CSCE 121 Introduction to Program Design & Concepts

Building Objects from Classes

Dr. Tim McGuire

Grateful acknowledgment to Dr. Philip Ritchey and Dr. Michael Moore for some of the material on which these slides are based.

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 abstract data types 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`

Concept of Class

- Class is a generic description
  - Think blueprint
- Object is an **instance** of the class
  - Think the blue house on the corner
  - And the red one on the next block
  - And the green one next door

  - I.e. there are multiple instances of a single class

a class is a template for defining objects

Object-Oriented Programming Terminology

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

Visibility

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

Introduction to Classes

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

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

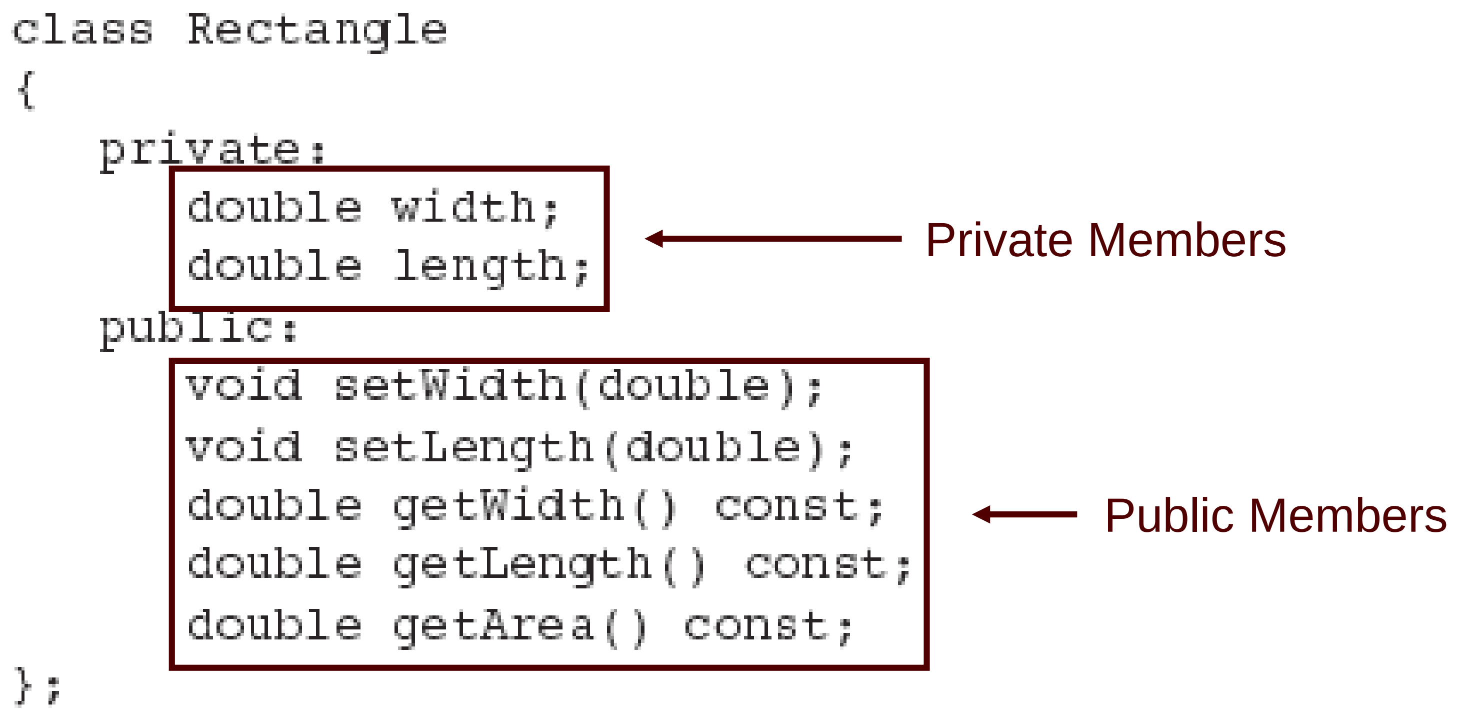
Class Example

```
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



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

Private Members
Public Members

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

- 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

- See **`rectangle0.cpp`** in Classes-Objects folder

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 = nullptr;
```

- 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.

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

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

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

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

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.

