Software Engineering 2 (C++)

CSY2006

Procedural and Object-Oriented Programming

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

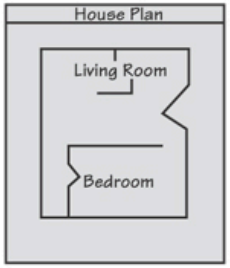
Object-Oriented Programming Terminology

- 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: members of a class

- methods or behaviors: member functions of a class

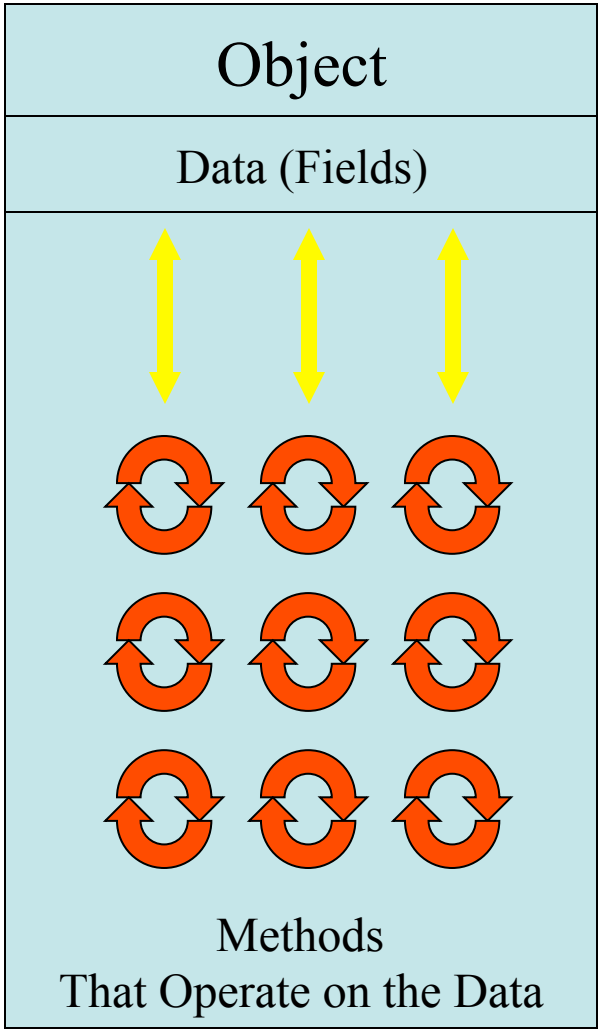
More on Objects

- encapsulation: combining (separation) of data and code into a single object
- data hiding: restricting access to certain members of an object
- data abstraction: implementation (low-level) details can be hidden
- Object reusability, Inheritance and Polymorphism

