

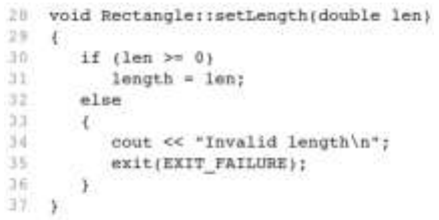
```
// Specification file for the Rectangle class
// This version uses some inline member functions.
#ifndef RECTANGLE_H
#define RECTANGLE_H

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
```

Notice that the getWidth, getLength, and getArea functions are declared inline, but the setWidth and setLength functions are not. They are still defined outside the class declaration.



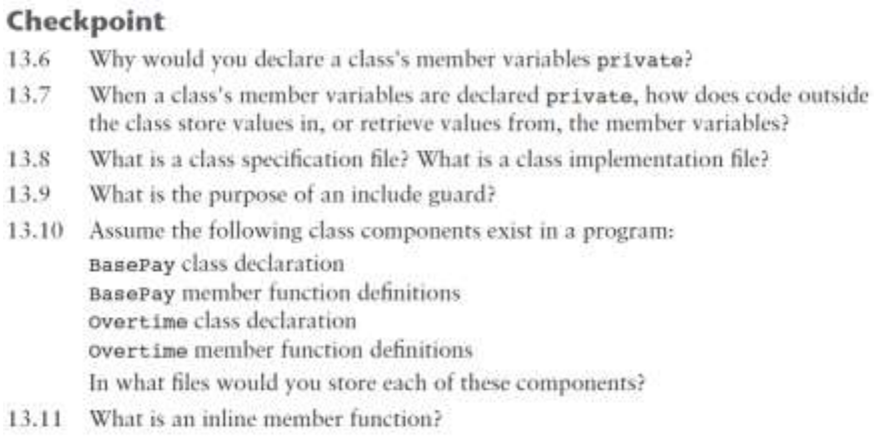
Contents of Rectangle.cpp (Version 2)

```
// Implementation file for the Rectangle class.
// In this version of the class, the getWidth, getLength,
// and getArea functions are written inline in Rectangle.h.
#include "Rectangle.h"   // Needed for the Rectangle class
#include <iostream>      // Needed for cout
#include <cstdlib>       // Needed for the exit function
using namespace std;

//***********************************************************
// 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);
   }
}
```

Inline functions are compiled differently than other functions. In the executable code, inline functions aren't "called" in the conventional sense. In a process known as *inline expansion*, the compiler replaces the call to an inline function with the code of the function itself. This means that the overhead needed for a conventional function call isn't necessary for an inline function, and can result in improved performance.



Checkpoint

13.6 Why would you declare a class's member variables `private`?

13.7 When a class's member variables are declared `private`, how does code outside the class store values in, or retrieve values from, the member variables?

13.8 What is a class specification file? What is a class implementation file?

13.9 What is the purpose of an include guard?

13.10 Assume the following class components exist in a program:
`BasePay` class declaration
`BasePay` member function definitions
`Overtime` class declaration
`Overtime` member function definitions
In what files would you store each of these components?

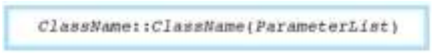
13.11 What is an inline member function?

Constructors

CONCEPT: A constructor is a member function that is automatically called when a class object is created.

A constructor is a member function that has the same name as the class. It is automatically called when the object is created in memory, or instantiated. It is helpful to think of constructors as initialization routines. They are very useful for initializing member variables or performing other setup operations.

The constructor's function header looks different than that of a regular member function. There is no return type—not even void. This is because constructors are not executed by explicit function calls and cannot return a value. The function header of a constructor's external definition takes the following form:



```
ClassName::ClassName(ParameterList)
```

Program 13-5

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

// Demo class declaration.

class Demo
{
public:
   Demo();     // Constructor
};

Demo::Demo()
{
   cout << "Welcome to the constructor!\n";
}

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

int main()
{
   Demo demoObject;  // Define a Demo object;

   cout << "This program demonstrates an object\n";
   cout << "with a constructor.\n";
   return 0;
}
```

```
Welcome to the constructor!
This program demonstrates an object
with a constructor.
```

The following code shows a better version of the Rectangle class, equipped with a constructor. The constructor initializes both width and length to 0.0.

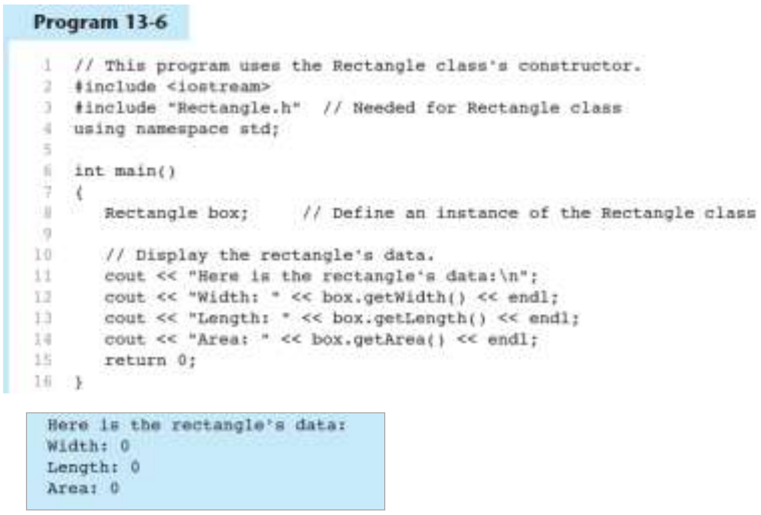
Contents of `Rectangle.h` (Version 3)

```cpp
// 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;
}
```

```

//************************************************************
// 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;
}
```

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

The Default Constructor

All of the examples we have looked at in this section demonstrate default constructors.

A *default constructor is* a constructor that takes no arguments. Like regular functions, constructors may accept arguments, have default arguments, be declared inline, and be overloaded.

If you write a class with no constructor whatsoever, when the class is compiled C++ will automatically write a default constructor that does nothing. For example, the first version of the Rectangle class had no constructor; so, when the class was compiled C++ generated the following constructor:

```
Rectangle::Rectangle()
{ }
```

Default Constructors and Dynamically Allocated Objects

Earlier we discussed how class objects may be dynamically allocated in memory. For example, assume the following pointer is defined in a program:

```
Rectangle *rectPtr;
```

This statement defines rectPtr as a Rectangle pointer. It can hold the address of any Rectangle object. But because this statement does not actually create a Rectangle object, the constructor does not execute.

Suppose we use the pointer in a statement that dynamically allocates a Rectangle object, as shown in the following code.

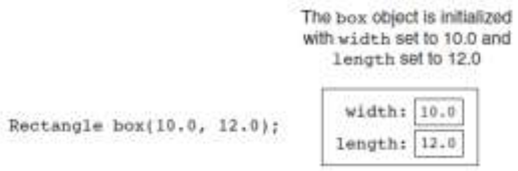
```
rectPtr = new Rectangle;
```

This statement creates a Rectangle object. When the Rectangle object is created by the new operator, its default constructor is automatically executed.

Passing Arguments to Constructors

CONCEPT: A constructor can have parameters, and can accept arguments when an object is created.

Constructors may accept arguments in the same way as other functions. When a class has a constructor that accepts arguments, you can pass initialization values to the constructor when you create an object. For example, the following code shows yet another version of the Rectangle class. This version has a constructor that accepts arguments for the rectangle's width and length.


The box object is initialized with width set to 10.0 and length set to 12.0
Rectangle box(10.0, 12.0);
width: 10.0
length: 12.0

Contents of `Rectangle.h` (Version 4)