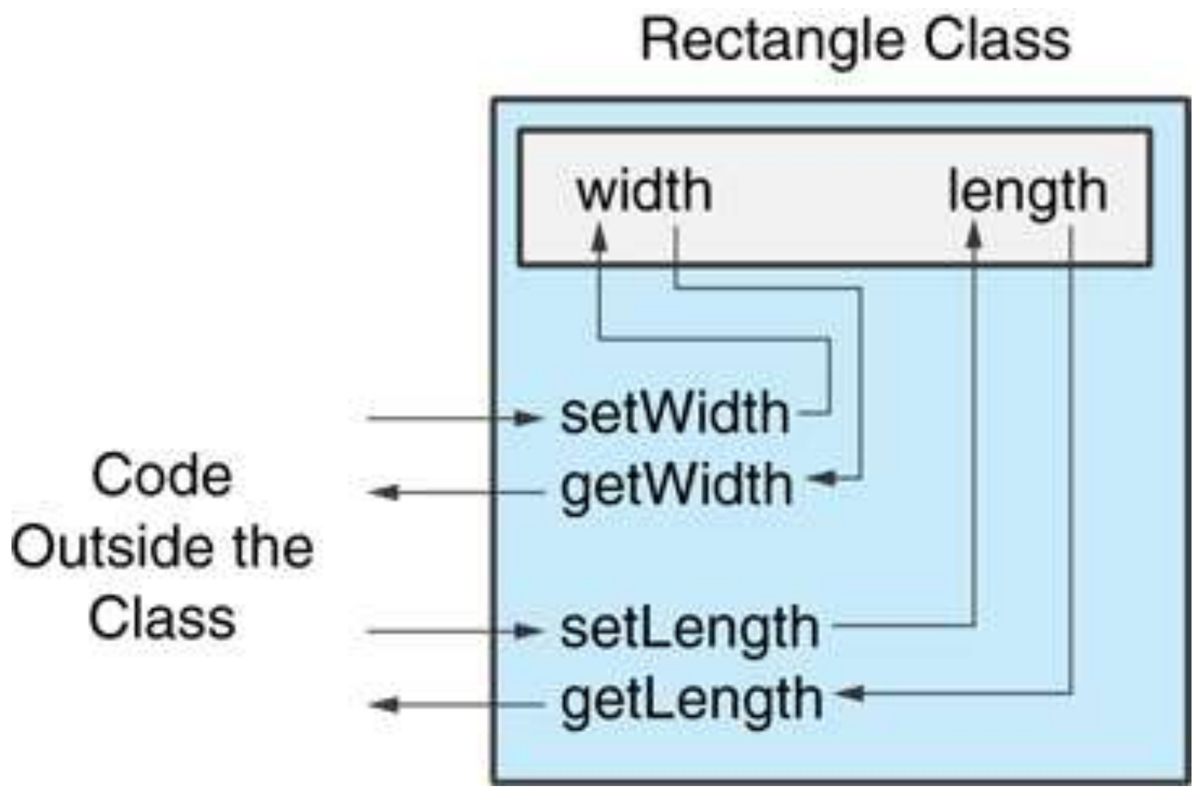


Image source: Tony Gaddis, *Starting out with Java*

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

- See Rectangle Version 1

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

- See Rectangle Version 2

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

Class 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

Constructors

- See Rectangle Version 3

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;
```

In-Place Initialization

- If you are using C++11 or later, you can initialize a member variable in its declaration statement, just as you can with a regular variable. However, sometimes initializations are more complicated, in which case a constructor is needed.
- This is known as in-place initialization. Here is an example:

```
class Rectangle
{
private:
   double width = 0.0;
   double length = 0.0;
public:
  Public member functions appear here…
};
```

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.

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);
```

Constructor Delegation

- Sometimes a class will have multiple constructors that perform a similar set of steps.
- In C++17, it is possible for one constructor to call another constructor in the same class.
- This is known as *constructor delegation*.
- For example, look at the following version of the `Rectangle` class:

```
class  Rectangle
{
…
     // Constructor #1 (default)
     Rectangle() : Rectangle(1.0, 1.0)
     { }


     // Constructor #1
    Rectangle(double w, double len)
    {
       width = w;
       length = len;

    }
```

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

Arrays of Objects

- Objects can be the elements of an array:

```
Rectangle box[40];
```

- The default constructor for the object is used when array is defined

Accessing Objects in an Array

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

```
box[2].setWidth(5.0);
box[2].setLength(10.0);
cout << box[2].getArea();
```