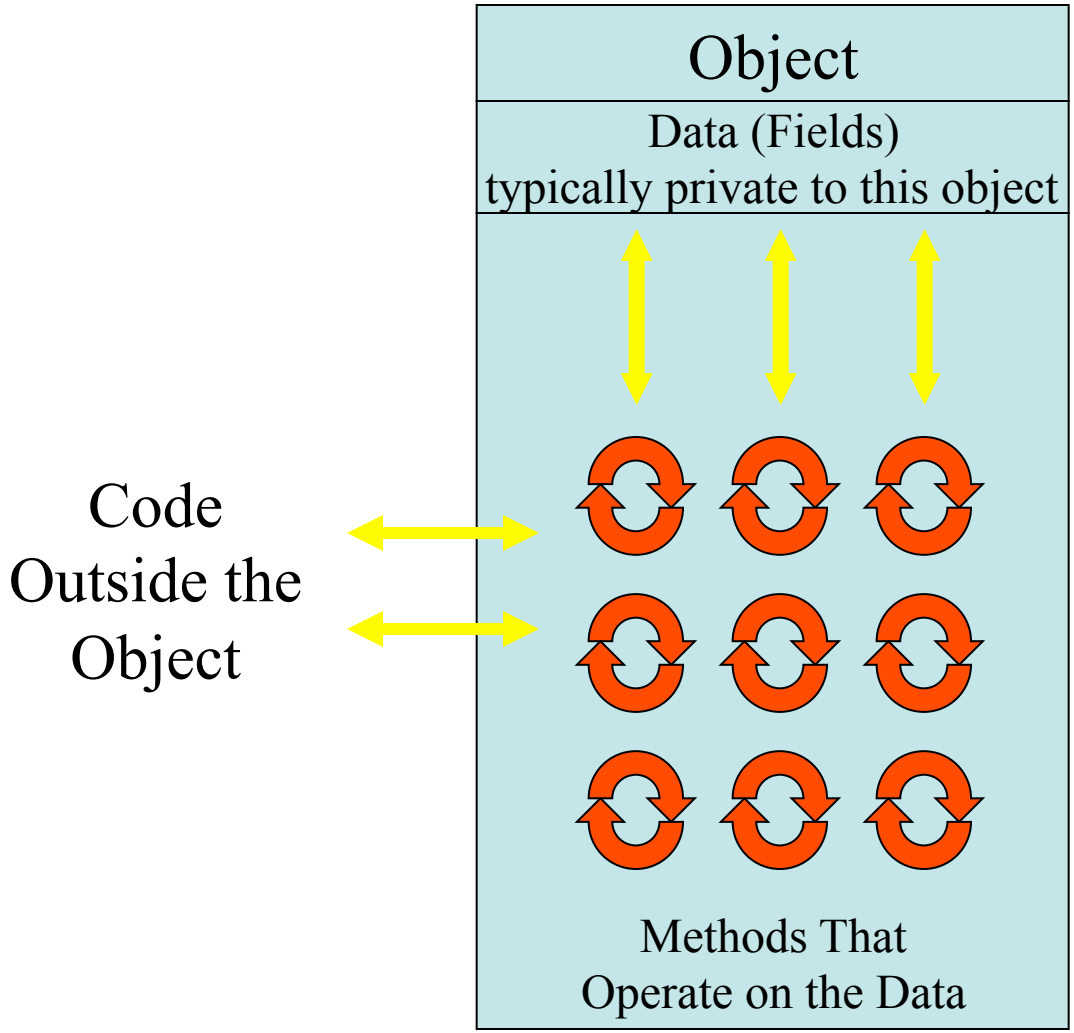
Object-Oriented Programming

- Object-oriented programming is centered on creating objects rather than procedures (methods/functions).
- Objects are a combination of data and procedures (methods/functions).
- Data in an object are known as *fields*.
- Procedures in an object are known as *methods*.

Object-Oriented Programming



Object-Oriented Programming



Object-Oriented Programming
Data Hiding

- Data hiding is important for several reasons.
  - It protects the data from accidental corruption by outside objects.
  - It hides the details of how an object works, so the programmer can concentrate on using it.
  - It allows the maintainer of the object to have the ability to modify the internal functioning of the object without “breaking” someone else's code.