```
// 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(double, double);   // 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 4)

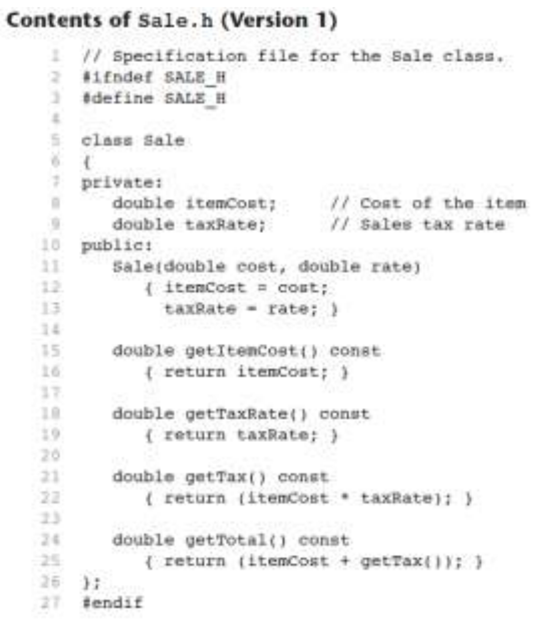
```
// Implementation file for the Rectangle class.
// This version has a constructor that accepts arguments.
#include "Rectangle.h"   // Needed for the Rectangle class
#include <iostream>      // Needed for cout
#include <cstdlib>       // Needed for the exit function
using namespace std;

//***********************************************************
// The constructor accepts arguments for width and length.  *
//***********************************************************

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

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

void Rectangle::setWidth(double w)
{
   if (w >= 0)
      width = w;
```

Program 13-7

```
// This program calls the Rectangle class constructor.
#include <iostream>
#include <iomanip>
#include "Rectangle.h"
using namespace std;

int main()
{
   double houseWidth,   // To hold the room width
          houseLength;  // To hold the room length

   // Get the width of the house.
   cout << "In feet, how wide is your house? ";
   cin >> houseWidth;

   // Get the length of the house.
   cout << "In feet, how long is your house? ";
   cin >> houseLength;

   // Create a Rectangle object.
   Rectangle house(houseWidth, houseLength);

   // Display the house's width, length, and area.
   cout << setprecision(2) << fixed;
   cout << "The house is " << house.getWidth()
        << " feet wide.\n";
   cout << "The house is " << house.getLength()
        << " feet long.\n";
   cout << "The house is " << house.getArea()
        << " square feet in area.\n";
   return 0;
}
```

```
In feet, how wide is your house? 30 [Enter]
In feet, how long is your house? 60 [Enter]
The house is 30.00 feet wide.
The house is 60.00 feet long.
The house is 1800.00 square feet in area.
```

The following code shows another example: the Sale class. This class might be used in a retail environment where sales transactions take place. An object of the Sale class represents the sale of an item.

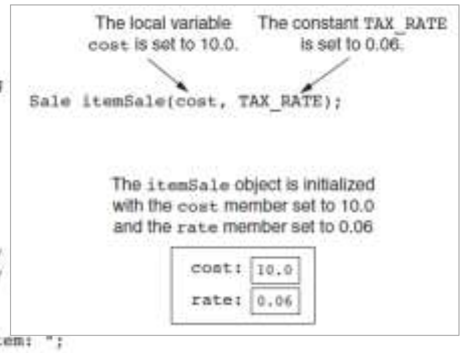


Contents of `Sale.h` (Version 1)

```
// Specification file for the Sale class.
#ifndef SALE_H
#define SALE_H

class Sale
{
private:
   double itemCost;     // Cost of the item
   double taxRate;      // Sales tax rate
public:
   Sale(double cost, double rate)
      { itemCost = cost;
        taxRate = rate; }

   double getItemCost() const
      { return itemCost; }

   double getTaxRate() const
      { return taxRate; }

   double getTax() const
      { return (itemCost * taxRate); }

   double getTotal() const
      { return (itemCost + getTax()); }
};
#endif
```

Program 13-8

```
// This program demonstrates passing
#include <iostream>
#include <iomanip>
#include "Sale.h"
using namespace std;

int main()
{
   const double TAX_RATE = 0.06;  //
   double cost;                   //

   // Get the cost of the item.
   cout << "Enter the cost of the item: ";
   cin >> cost;

   // Create a Sale object for this transaction.
   Sale itemSale(cost, TAX_RATE);

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

   // Display the sales tax and total.
   cout << "The amount of sales tax is $"
        << itemSale.getTax() << endl;
   cout << "The total of the sale is $";
   cout << itemSale.getTotal() << endl;
   return 0;
}
```


The local variable cost is set to 10.0.
The constant TAX_RATE is set to 0.06.
Sale itemSale(cost, TAX_RATE);
The itemSale object is initialized with the cost member set to 10.0 and the rate member set to 0.06
cost: 10.0
rate: 0.06

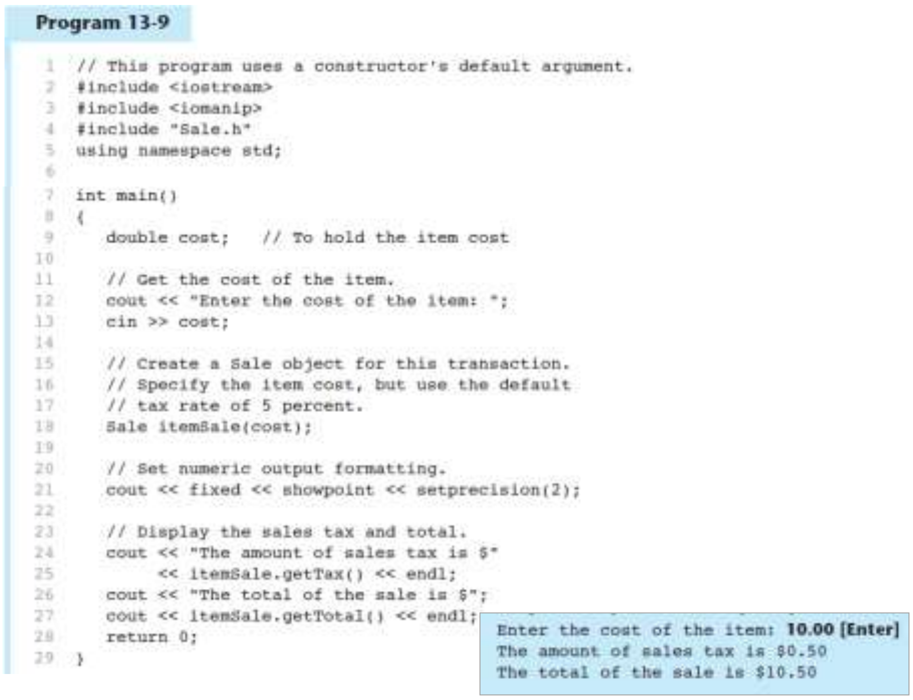
```
Enter the cost of the item: 10.00 [Enter]
The amount of sales tax is $0.60
The total of the sale is $10.60
```

Using Default Arguments with Constructors

Like other functions, constructors may have default arguments.

