

# CIT237

## Chapter 13: Introduction to Classes (Part 1)

October 21, 2019

**NOTICE to Students:** Portions of these lecture slides are

Copyright 2018 Pearson Education, Inc.

We have a license to use this material *only* in the context of this course.

There is NO PERMISSION to share these lecture slides with anyone not taking the course.

There is NO PERMISSION to post these lecture slides on the Internet.

# Reminders

- Quiz 4 will be held at the start of class on  
Wednesday, October 23.
- The material covered on Quiz 4 will be:
  - Lectures of October 7 and October 16.
  - Chapters 11 and 12.
- Programming Project 2:
  - The due date is EXTENDED to:  
Monday, October 28
  - If you are having difficulty, talk to me during the Lab portion of today's class.

# What's the Big Deal about Object Oriented Programming (OOP)?

- The sort of programming we have done up to now is called “Procedural Programming” (sometimes called “Modular Programming”).
  - We write our code, group it into modules (functions) so we can re-use portions of it, and it seems to work.
- So why do we need OOP?
  - For very small programs, written by only one person, very often we do not *really* need OOP.
  - But as systems become larger and more complex, with many programmers, traditional “procedural” programming methods become difficult to “get right”.

# Abstract Data Type (ADT)

- Created by the programmer
- Composed of one or more *primitive data types* (or other abstract data types).
- May include a defined set of operations  
(see also: Section 11.1 of the textbook)

# Abstract Data Type -- Example

- Let us define a data type, which we will call “Date”.
- A Date value contains three primitive data members:
  - Month: integer from 1 to 12
  - Day: integer from 1 to monthLength, where monthLength is one of {28, 29, 30, 31}
  - Year: integer
- Textual Representation (one of many):

03/25/2013

# Representing an ADT in C++

What if we use a “struct”?

```
struct Date
{
    int month;
    int day;
    int year;
};

Date thisDay;
thisDay.month = 3;
thisDay.day = 25;
thisDay.year = 2013;
```

# Operations on the “Date” ADT

Let's define several operations on “Date”:

- Increment:
  - Increment (03/25/2013): result is 03/26/2013
  - Increment (01/31/2012): result is 02/01/2012
- Decrement:
  - Decrement (01/30/2012): result is 01/29/2012
  - Decrement (01/01/2012): result is 12/31/2011
- Interval(d1, d2): number of days between two dates.
- NextWeek(d1): the Date value of d1 + seven days.
  - NextWeek(03/25/2013): result is 04/01/2013

# Implementing the Operations

1. We need a mechanism for writing code to implement the operations for the ADT.
2. We would like to be able to use meaningful names for the operations.
3. We would like to be able to re-use an operation implementation over and over, with different data values each time.

C++ functions are a perfect match for these requirements.



# So What Does this Get Us?

- Now we have a way to
  - represent complex data
  - implement specific operations on the data
- Does this help us write better programs?
  - It can help, but there are no absolute guarantees.

# Abstraction and Data Types

- Abstraction: a definition that captures general characteristics without necessarily specifying all of the details.
- Data Type defines the *values* that can be stored in a variable and the *operations* that can be performed on it.
- The user of an abstract data type (that is, the person who writes the calling program) does not need to know the details of how the data is stored, or how the operations are implemented.

# Procedural and Object-Oriented Programming

- Procedural programming focuses on the process/actions that occur in a program.
  - Often large amounts of data are defined in a central location, and there are many sections of the code which might modify any one portion of the data.
- Object-Oriented programming is based on the data and the functions that operate on it.
  - Objects can be thought of as instances of ADTs that represent the data and its functions.

# Limitations of Procedural Programming

- If the data structures change in a large system, many functions must also be changed.
- Programs that are based on complex function hierarchies can be:
  - difficult to understand and maintain,
  - difficult to modify and extend,
  - difficult to avoid creating new bugs during enhancement efforts.
- These problems are especially difficult if
  - a project has many programmers
  - the programming team is spread across a wide geographic area

# Object-Oriented Programming

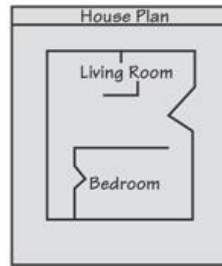
## Terminology

- class: like a `struct` (allows bundling of related variables), but also logically groups the functions (code) with the data.
- 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.



# Key Concept to Remember:

- An *object* is an **instance** of a *class*.
- Everyday examples:
  - Boston is an **instance** of a city.
  - BHCC is an **instance** of a college.
  - CIT-237 is an **instance** of a course.
  - Room D-116 is an **instance** of a classroom.

# More Terminology and Concepts

- Terminology:
  - Attributes: member variables of a class
  - Methods or Behaviors: member functions of a class
- Concepts:
  - Encapsulation: refers to the combining of data and code into a single object.
  - Data Hiding: refers to the ability to restrict 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.