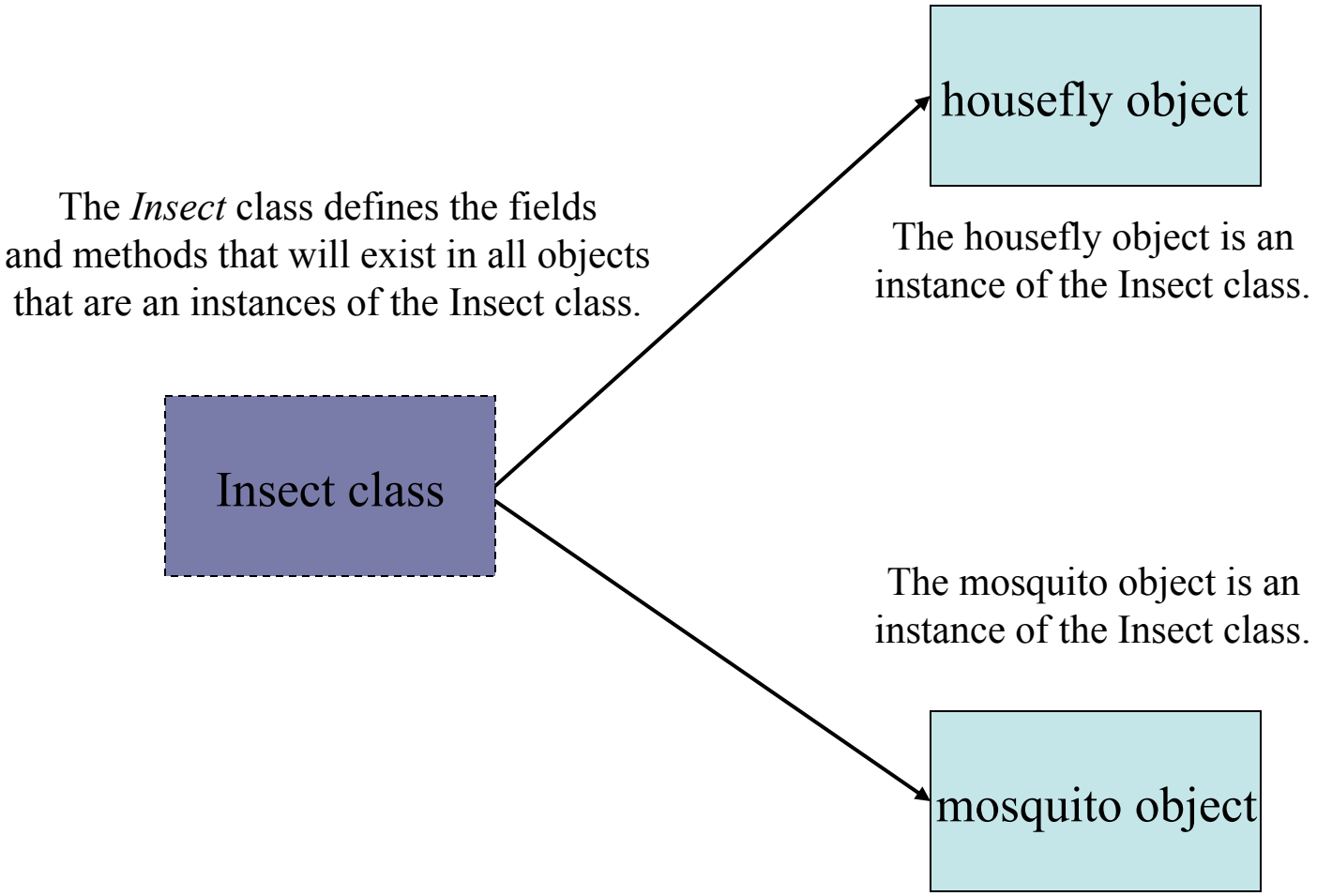
Object-Oriented Programming
Code Reusability

- Object-Oriented Programming (OOP) has encouraged object reusability.
- A software object contains data and methods that represents a specific concept or service.
- An object is not a stand-alone program.
- Objects can be used by programs that need the object's service.
- Reuse of code promotes the rapid development of larger software projects.

Classes and Objects

- The programmer determines the fields and methods needed, and then creates a class.
- A class can specify the fields and methods that a particular type of object may have.
- A class is a “blueprint” or “template” that objects may be created from.
- A class is not an object, but it can be a description of an object.
- An object created from a class is called an *instance* of the class.

Classes and Objects



Class and Method Definitions

A class as a blueprint (template)

Class Name: Automobile

Data:

amount of fuel_____

speed _____

license plate _____

Methods (actions):