Contents of `Sale.h` (Version 2)

```
// This version of the Sale class uses a default argument
// in the constructor.
#ifndef SALE_H
#define SALE_H

class Sale
{
private:
   double itemCost;     // Cost of the item
   double taxRate;      // Sales tax rate
public:
   Sale(double cost, double rate = 0.05)
      { itemCost = cost;
        taxRate = rate; }

   double getItemCost() const
      { return itemCost; }

   double getTaxRate() const
      { return taxRate; }

   double getTax() const
      { return (itemCost * taxRate); }

   double getTotal() const
```



Program 13-9

```
// This program uses a constructor's default argument.
#include <iostream>
#include <iomanip>
#include "Sale.h"
using namespace std;

int main()
{
   double cost;   // To hold the item cost

   // Get the cost of the item.
   cout << "Enter the cost of the item: ";
   cin >> cost;

   // Create a Sale object for this transaction.
   // Specify the item cost, but use the default
   // tax rate of 5 percent.
   Sale itemSale(cost);

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

   // Display the sales tax and total.
   cout << "The amount of sales tax is $"
        << itemSale.getTax() << endl;
   cout << "The total of the sale is $";
   cout << itemSale.getTotal() << endl;
   return 0;
}
```

```
Enter the cost of the item: 10.00 [Enter]
The amount of sales tax is $0.50
The total of the sale is $10.50
```

More About the Default Constructor

If a constructor has default arguments for all its parameters, it can be called with no explicit arguments. It then becomes the default constructor. For example, suppose the constructor for the Sale class had been written as the following:

```
Sale(double cost = 0.0, double rate = 0.05)
{ itemCost = cost;
taxRate = rate; }
```

This constructor has default arguments for each of its parameters. As a result, the constructor can be called with no arguments, as shown here:

```
Sale itemSale;
```

Classes with No Default Constructor

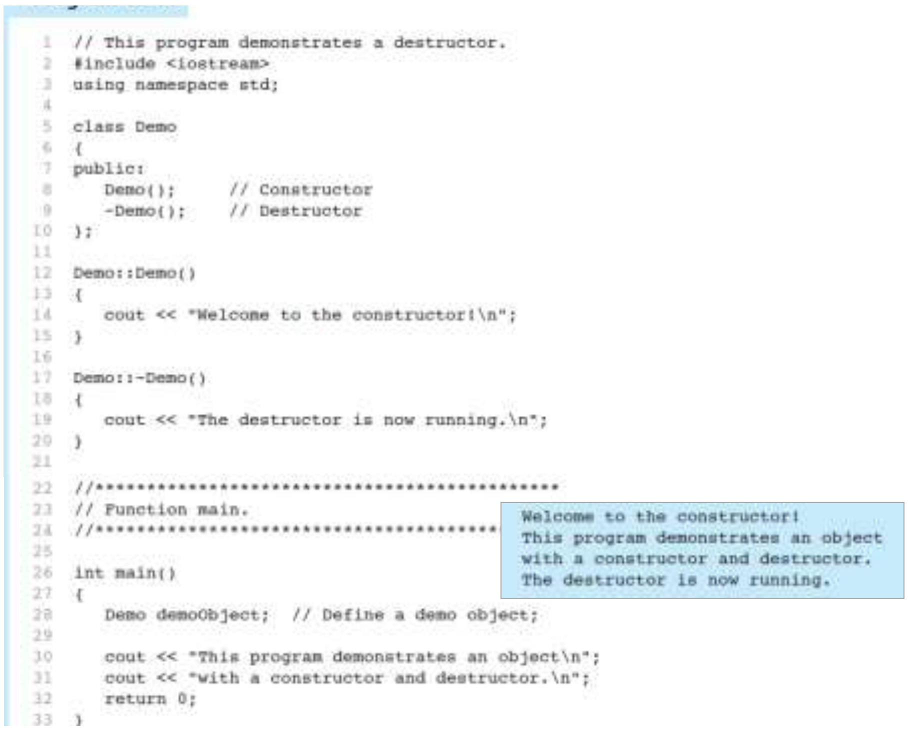
When all of a class’s constructors require arguments, then the class does not have a default constructor. In such a case you must pass the required arguments to the constructor when creating an object. Otherwise, a compiler error will result.

Destructors

CONCEPT: A destructor is a member function that is automatically called when an object is destroyed.

Destructors are member functions with the same name as the class, preceded by a tilde character (**~**). For example, the destructor for the Rectangle class would be named ~Rectangle.

Destructors are automatically called when an object is destroyed. They perform shutdown procedures when the object goes out of existence. For example, a common use of destructors is to free memory that was dynamically allocated by the class object.



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

class Demo
{
public:
   Demo();      // Constructor
   ~Demo();     // Destructor
};

Demo::Demo()
{
   cout << "Welcome to the constructor!\n";
}

Demo::~Demo()
{
   cout << "The destructor is now running.\n";
}

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

int main()
{
   Demo demoObject;  // Define a demo object;

   cout << "This program demonstrates an object\n";
   cout << "with a constructor and destructor.\n";
   return 0;
}
```

```
Welcome to the constructor!
This program demonstrates an object
with a constructor and destructor.
The destructor is now running.
```

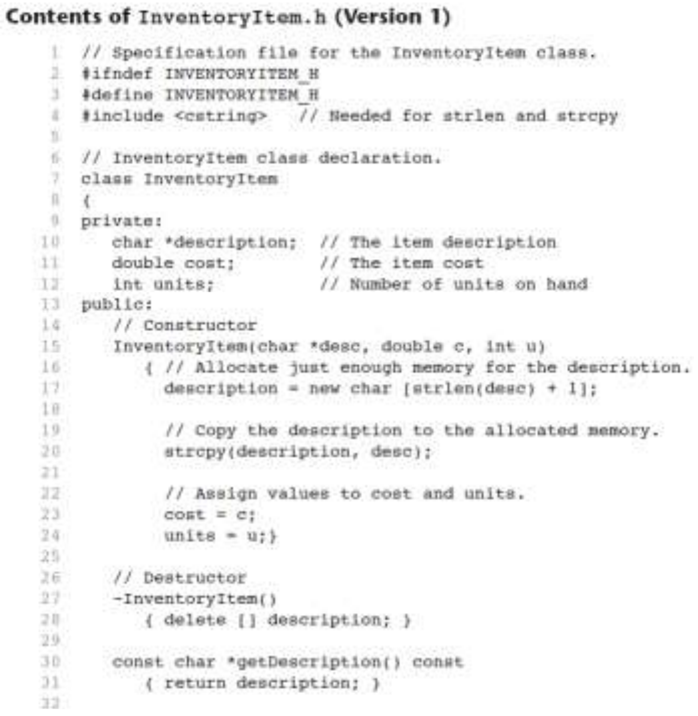
In addition to the fact that destructors are automatically called when an object is destroyed, the following points should be mentioned:

- Like constructors, destructors have no return type.
- Destructors cannot accept arguments, so they never have a parameter list.

The following code shows a more practical example of a class with a destructor. The InventoryItem class holds the following data about an item that is stored in inventory:

- The item's description
- The item's cost
- The number of units in inventory

The constructor accepts arguments for all three items. The description is passed as a pointer to a C-string. Rather than storing the description in a char array with a fixed size, the constructor gets the length of the C-string and dynamically allocates just enough memory to hold it. The destructor frees the allocated memory when the object is destroyed.



Contents of InventoryItem.h (Version 1)

```cpp
// 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
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;

   // 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
```

Destructors and Dynamically Allocated Class Objects

If a class object has been dynamically allocated by the new operator, its memory should be released when the object is no longer needed. For example, in the following code objectPtr is a pointer to a dynamically allocated InventoryItem class object.