# Declaring a Class

- In C++ we can use a class to implement an ADT.
- Objects are created from a `class`
- C++ Syntax:

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

# Example 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;
};
```

# Access Specifiers

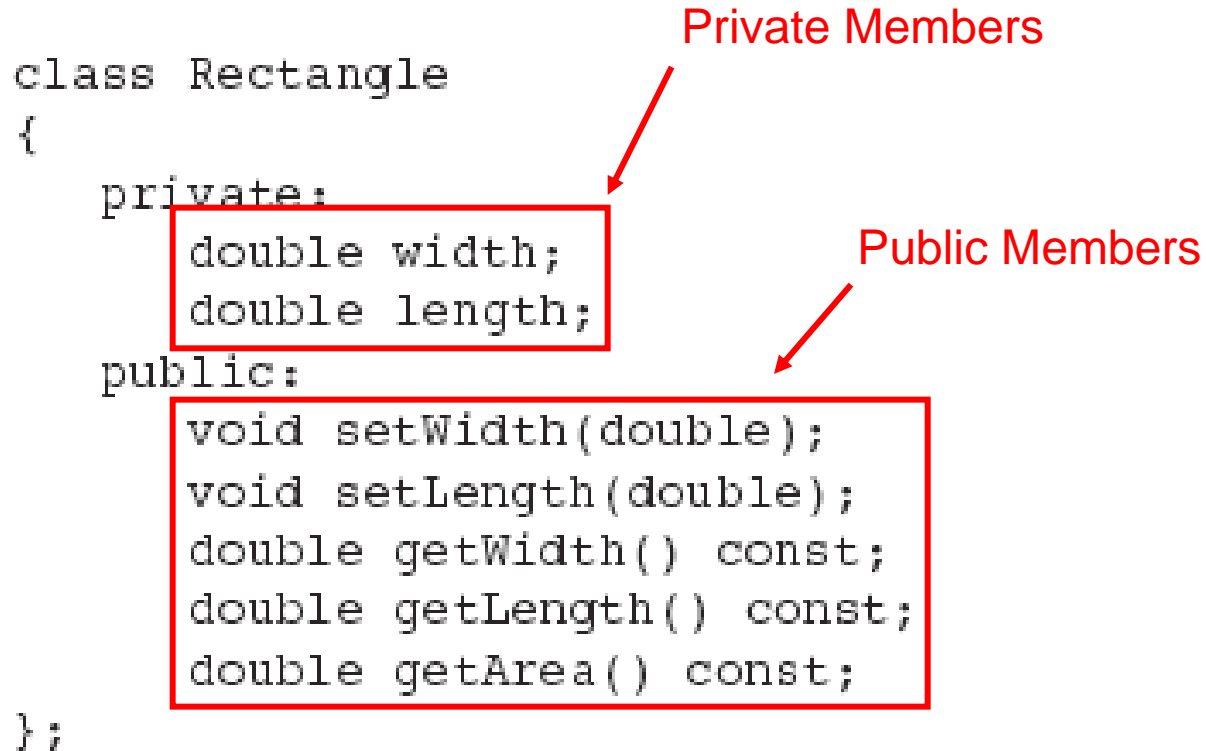
- Used to control access to members of the class.
  - **public**: can be accessed by functions outside of the class.
  - **private**: can only be called by or accessed by functions that are members of the class.
- Can be listed in any order in a class.
- Can appear multiple times in a class.
- If not specified, the default is **private**.

# 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



The diagram illustrates the structure of a C++ class. The code defines a class named 'Rectangle'. Inside the class, there are two private members: 'double width;' and 'double length;', which are enclosed in a red box. Above this box, the text 'Private Members' is written in red, with a red arrow pointing to the box. Below the private members, there are five public members: 'void setWidth(double);', 'void setLength(double);', 'double getWidth() const;', 'double getLength() const;', and 'double getArea() const;'. These are enclosed in another red box. To the right of this box, the text 'Public Members' is written in red, with a red arrow pointing to the box.

## 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;
```

# Declaring / Defining a Member Function

- Function **declaration**: Put the function prototype inside the class declaration (between the curly braces that follow the **class** keyword).
- Function **definition**: the actual code for the function is *usually* outside the class declaration:
  - place the class name and scope resolution operator (::) immediately before the function name:

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

# Accessors and Mutators

## (“getters and setters”)

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

# 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 (from code outside the class) using the dot operator.



# Program 13-1: Class Declaration

```
#include <iostream>
using namespace std;
class Rectangle
{
    private:
        double width;
        double length;
    public:
        void setWidth(double);
        void setLength(double);
        double getWidth() const;
        double getLength() const;
        double getArea() const;
};
```