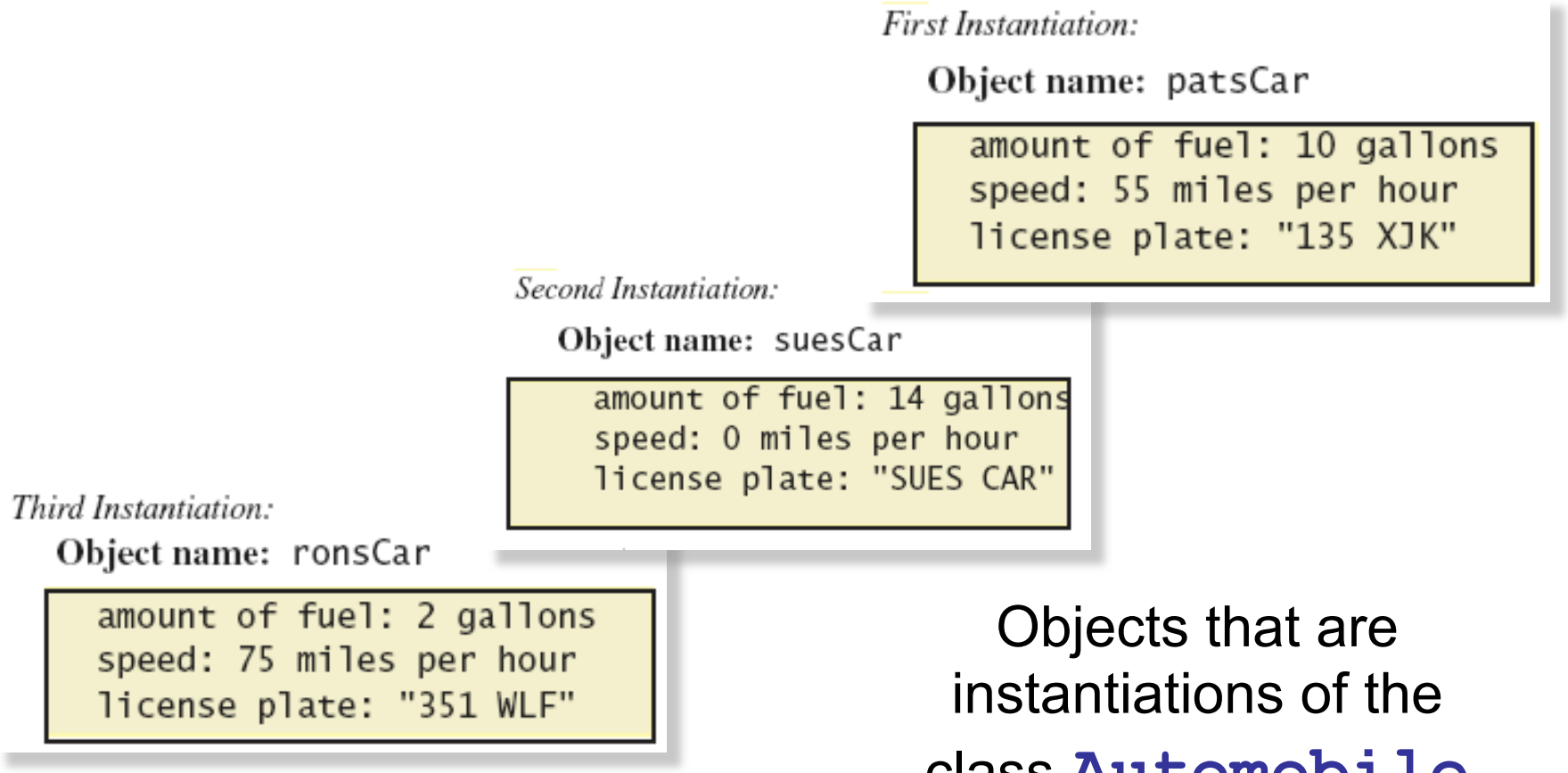
accelerate:

How: Press on gas pedal.

decelerate:

How: Press on brake pedal.