```
// Define an InventoryItem pointer.
InventoryItem *objectPtr;
// Dynamically create an InventoryItem object.
objectPtr = new InventoryItem("Wrench", 8.75, 20);
```

Destructors and Dynamically Allocated Class Objects

The following statement shows the delete operator being used to destroy the dynamically created object.

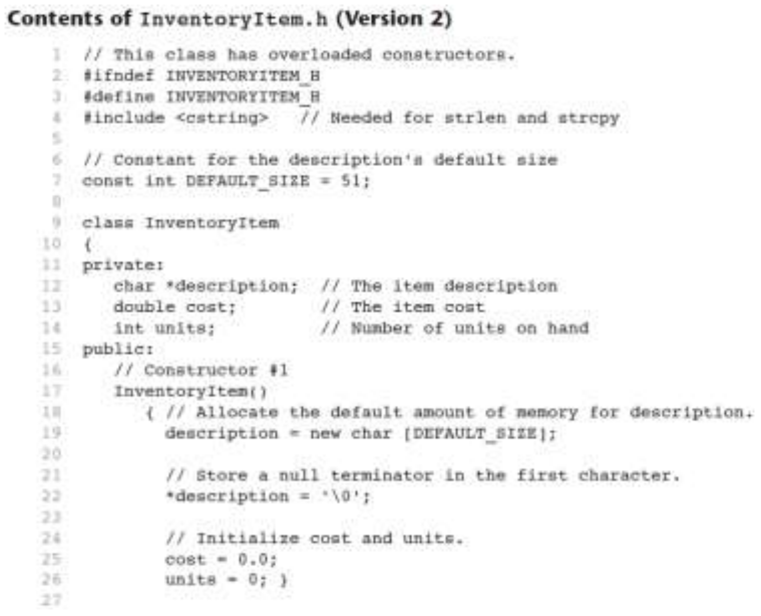
```
delete objectPtr;
```

When the object pointed to by objectPtr is destroyed, its destructor is automatically called.

Overloading Constructors

CONCEPT: A class can have more than one constructor.

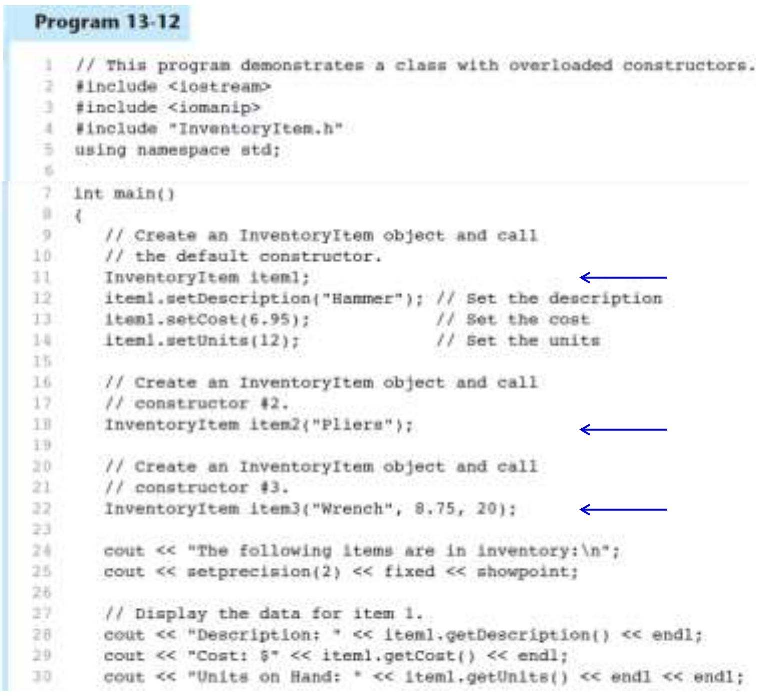
A new version of the InventoryItem class appears in the following code listing with three constructors. To simplify the code listing, all the member functions are written inline.



Contents of `InventoryItem.h` (Version 2)

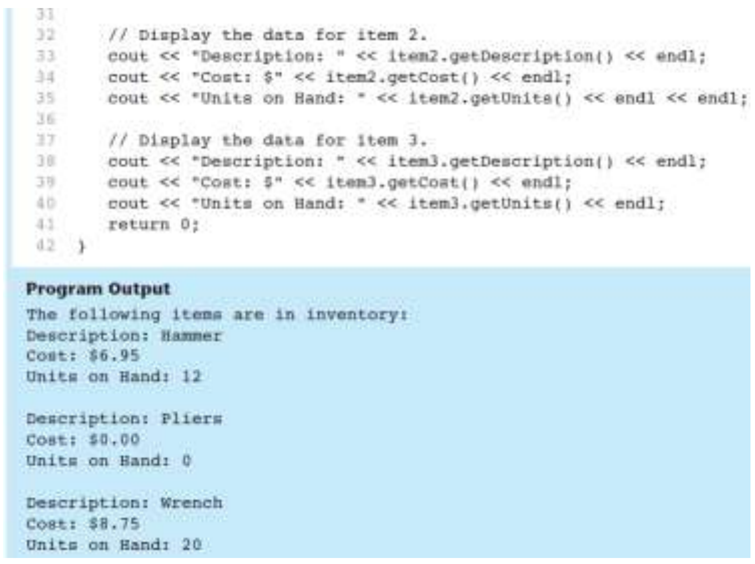
```
// This class has overloaded constructors.
#ifndef INVENTORYITEM_H
#define INVENTORYITEM_H
#include <cstring>   // Needed for strlen and strcpy

// Constant for the description's default size
const int DEFAULT_SIZE = 51;

class InventoryItem
{
private:
   char *description;  // The item description
   double cost;        // The item cost
   int units;          // Number of units on hand
public:
   // Constructor #1
   InventoryItem()
      { // Allocate the default amount of memory for description.
        description = new char [DEFAULT_SIZE];

        // Store a null terminator in the first character.
        *description = '\0';

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

   // Constructor #2
   InventoryItem(char *desc)
      { // Allocate just enough memory for the description.
        description = new char [strlen(desc) + 1];

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

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

   // Constructor #3
   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; }

   // Mutator functions
   void setDescription(char *d)
      { strcpy(description, d); }

