Programming in C/C++

Chapter 10: Introduction to Objects and Classes

(Chapter 12 in the old book)

Kristina Shroyer

Objectives

You should be able to describe:

- Class Data Types
  - Also called User Defined Data Types or Abstract Data Types
- Object-Based Programming
- Classes
- Constructors
- Destructors
- Examples
- Common Programming Errors

Class Data Types

- **General Concept:**
  - C++ allows you to create your own data types (class data types) by grouping a set of variables and the operations you want to act on that set of variables into a single structure which will be the definition of a new data type
  - So for you've written two types of programs
    - Programs that keep data in individual variables (built in types and class types)
    - Programs that use arrays:
      - Arrays make it possible to access list of data of ***same data type*** using single variable name

- Sometimes a relationship exists between two or more separate variables (of ***different*** data types) – in addition you might want to perform actions on these sets of variables
  - You can do this by creating your own class data types – some examples
    - Mailing list data (name, address, city, state, zip code)
      - Actions: changeName, changeAddress, printData
    - Parts inventory data (name of part, number in inventory)
      - Operations: addNewPart, increaseNumberOfParts, decreaseNumberOfParts
  - A class data type is a user defined data type (or one in the C++ library) that contains more than one variable and the variables can be different data types
    - In addition the class data type defines what type of actions operations can be performed on instances of the data type
    - Remember a data type is a set of data and the operations that act on that set of data

Class Data Types/Abstract Data Types

- The first thing we are going to discuss is the concept of an ***abstract data type (class data type)***
  - We're going to see that an object oriented program utilizes instances of abstract data types – it's just various instances of abstract data types working together
- In Chapter 2 we discussed data types
  - ***Data Type:*** A set of values **AND** operations that can be applied to these values
- We said there were two categories of data types
  - Primitive Data Types
  - Class Data Types
- So far most of the data types we've been using are **primitive** data types
  - The one class data type we've been using are ***strings***
- ***Concept:*** An ***class/abstract data type*** is a data type:
  1. ***That specifies all of the values the data type can hold and the all of the operations that can be applied to those values***
     - It is usually user created (even ones in the C++ library like strings were originally user created) and the operations are usually functions
     - It usually has one or more "variables" that make up the "set of data" of the class data type – think of our address book and think about how the string contains many characters -
       - In other words the set of data is more complicated than with primiitve types
     - The operations are usually functions that act on the set of data
  2. ***That can be used by a program without the need for anyone using the ADT (Abstract Data Type) to know how the data type itself is implemented***
     - **We're going to see that objects are instances of abstract data types that work together in a program…so an object oriented program is one that consists of many objects (instances of class data types) communicating with each other**

Class Data Types/Abstract Data Types

- **We know that a data type is, but what does the word *abstract* mean in terms of an *Abstract/Class Data Type*?**

- **Abstraction**
  - A general model of something
  - Two Parts to the definition of abstraction for our purposes

1. **An abstraction defines something (an object) in a way that allows someone to understand the general characteristics of that object without understanding the details of specific instances of that object**
   - Example:
   - "Dog" – The term "Dog" defines a general type of animal.
     - The term "Dog" captures the essence of what a dog is without specifying the detailed characteristics of any kind of dog

2. **Abstraction allows the details and operations of an object to be kept separate from the idea of what it can do and how to operate it**
   - Examples:
   - "Automobile"
     - Most everyone understands what an automobile is and how to drive one
     - However, very few people understand exactly how and automobile works or what parts go into building one
       - You can drive an automobile without understanding the details of how it works (you know to press the accelerator and brake but don't know what the car does internally to actually speed up or stop)
   - DVD Players
   - Computers

Class Data Types/Abstract Data Types

- Abstraction in Software Development
  - Abstraction occurs in programming as well as in the everyday world
  - As programmers we can use ADT's (and primitive data types) without understanding the details of how they work…we only need to know what the object does and how to use it
- A programmer needs to be able to use certain objects and routines (functions) without having to be bothered with understanding the details of their implementation
  - You have been doing this since the beginning of this class
    - Using objects such as **`cin`**, **`cout`**
    - Using functions such as **`pow`** and **`sqrt`**
  - All you need to know to use these objects and functions is what they do and the interface for using them
- **Examples:**
  - You only need to know the **`sqrt`** function must be called with one numeric argument and that it returns the square root of that argument
  - You only need to know how to format and use **`cout`** properly by separating separate data types with the << operator and knowing what types of data it can display
  - We've also used the **`string`** class (The string class is a definition of a class data type – when you create a specific string based on that definition you have an instance/object of type string)
    - All we needed to know was:
      - what a string is (a sequence of characters)
      - what operations (functions) we can perform on the string (Example: the length() function, the compare() function)
    - Note that the string class is an abstract data type – it defines the set of data that make up a string and the operations that can be performed on a string data types
      - When you create a string object (variable of type string) you are creating and instance of a string data type

Class Data Types/Abstract Data Types

- Abstraction in Software Development

- ***Data Abstraction***
  - Abstraction applies to data types as well
  - To use any data type as a programmer you only need to know two things
    1. The values it can hold
    2. What operations can be applied on those values