Class and Method Definitions



Objects that are instantiations of the class **`Automobile`**

Introduction to Classes

Introduction to Classes

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

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

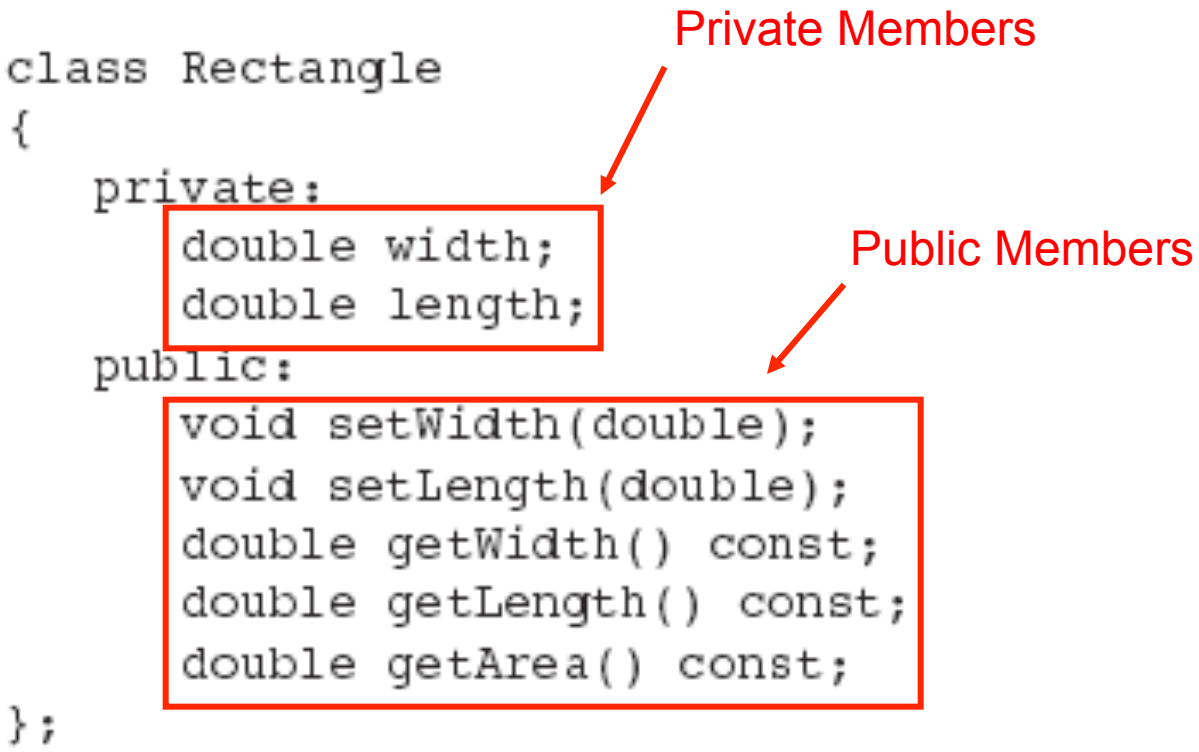
Class Example

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

Access Specifiers

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

Class Example



More on Access Specifiers

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

Using const With Member Functions

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

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

Defining a Member Function

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

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

Accessors and Mutators

- Mutator: a member function that stores a value in a private member variable, or changes its value in some way

- Accessor: function that retrieves a value from a private member variable. Accessors do not change an object's data, so they should be marked `const`.

Defining an Instance of a Class

Defining an Instance of a Class

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

```
Rectangle r;
```

- Access members using dot operator:

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

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

Program 13-1

```
1  // This program demonstrates a simple class.
2  #include <iostream>
3  using namespace std;
4
5  // Rectangle class declaration.
6  class Rectangle
7  {
8     private:
9        double width;
10       double length;
11    public:
12       void setWidth(double);
13       void setLength(double);
14       double getWidth() const;
15       double getLength() const;
16       double getArea() const;
17 };
18
19 //***************************************************
20 // setWidth assigns a value to the width member.    *
21 //***************************************************
22
23 void Rectangle::setWidth(double w)
24 {
25    width = w;
26 }
27
28 //***************************************************
29 // setLength assigns a value to the length member. *
30 //***************************************************
31
```

Program 13-1 (Continued)

```
32  void Rectangle::setLength(double len)
33  {
34     length = len;
35  }
36
37  //*************************************************
38  // getWidth returns the value in the width member. *
39  //*************************************************
40
41  double Rectangle::getWidth() const
42  {
43     return width;
44  }
45
46  //*************************************************
47  // getLength returns the value in the length member. *
48  //*************************************************
49
50  double Rectangle::getLength() const
51  {
52     return length;
53  }
54
```