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

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

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

   double getCost() const
      { return cost; }

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



Program 13-12

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

int main()
{
   // Create an InventoryItem object and call
   // the default constructor.
   InventoryItem item1;
   item1.setDescription("Hammer"); // Set the description
   item1.setCost(6.95);            // Set the cost
   item1.setUnits(12);             // Set the units

   // Create an InventoryItem object and call
   // constructor #2.
   InventoryItem item2("Pliers");

   // Create an InventoryItem object and call
   // constructor #3.
   InventoryItem item3("Wrench", 8.75, 20);

   cout << "The following items are in inventory:\n";
   cout << setprecision(2) << fixed << showpoint;

   // Display the data for item 1.
   cout << "Description: " << item1.getDescription() << endl;
   cout << "Cost: $" << item1.getCost() << endl;
   cout << "Units on Hand: " << item1.getUnits() << endl << endl;
```



```

   // Display the data for item 2.
   cout << "Description: " << item2.getDescription() << endl;
   cout << "Cost: $" << item2.getCost() << endl;
   cout << "Units on Hand: " << item2.getUnits() << endl << endl;

   // Display the data for item 3.
   cout << "Description: " << item3.getDescription() << endl;
   cout << "Cost: $" << item3.getCost() << endl;
   cout << "Units on Hand: " << item3.getUnits() << endl;
   return 0;
}
```

Program Output

```
The following items are in inventory:
Description: Hammer
Cost: $6.95
Units on Hand: 12

Description: Pliers
Cost: $0.00
Units on Hand: 0

Description: Wrench
Cost: $8.75
Units on Hand: 20
```

Only One Default Constructor and One Destructor

When an object is defined without an argument list for its constructor, the compiler automatically calls the default constructor. For this reason, a class may have only one default constructor. If there were more than one constructor that could be called without an argument, the compiler would not know which one to call by default.

Classes may also only have one destructor. Because destructors take no arguments, the compiler has no way to distinguish different destructors.

Other Overloaded Member Functions

Member functions other than constructors can also be overloaded. This can be useful because sometimes you need several different ways to perform the same operation. For example, in the InventoryItem class we could have overloaded the setCost function as shown here:

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

    void setCost(char *c)
    { cost = atof(c); }
```

←

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

void setCost(char *c)
{ cost = atof(c); }
```

The first version of the function accepts a double argument and assigns it to cost. The second version of the function accepts a char pointer. This could be used where you have the cost of the item stored in a string. The function calls the atof function to convert the string to a double, and assigns its value to cost.

Private Member Functions

CONCEPT: A private member function may only be called from a function that is a member of the same class.

It is used for internal processing by the class, not for use outside of the class

I wonder if there is a reason why we can't initialize members at their declaration.

```
class Foo
{
    int Bar = 42; // this is invalid
};
```

The initialization of non-static members could not be done like this prior to C++11. If you compile with a C++11 compiler, it should happily accept the code you have given.

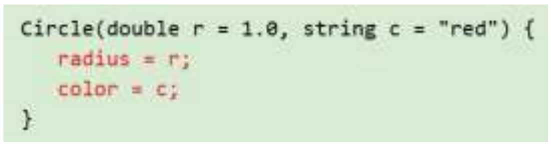
The main reason is that initialization applies to an object, or an instance, and in the declaration in the class there is no object or instance; you don't have that until you start constructing.

Because each object has its own data member. a class is only a template, there is no space for it. you can initialize data member in constructor because before initialize operator new has been called, space is ready for it.

http://stackoverflow.com

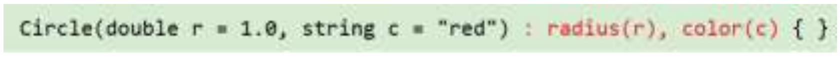
Constructor's Member Initializer List

Instead of initializing the private data members inside the body of the constructor, as follows:



```
Circle(double r = 1.0, string c = "red") {
   radius = r;
   color = c;
}
```

We can use an alternate syntax called *member initializer list* as follows:



```
Circle(double r = 1.0, string c = "red") : radius(r), color(c) { }
```

Arrays of Objects

CONCEPT: You may define and work with arrays of class objects.

As with any other data type in C++, you can define arrays of class objects. An array of InventoryItem objects could be created to represent a business's inventory records. Here is an example of such a definition:

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

This statement defines an array of 40 InventoryItem objects. The name of the array is inventory, and the default constructor is called for each object in the array.

If you wish to define an array of objects and call a constructor that requires arguments, you must specify the arguments for each object individually in an initializer list. Here is an example:

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

If a 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) };
```

This statement calls the third constructor in the InventoryItem class declaration for each object in the inventory array.

It isn’t necessary to call the same constructor for each object in an array. For example, look at the following statement.

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

This statement calls the second constructor for inventory[0] and inventory[2], and calls the third constructor for inventory[1].

If you do not provide an initializer for all of the objects in an array, the default constructor will be called for each object that does not have an initializer. For example, the following statement defines an array of three objects, but only provides initializers for the first two. The default constructor is called for the third object.

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

In summary, if you use an initializer list for class object arrays, there are three things to remember:

- If there is no default constructor you must furnish an initializer for each object in the array.
- If there are fewer initializers in the list than objects in the array, the default constructor will be called for all the remaining objects.
- If a constructor requires more than one argument, the initializer takes the form of a constructor function call.

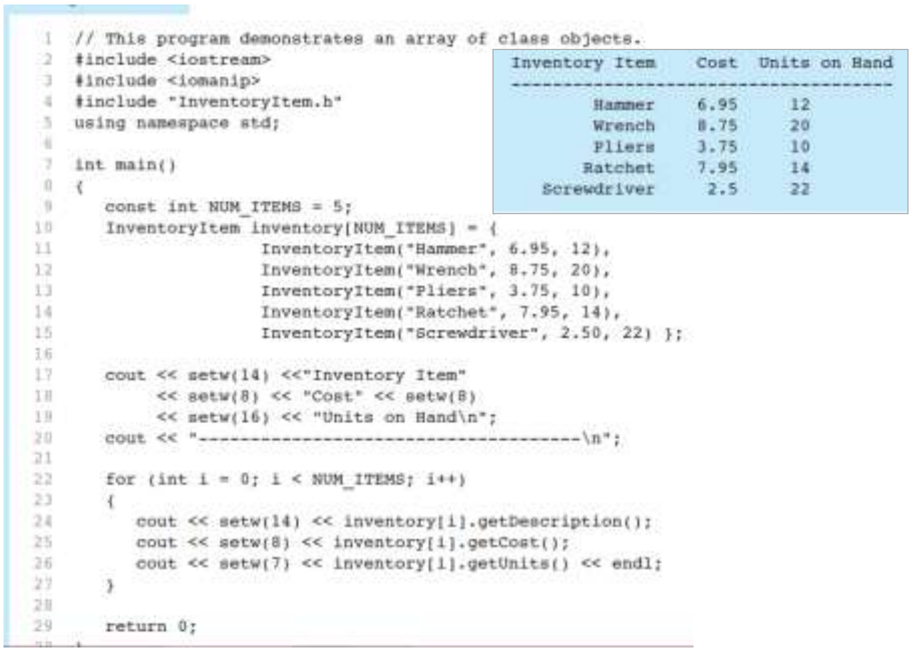
Accessing Members of Objects in an Array

Objects in an array are accessed with subscripts, just like any other data type in an array.