- **Example 1:** To use a `double` you need to know:
  - The values it can hold: it can only hold numeric values (not strings such as "5.1")
    - It is used to hold fractional values, so 5 as a `double` is 5.0
  - What operations can be performed on it: It can be used with addition, subtraction, multiplication, and division but NOT with modulus (mod only works on integers)
  - You do NOT need to know anything else about double in order to use it. You don't need to know how it is stored in memory or how the arithmetic operations that can be performed are carried out by the computer

- **Example 2 :** The string class. (the `string` class is an ADT – object)
  - To use the `string` class all we needed to know was:
    - what a string is (a sequence of characters)
    - what operations (functions) we can perform on the string (Example: the length() function, the compare() function)

- This idea is known as data abstraction
  - ***Data Abstraction:*** the separation of a data type's logical properties from its implementation details

Class Data Types/Abstract Data Types

- So let's relate the concept of ADTs to programming (OOP = Object Oriented Programming)
  - In OOP we define ADTs to be used in programs (we use classes to do this).
    - These ADTs are ***reusable*** and can be used by many different programs just by creating an instance of the ADT (an object).
      - Think of the **`string`**
    - This is similar to how we create variables that are instances of primitive data types

- ***Abstract Data Types (Class Data Types)***
  - **Definition:** The term **abstract data type (the definition of an object)** describes any data type for which the implementation details are kept separate from the logical properties needed to use it
    - In common usage the term is applied only to data types created by the programmer (this is why you may hear it referred to as a **User Defined Data Type**)
    - Abstract data type usually group together more than one type of variable to create a set of data representing the type
  - A programmer usually creates an abstract data type by:
    - Defining a set of values the data type can hold
    - Defining a set of operations that can be performed on that data
      - Creating a set of functions to carry out these operations
  - In C++ and other object oriented languages, programmer created Abstract Data Types (User Defined Data Types) are normally implemented as **classes**.

- In object oriented programs, objects (instances) of ADT's interact with each other

Object Oriented/Procedural Programming

- Before we continue with the technical discussion of objects let's contrast object oriented programming and procedural programming and see why object oriented programming is the technique used (why it is better).

- ***There are two common programming methods in practice today***
  - ***Procedural Programming***
    - focuses on the process/actions that occur in a program that act on ***data that is SEPARATE from the processes or actions (not defined in the same structure)***
    - In procedural programming data values are treated as passive quantities to be acted upon by functions
    - Procedural Programming does not correspond very well with the idea of an abstract data type
      - An ADT consists of BOTH data values and the operations on those values in one unit communicating directly with each other and belonging together
      - It is preferable to view an ADT as defining an ACTIVE data structure
        - One that combines both data an operations in to a single cohesive unit (procedural keeps data and actions separate)

  - ***Object Oriented Programming***
    - Supports the idea of ADTs by using a structured type known as a class
      - We'll see that a class is a definition of an abstract data type
      - Just as we create instances of primitive data types (ints, doubles etc) called variables we also create instances of classes (definitions of ADTs)…these instances are called objects
    - Is based on the idea of an ADT (an ADT's data and the functions that operate on that data).
    - A class is used to define an ADT
      - Objects are *instances* of class Abstract Data Types, or User Defined Data Types
        - So the data type itself is the model/definition and when we create a specific one of the model then we have an instance in our program (a variable is an instance of a primitive type)

Procedural Programming

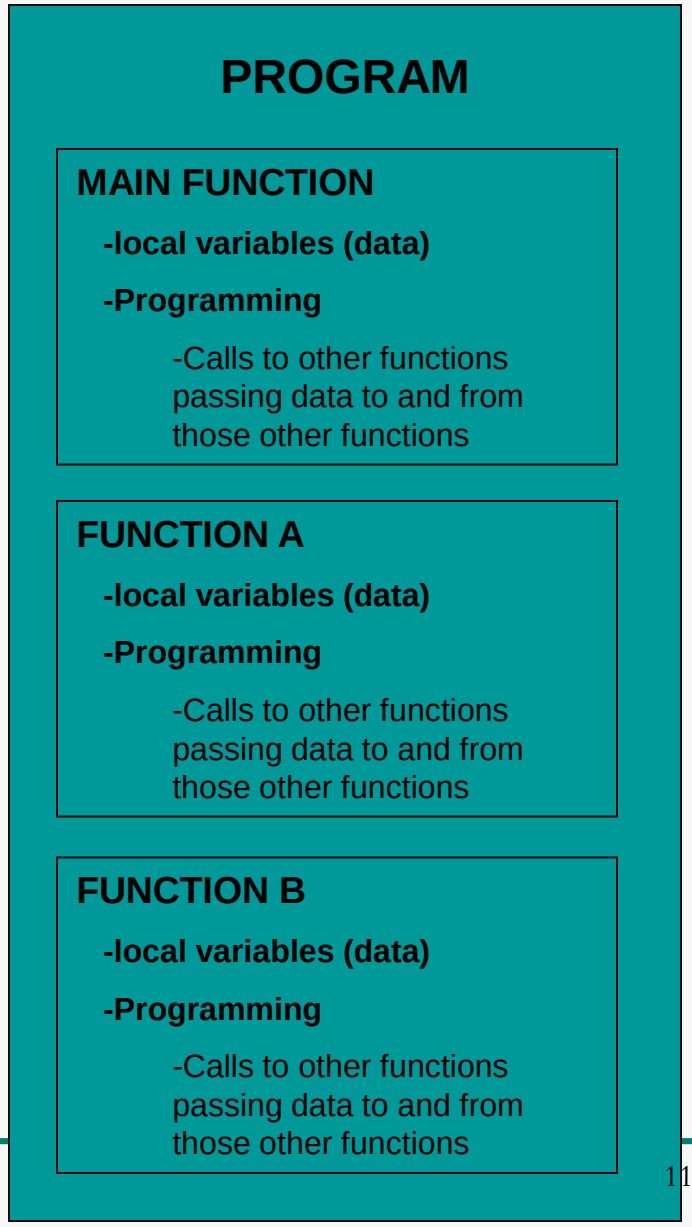
- What is Procedural Programming?
  - In procedural programming the programmer constructs procedures (functions in C++) that operate on data that is *separate* from those procedures (functions)

```
int main
{
     //data
     int x;
     int y = 1;
     double z;

     //calls to procedures (functions in your programs)
     x = function1(y);
     z = function2(x);
     //remember that data can be passed between functions

}

//your function definitions, again remember your data is passed in and out of these
//methods

int function1(int num1)
{

}

double function2(int num2)
{
}
```