# Program 13-1: Mutator Functions

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

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

# Program 13-1: Accessor Functions

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

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

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

# Program 13-1: main Function

```
int main()
{
    Rectangle box;    // Define Rectangle instance
    double rectWidth, rectLength; // local vars
    cout << "What is the width? ";
    cin >> rectWidth;
    cout << "What is the length? ";
    cin >> rectLength;
    box.setWidth(rectWidth);
    box.setLength(rectLength);
    cout << "Width: " << box.getWidth() << endl;
    cout << "Length: " << box.getLength() << endl;
    cout << "Area: " << box.getArea() << endl;
    return 0;
}
```

# Multiple Objects of the Same Class

- Program 13-2 (page 736) *re-uses* the Rectangle class
  - Creates three Rectangle objects:  
`kitchen, bedroom, den;`
  - Each of these objects its own private **length** and **width** variables.
  - After setting the **length**, **width** values for all 3 objects, the program calculates the total area:  
`totalArea = kitchen.getArea()  
          + bedroom.getArea()  
          + den.getArea();`

# A Member Function to Operate on Multiple objects

- Suppose we wish to write a function which is part of the class, called `totalArea`, which would calculate the area of TWO rectangles, and return the total area to the caller.
- The function prototype would be:  

```
double totalArea(Rectangle& );
```

# Example Function: `totalArea`

- The “current” object does not need a calling parameter.

```
double Rectangle::totalArea(Rectangle& rect)
{
    return getArea() + rect.getArea();
}
```

- We say that object **r1** invokes the function, passing **r2** as the argument:

```
Rectangle r1, r2;
r1.totalArea(r2);
```

# Avoiding Stale Data

- Some data is the result of a calculation.
- In the Rectangle class, the **getArea()** function re-calculates the area value whenever it is needed:

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

- If we were to redesign the Rectangle class to use an area variable, its value would be dependent on the length and the width values.
- Then, if we were to change length or width without updating area, then area would become *stale* (out of date).
- To avoid stale data, it is usually best to re-calculate the value of dependent data within a member function, rather than store it in a variable.

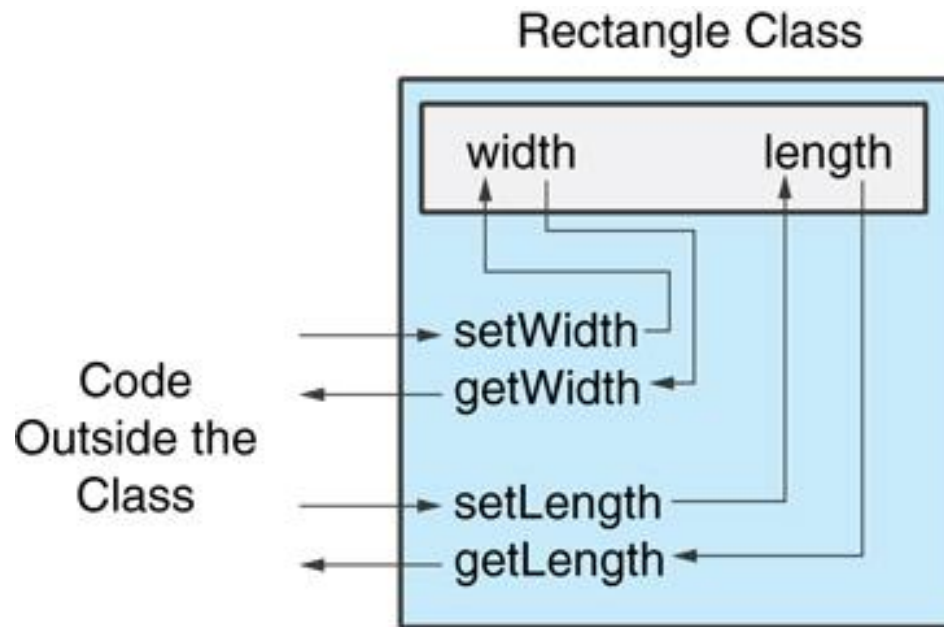


# Why Have Private Members?

- Making data members `private` provides data protection
- From outside the class, `private` member variables can be accessed only through `public` functions
- Public functions define the class's public interface

# Public Interface

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



# Abstract Data Type (formal definition)

A mathematical model which may be used to capture the essentials of a problem domain in order to translate it into a computer program; examples include: queues, lists, stacks, trees, graphs, and sets.

-- McGraw-Hill Dictionary of Electrical and Computer Engineering (copyright 2003).

(Do not let the fancy words intimidate you – ADT is a useful concept which can help you design and understand programs, but it does not *need* to be difficult to understand.)

# Summary

- Abstract Data Type (ADT)
  - created from other data types
  - includes operations on the data
- Class
  - Similar to Abstract Data Type.
  - Specifies data members and member functions.
  - Data members usually private.
  - Member functions usually public.
- An *object* is an instance of a *class*.