For example, to call the setUnits member function of inventory[2], the following statement could be used:

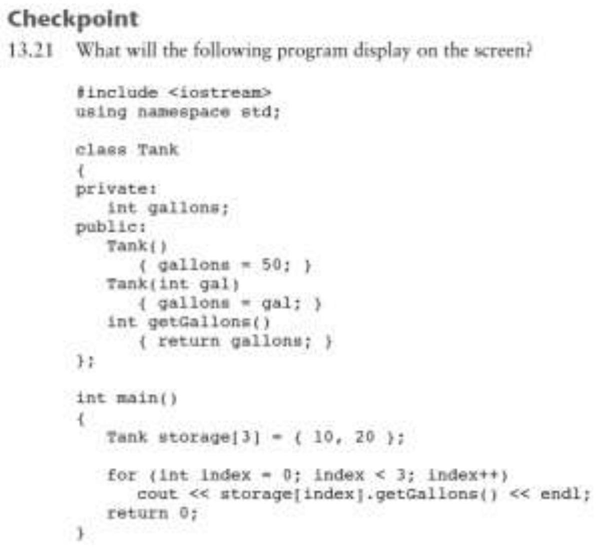
```
    inventory[2].setUnits(30);
```

This statement sets the units variable of inventory[2] to the value 30.



```
// 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";

   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;
```

```
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
```



Checkpoint

13.21 What will the following program display on the screen?

```
#include <iostream>
using namespace std;

class Tank
{
private:
   int gallons;
public:
   Tank()
      { gallons = 50; }
   Tank(int gal)
      { gallons = gal; }
   int getGallons()
      { return gallons; }
};

int main()
{
   Tank storage[3] = { 10, 20 };

   for (int index = 0; index < 3; index++)
      cout << storage[index].getGallons() << endl;
   return 0;
}
```

13.22 What will the following program display on the screen?

```
#include <iostream>
using namespace std;

class Package
{
private:
   int value;
public:
   Package()
      { value = 7; cout << value << endl; }
   Package(int v)
      { value = v; cout << value << endl; }
   ~Package()
      { cout << value << endl; }
};

int main()
{
   Package obj1(4);
   Package obj2();
   Package obj3(2);
   return 0;
}
```

13.26 Complete the following program so it defines an array of `Yard` objects. The program should use a loop to ask the user for the length and width of each `Yard`.

```
#include <iostream>
using namespace std;

class Yard
{
private:
   int length, width;
public:
   Yard()
      { length = 0; width = 0; }
   setLength(int len)
      { length = len; }
   setWidth(int w)
      { width = w; }
};

int main()
{
   // Finish this program
}
```

Vector of Rectangle objects using non default constructor

```
#include <vector>
Int main()
{
  vector<Rectangle> rectangles;
  for (int i=0; i<5; i++)
      rectangles.push_back(Rectangle(i,i));

  for (int i=0; i<5; i++)
      cout << rectangles.at(i).getArea()<<endl;
  return 0;
}
```

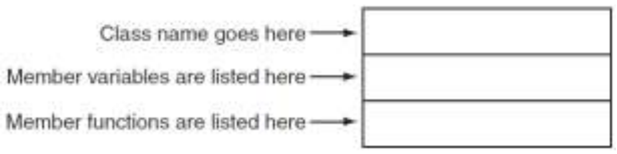
The Unified Modeling Language (UML)

CONCEPT: The Unified Modeling Language provides a standard method for graphically depicting an object-oriented system.

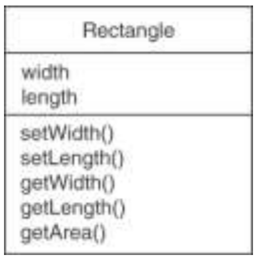
When designing a class it is often helpful to draw a UML diagram. *UML stands for Unified Modeling Language.*

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

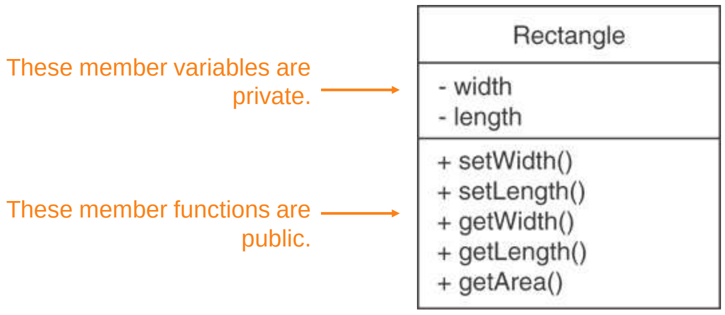
Example: A Rectangle Class



```
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
```

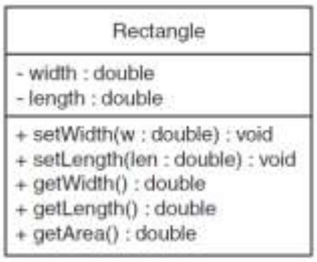
To indicate the data type of a function's parameter variable, place a colon followed by the name of the data type after the name of the variable.

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

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

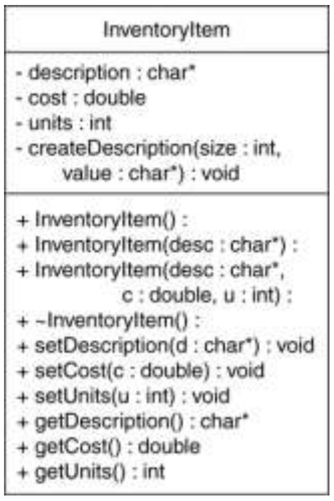


Showing Constructors and Destructors

No return type listed for constructors or destructors

Constructors →

Destructor →


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

Algorithm Workbench

43. Write a class declaration named Circle with a private member variable named radius. Write set and get functions to access the radius variable, and a function

named getArea that returns the area of the circle. The area is calculated as

3.14159 * radius * radius

44. Add a default constructor to the Circle class in question 43. The constructor should initialize the radius member to 0.

45. Add an overloaded constructor to the Circle class in question 44. The constructor should accept an argument and assign its value to the radius member variable.

46. Write a statement that defines an array of five objects of the Circle class in question

45. Let the default constructor execute for each element of the array.

47. Write a statement that defines an array of five objects of the Circle class in question

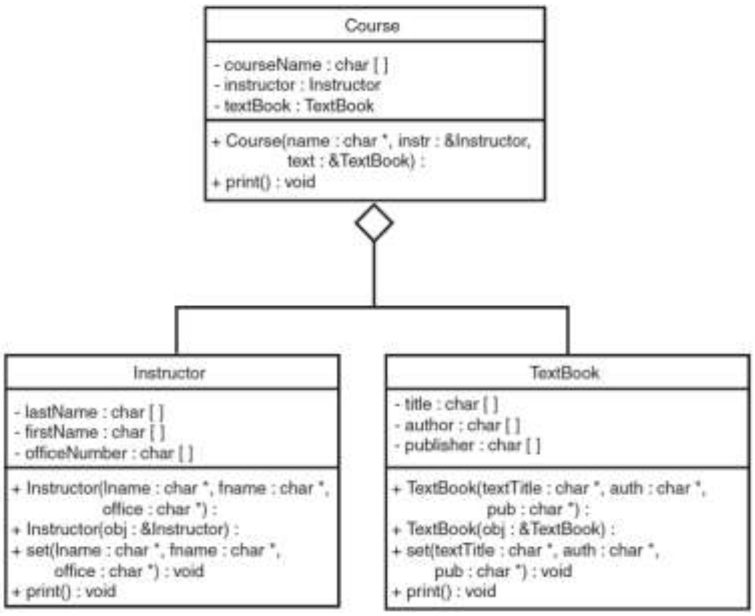
45. Pass the following arguments to the elements' constructor: 12, 7, 9, 14, and 8.