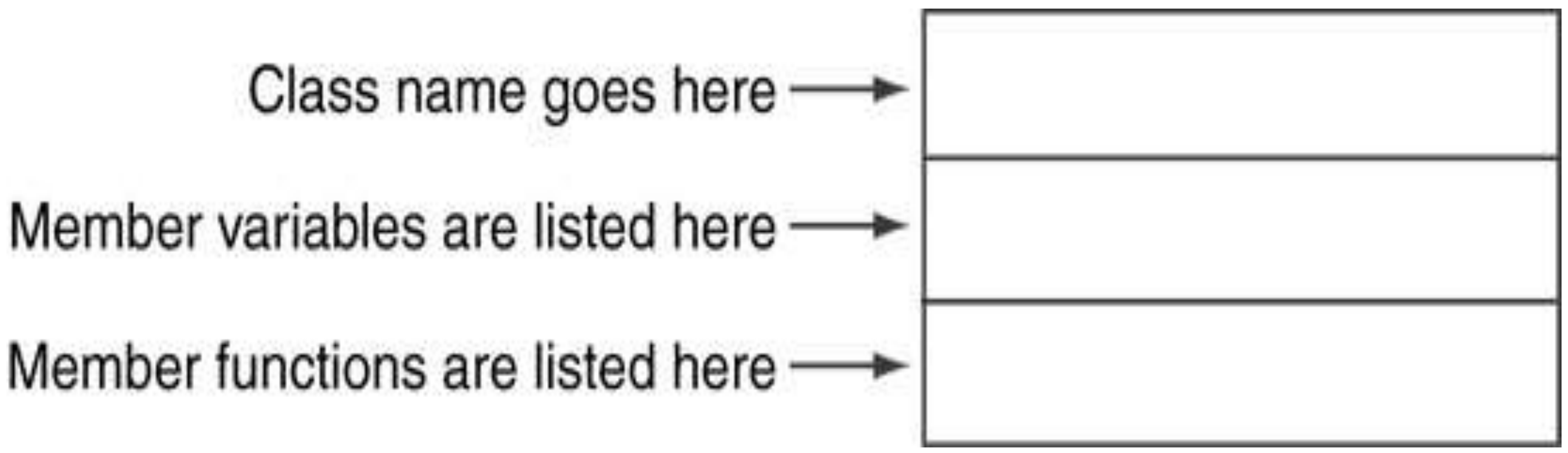
UML Diagrams

The Unified Modeling Language

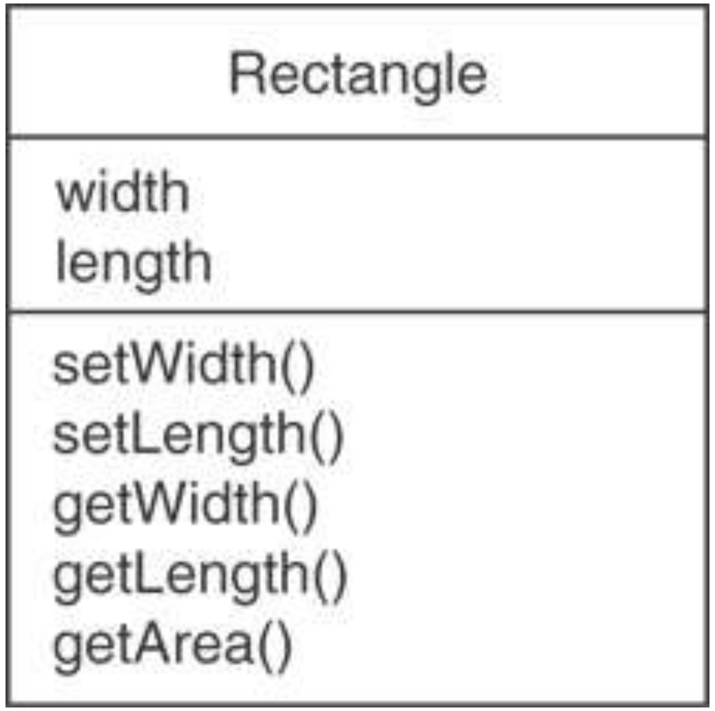
- *UML* stands for *Unified Modeling Language*.
- The UML provides a set of standard diagrams for graphically depicting object-oriented systems

UML Class Diagram

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


Class name goes here
Member variables are listed here
Member functions are listed here

Example: A Rectangle Class



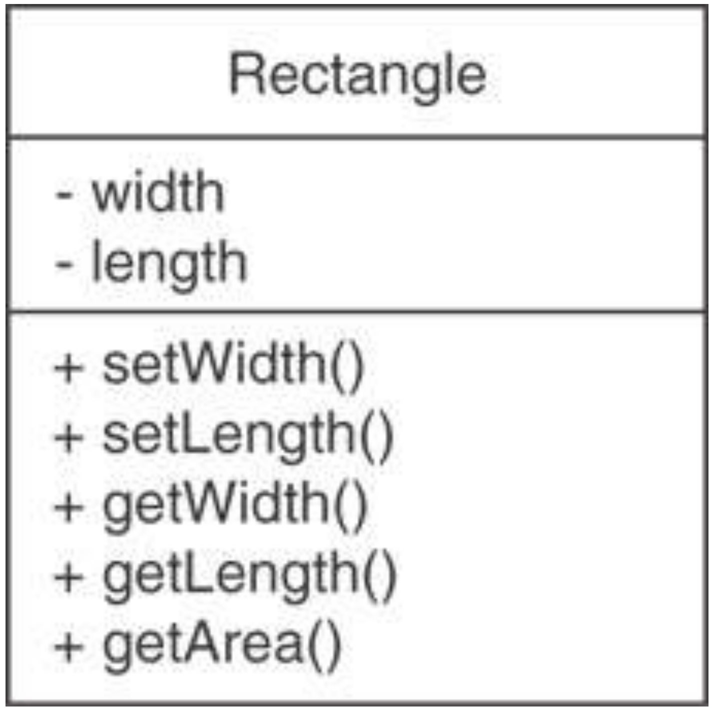
```
class Rectangle
{
   private:
      double width;
      double length;
   public:
      bool setWidth(double);
      bool setLength(double);
      double getWidth() const;
      double getLength() const;
      double getArea() const;
};
```

UML Access Specification Notation

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

These member variables are private.

These member functions are public.



UML Data Type Notation

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

```
- width : double
- length : double
```

UML Parameter Type Notation

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

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

UML Function Return Type Notation

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

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