Program 13-1 (Continued)

```
55  //*******************************************************
56  // getArea returns the product of width times length. *
57  //*******************************************************
58
59  double Rectangle::getArea() const
60  {
61     return width * length;
62  }
63
64  //*******************************************************
65  // Function main                                       *
66  //*******************************************************
67
68  int main()
69  {
70     Rectangle box;     // Define an instance of the Rectangle class
71     double rectWidth;  // Local variable for width
72     double rectLength; // Local variable for length
73
74     // Get the rectangle's width and length from the user.
75     cout << "This program will calculate the area of a\n";
76     cout << "rectangle. What is the width? ";
77     cin >> rectWidth;
78     cout << "What is the length? ";
79     cin >> rectLength;
80
81     // Store the width and length of the rectangle
82     // in the box object.
83     box.setWidth(rectWidth);
84     box.setLength(rectLength);
```

Program 13-1 (Continued)

```
85
86      // Display the rectangle's data.
87      cout << "Here is the rectangle's data:\n";
88      cout << "Width: " << box.getWidth() << endl;
89      cout << "Length: " << box.getLength() << endl;
90      cout << "Area: " << box.getArea() << endl;
91      return 0;
92  }
```

Program Output

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

Avoiding Stale Data

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

Pointer to an Object

- Can define a pointer to an object:

```
Rectangle *rPtr;
```

- Can access public members via pointer:

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

Dynamically Allocating an Object

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

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

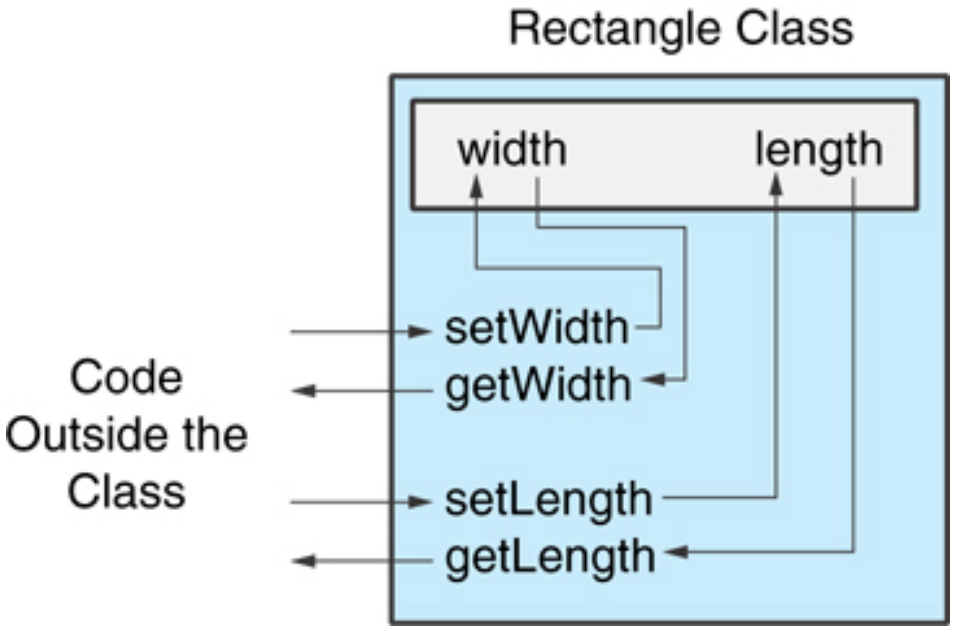
See: http://www.cplusplus.com/doc/tutorial/dynamic/

Why Have Private Members?

Why Have Private Members?

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

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



Separating Specification from Implementation

Separating Specification from Implementation

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

Separating Specification from Implementation

- #ifndef and #endif are preprocessor directives which are called *include guards*. It prevents header file from accidentally being included more than once.
- Notice use of quotes #include “Rectangle.h” for user-defined files in header folder (instead of #include <Rectangle>)
- Separating specification from implementation provides flexibility: When you give your code to another programmer, you can give specification file and compiled object file of class implementation. This protects your code (from changes/bugs).