48. Write a for loop that displays the radius and area of the circles represented by the array you defined in question 47.

Aggregation

CONCEPT: Aggregation occurs when a class contains an instance of another class.

Aggregation in UML Diagrams


Course
- courseName : char [ ]
- instructor : Instructor
- textBook : TextBook
+ Course(name : char *, instr : &Instructor, text : &TextBook) :
+ print() : void
Instructor
- lastName : char [ ]
- firstName : char [ ]
- officeNumber : char [ ]
+ Instructor(lname : char *, fname : char *, office : char *) :
+ Instructor(obj : &Instructor) :
+ set(lname : char *, fname : char *, office : char *) : void
+ print() : void
TextBook
- title : char [ ]
- author : char [ ]
- publisher : char [ ]
+ TextBook(textTitle : char *, auth : char *, pub : char *) :
+ TextBook(obj : &TextBook) :
+ set(textTitle : char *, auth : char *, pub : char *) : void
+ print() : void

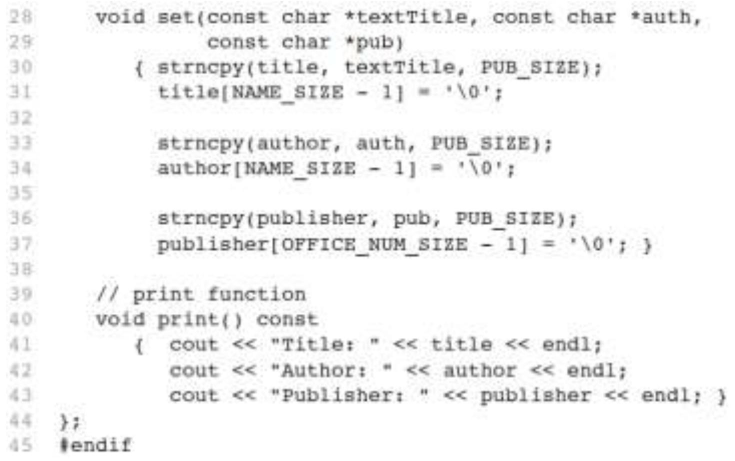
Contents of `Instructor.h`

```
#ifndef INSTRUCTOR
#define INSTRUCTOR
#include <iostream>
#include <cstring>
using namespace std;

// Constants for array sizes
const int NAME_SIZE = 51;
const int OFFICE_NUM_SIZE = 21;

// Instructor class
class Instructor
{
private:
   char lastName[NAME_SIZE];            // Last name
   char firstName[NAME_SIZE];           // First name
   char officeNumber[OFFICE_NUM_SIZE];  // Office number
public:
   // The default constructor stores empty strings
   // in the char arrays.
   Instructor()
      { set("", "", ""); }

   // Constructor
   Instructor(char *lname, char *fname, char *office)
```

```
      { set(lname, fname, office); }

   // set function
   void set(const char *lname, const char *fname,
            const char *office)
      { strncpy(lastName, lname, NAME_SIZE);
        lastName[NAME_SIZE - 1] = '\0';

        strncpy(firstName, fname, NAME_SIZE);
        firstName[NAME_SIZE - 1] = '\0';

        strncpy(officeNumber, office, OFFICE_NUM_SIZE);
        officeNumber[OFFICE_NUM_SIZE - 1] = '\0'; }

   // print function
   void print() const
      {  cout << "Last name: " << lastName << endl;
         cout << "First name: " << firstName << endl;
         cout << "Office number: " << officeNumber << endl; }
};
#endif
```

Contents of TextBook.h

```
#ifndef TEXTBOOK
#define TEXTBOOK
#include <iostream>
#include <cstring>
using namespace std;

// Constant for array sizes
const int PUB_SIZE = 51;

// TextBook class
class TextBook
{
private:
   char title[PUB_SIZE];     // Book title
   char author[PUB_SIZE];    // Author name
   char publisher[PUB_SIZE]; // Publisher name
public:
   // The default constructor stores empty strings
   // in the char arrays.
   TextBook()
      { set("", "", ""); }

   // Constructor
   TextBook(char *textTitle, char *auth, char *pub)
      { set(textTitle, auth, pub); }

   // set function
```



```
   void set(const char *textTitle, const char *auth,
            const char *pub)
      { strncpy(title, textTitle, PUB_SIZE);
        title[NAME_SIZE - 1] = '\0';

        strncpy(author, auth, PUB_SIZE);
        author[NAME_SIZE - 1] = '\0';

        strncpy(publisher, pub, PUB_SIZE);
        publisher[OFFICE_NUM_SIZE - 1] = '\0'; }

   // print function
   void print() const
      {  cout << "Title: " << title << endl;
         cout << "Author: " << author << endl;
         cout << "Publisher: " << publisher << endl; }
};
#endif
```

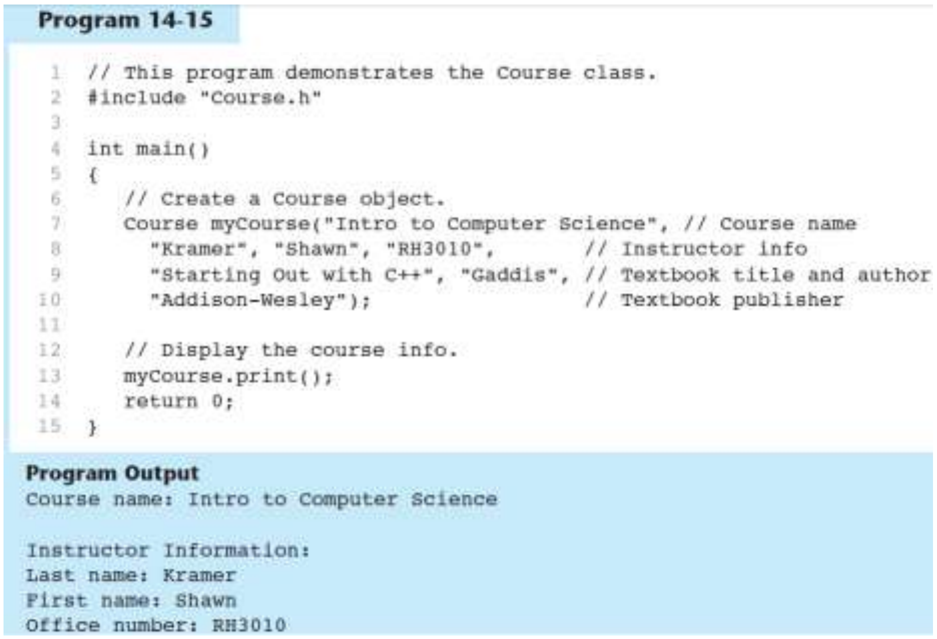
When an instance of one class is a member of another class, it is said that there is a "has a" relationship between the classes. For example, the relationships that exist among the Course, Instructor, and TextBook classes can be described as follows:

- The course *has an instructor.*
- The course *has a textbook.*

Contents of `Course.h`

```cpp
#ifndef COURSE
#define COURSE
#include <iostream>
#include <cstring>
#include "Instructor.h"
#include "TextBook.h"
using namespace std;

// Constant for course name
const int COURSE_SIZE = 51;

class Course
{
private:
   char courseName[COURSE_SIZE];  // Course name
   Instructor instructor;         // Instructor
   TextBook textbook;             // Textbook
```

```cpp
public:
   // Constructor
   Course(const char *course, const char *instrLastName,
          const char *instrFirstName, const char *instrOffice,
          const char *textTitle, const char *author,
          const char *publisher)
      { // Assign the course name.
        strncpy(courseName, course, COURSE_SIZE);
        courseName[COURSE_SIZE - 1] = '\0';

        // Assign the instructor.
        instructor.set(instrLastName, instrFirstName, instrOffice);

        // Assign the textbook.
        textbook.set(textTitle, author, publisher); }

   // print function
   void print() const
      {  cout << "Course name: " << courseName << endl << endl;
         cout << "Instructor Information:\n";
         instructor.print();
         cout << "\nTextbook Information:\n";
         textbook.print();
         cout << endl; }
};
```



Program 14-15

```
// This program demonstrates the Course class.
#include "Course.h"

int main()
{
   // Create a Course object.
   Course myCourse("Intro to Computer Science", // Course name
     "Kramer", "Shawn", "RH3010",        // Instructor info
     "Starting Out with C++", "Gaddis", // Textbook title and author
     "Addison-Wesley");                 // Textbook publisher

   // Display the course info.
   myCourse.print();
   return 0;
}
```

Program Output

```
Course name: Intro to Computer Science

Instructor Information:
Last name: Kramer
First name: Shawn
Office number: RH3010
```

Instance and Static Members

CONCEPT: Each instance of a class has its own copies of the class's instance variables. If a member variable is declared static, however, all instances of that class have access to that variable. If a member function is declared static, it may be called without any instances of the class being defined.

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

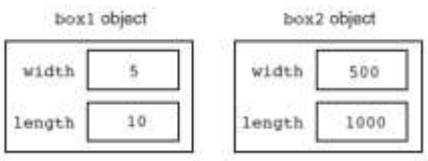
`static` member function: can be used to access `static` member variable; can be called before any objects are defined

Instance Variables

Each class object (an instance of a class) has its own copy of the class's member variables.

An object's member variables are separate and distinct from the member variables of other objects of the same class.

```
Rectangle box1, box2;

// Set the width and length for box1.
box1.setWidth(5);
box1.setLength(10);

// Set the width and length for box2.
box2.setWidth(500);
box2.setLength(1000);
```



Static Members

It is possible to create a member variable or member function that does not belong to any instance of a class. Such members are known as a *static member variables and static member functions.*

Static Member Variables

When a member variable is declared with the key word static, there will be only one copy of the member variable in memory, regardless of the number of instances of the class that might exist. A single copy of a class's static member variable is shared by all instances of the class.

For example, the following Tree class uses a static member variable to keep count of the number of instances of the class that are created.

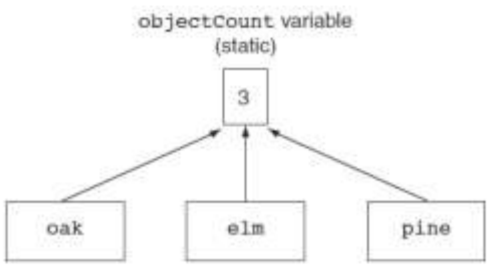
Contents of `Tree.h`

Static member declared here.

```
// Tree class
class Tree
{
private:
   static int objectCount;    // Static member variable.
public:
   // Constructor
   Tree()
      { objectCount++; }

   // Accessor function for objectCount
   int getObjectCount() const
      { return objectCount; }
};

// Definition of the static member variable, written
// outside the class.
int Tree::objectCount = 0;
```

Static member defined here.

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



Instances of the `Tree` class

Static Member Functions

You declare a static member function by placing the static keyword in the function’s prototype.

```
static ReturnType FunctionName (ParameterTypeList);
```

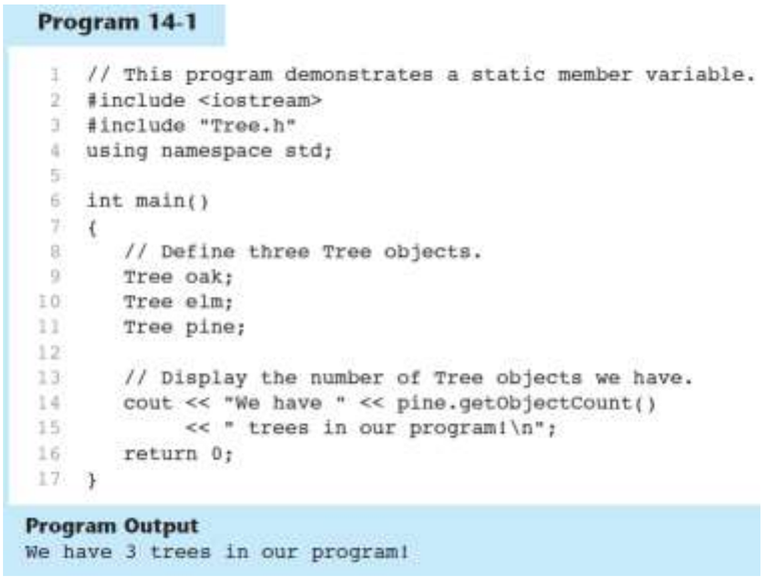
- Declared with `static` before return type:

```
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();
```



Program 14-1

```
// This program demonstrates a static member variable.
#include <iostream>
#include "Tree.h"
using namespace std;

int main()
{
   // Define three Tree objects.
   Tree oak;
   Tree elm;
   Tree pine;

   // Display the number of Tree objects we have.
   cout << "We have " << pine.getObjectCount()
        << " trees in our program!\n";
   return 0;
}
```

Program Output

```
We have 3 trees in our program!
```

Checkpoint

14.1 What is the difference between an instance member variable and a static member variable?

14.2 Static member variables are declared inside the class declaration. Where are static member variables defined?

14.3 Does a static member variable come into existence in memory before, at the same time as, or after any instances of its class?

14.4 What limitation does a static member function have?

14.5 What action is possible with a static member function that isn't possible with an instance member function?

HW

Lab 13

1. Students should read the Pre-lab Reading Assignment before coming to lab.
2. Students should complete the Pre-lab Writing Assignment before coming to lab. (photocopy or copy/paste, answer then print and bring to class.)