Inline Member Functions

Inline Member Functions

- Member functions can be defined
  - inline: in class declaration
  - after the class declaration

- Inline appropriate for short function bodies:

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

Rectangle Class with Inline Member Functions

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

Tradeoffs – Inline vs. Regular Member Functions

- Regular functions – when called, compiler stores return address of call, allocates memory for local variables, etc.

- Code for an inline function is copied into program in place of call – larger executable program, but no function call overhead, hence faster execution

Constructors

Constructors

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

Contents of `Rectangle.h` (Version 3)

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

Contents of `Rectangle.cpp` (Version 3)

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

Continues...

Contents of `Rectangle.ccp` Version3 (continued)

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

Program 13-6

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

Program 13-6 *(continued)*

Program Output

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

Default Constructors

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

```
Rectangle r;
```

Passing Arguments to Constructors

Passing Arguments to Constructors

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

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

  - Use parameters in the definition:

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

Passing Arguments to Constructors

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

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

More About Default Constructors

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

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

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

```
Rectangle r;
```

Classes with No Default Constructor

- When all of a class's constructors require arguments, then the class has NO default constructor.

- When this is the case, you must pass the required arguments to the constructor when creating an object.

Destructors

Destructors

- Member function automatically called when an object is destroyed
- Destructor name is ~classname, *e.g.*, `~Rectangle`
- Has no return type; takes no arguments
- Only one destructor per class, *i.e.*, it cannot be overloaded
- If constructor allocates dynamic memory, destructor should release it

Contents of `InventoryItem.h` (Version 1)

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

Contents of `InventoryItem.h` Version1 (Continued)

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

Program 13-11

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

Program 13-11 *(continued)*

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

Program Output

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

Constructors, Destructors, and Dynamically Allocated Objects

- When an object is dynamically allocated with the new operator, its constructor executes:

```
Rectangle *r = new Rectangle(10, 20);
```

- When the object is destroyed, its destructor executes:

```
delete r;
```

Overloading Constructors

Overloading Constructors

- A class can have more than one constructor

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

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

```
1  // This class has overloaded constructors.
2  #ifndef INVENTORYITEM_H
3  #define INVENTORYITEM_H
4  #include <string>
5  using namespace std;
6
7  class InventoryItem
8  {
9  private:
10    string description; // The item description
11    double cost;        // The item cost
12    int units;          // Number of units on hand
13 public:
14    // Constructor #1
15    InventoryItem()
16       { // Initialize description, cost, and units.
17         description = "";
18         cost = 0.0;
19         units = 0; }
20
21    // Constructor #2
22    InventoryItem(string desc)
23       { // Assign the value to description.
24         description = desc;
25
26         // Initialize cost and units.
27         cost = 0.0;
28         units = 0; }
```

Continues...

```
29
30      // Constructor #3
31      InventoryItem(string desc, double c, int u)
32        { // Assign values to description, cost, and units.
33          description = desc;
34          cost = c;
35          units = u; }
36
37      // Mutator functions
38      void setDescription(string d)
39         { description = d; }
40
41      void setCost(double c)
42         { cost = c; }
43
44      void setUnits(int u)
45         { units = u; }
46
47      // Accessor functions
48      string getDescription() const
49         { return description; }
50
51      double getCost() const
52         { return cost; }
53
54      int getUnits() const
55         { return units; }
56  };
57  #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

Using Private Member Functions

Using Private Member Functions

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

Using Private Member Functions

```
 7  class ContactInfo{
 8  private:
 9      char *name;      // The contact's name
10      char *phone;     // The contact's phone number
11      // Private member function: initName
12      // This function initializes the name attribute.
13      void initName(char *n)
14      {     name = new char[strlen(n) + 1];
15            strcpy(name, n); }
16      // Private member function: initPhone
17      // This function initializes the phone attribute.
18      void initPhone(char *p)
19      {    phone = new char[strlen(p) + 1];
20           strcpy(phone, p); }
21  public:
22      // Constructor
23      ContactInfo(char *n, char *p)
24      { // Initialize the name attribute.
25        initName(n);
26        // Initialize the phone attribute.
27        initPhone(n); }
28
29      // Destructor
30      ~ContactInfo()
31      { delete [] name;
32        delete [] phone; }
```

Arrays of Objects

Arrays of Objects

- Objects can be the elements of an array:

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

- Default constructor for object is used when array is defined

Arrays of Objects

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

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

Arrays of Objects

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

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

Arrays of Objects

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

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

Accessing Objects in an Array

- Objects in an array are referenced using subscripts

- Member functions are referenced using dot notation:

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

Program 13-13

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

Program 13-3 (Continued)

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

Program Output

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

Instance and Static Members

- instance variable: a member variable in a class. Each object has its own copy.

- `static` variable: one variable shared among all objects of a class

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

`static` member variable

Contents of `Tree.h`

Static member declared here.

```
 1  // Tree class
 2  class Tree
 3  {
 4  private:
 5     static int objectCount;     // Static member variable.
 6  public:
 7     // Constructor
 8     Tree()
 9        { objectCount++; }
10
11     // Accessor function for objectCount
12     int getObjectCount() const
13        { return objectCount; }
14  };
15
16  // Definition of the static member variable, written
17  // outside the class.
18  int Tree::objectCount = 0;
```

Static member defined here.

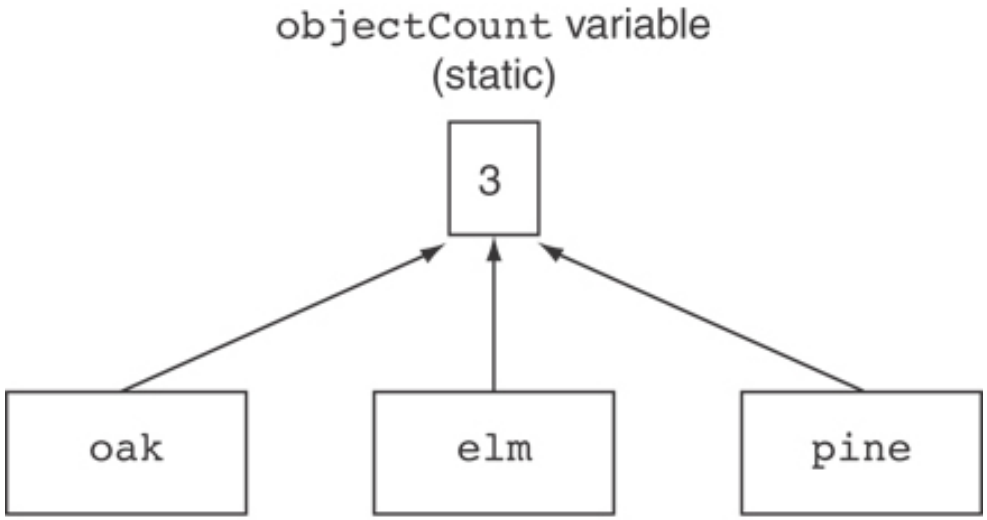
Program 14-1

```
1  // This program demonstrates a static member variable.
2  #include <iostream>
3  #include "Tree.h"
4  using namespace std;
5
6  int main()
7  {
8     // Define three Tree objects.
9     Tree oak;
10    Tree elm;
11    Tree pine;
12
13    // Display the number of Tree objects we have.
14    cout << "We have " << pine.getObjectCount()
15         << " trees in our program!\n";
16    return 0;
17 }
```

Program Output

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

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



Instances of the `Tree` class

`static` member function

- Declared with `static` before return type:

```
static int getObjectCount() const
{ return objectCount; }
```

- Static member functions can only access static member data
- Can be called independent of objects:

```
int num = Tree::getObjectCount();
```

Modified Version of `Tree.h`

```
 1  // Tree class
 2  class Tree
 3  {
 4  private:
 5     static int objectCount;    // Static member variable.
 6  public:
 7     // Constructor
 8     Tree()
 9        { objectCount++; }
10
11     // Accessor function for objectCount
12     static int getObjectCount() const
13        { return objectCount; }
14  };
15
16  // Definition of the static member variable, written
17  // outside the class.
18  int Tree::objectCount = 0;
```

Now we can call the function like this:

```
cout << "There are " << Tree::getObjectCount()
     << " objects.\n";
```