  - Procedural programming focuses on data and methods that operate on that data by passing the data between each other
- So far in this course you have been doing procedural programming and linear programming

Procedural Programming

- What is Procedural Programming?
  - Procedural Programming Diagram
  - This procedural program will grow to have more and more functions as it becomes more complex
  - While functions are easier to trace and understand than having the entire program in one main method (linear), as a program becomes more complex and contains increasingly more methods it can become harder to read and follow
    - This makes the program harder to:
      - Design
      - Modify
      - Debug



Procedural Programming

- Limitations of Procedural Programming
  - Complex Programs
    - A program may be broken into many different functions to allow easier understanding of the program
      - However, there is a limit to the number of functions a person is capable of understanding and remembering
      - It's common for real world programs to have hundreds of functions that interact with each other in many different ways
      - This complexity creates problems for a new programmer brought in on a project
  - Programs that are Difficult to Modify, Debug, or Extend
    - What happens when a program reaches a certain level of complexity?
      - It becomes difficult to modify (extend, change or debug) and understand how modifications may affect other parts of the program
        - Functions pass data amongst each other and modify it creating dependencies between two or more functions (what happens in one function affects another)
        - When a programmer alters one function the changes they make may unknowingly affect other methods in ways the programmer did not intend

Procedural Programming

- Limitations of Procedural Programming
  - Excessive global data
    - The public accessibility of global data can open the door for a programmer to write code that accidentally corrupts vital data
    - Errors in these types of programs be hard to find
    - It is difficult to modify a function in this type of program without having to also modify other functions that are affected by that function's data
  - These types of procedural programming problems can cost companies money
    - Programs that are hard to modify and debug take more time for the software developers/programmers which in turn costs money
    - Programs that are not readable make it hard for a new software developer to start later in the project or to come in later to maintain code
      - This happens often
    - Object Oriented programming has been shown to reduce these problems and related costs

Objects in the Real World – Big Picture

- First let's think about objects in the real world
  - Object Oriented Programming is based on a software model similar to how people view objects in the real world
- Everywhere in the real world you see objects
  - People, animals, plants, cars…etc.
  - Humans think in terms of objects – telephones, houses, traffic lights ovens – everything we think about is an object
  - Computer programs can also be viewed as objects composed of a lot of interacting software objects
- There are different kinds of objects (some are inanimate and some are animate – move around and do things)
- Objects of both kinds however have two things in common.
  - All objects have attributes (size, color, weight…..etc)
  - All objects have behaviors (a ball rolls, bounces, deflates etc.)
  - So each object of the same type has the same attribute and behavior definitions available to them but the specific degree of an attribute a specific object has or which behaviors specific objects perform are unique to the individual instances of each object
- Humans learn about existing objects by studying their attributes and learning their behaviors. Comparisons can be made between the attributes and behaviors of different objects. (for example babies vs. adults)
- Thinking about humans as an object in terms of software and programming think of it this way
  - People have their their own classifications of attributes and behaviors
    - The formal definition of what people are in terms of their attributes and behavior describe the abstract data type for a person (an abstract data type can be represented by the class structure in C++)
    - Each individual that is a person with these attributes and behaviors available to them is an instance of the people(person) class. Each individual has these behaviors available for them to use or perform at any time

Objects – Big Picture

- What is Object Oriented Software Design (OOD)?
  - OOD Models software in terms similar to those that people use to describe real world objects
    - Objects of a certain class (a certain ADT) such as a class of vehicles, have the same characteristics (attributes and behaviors) in common
  - OOD provides a natural intuitive way to view software design where program objects are modeled/classified by their attributes and behaviors just as real world objects are
  - OOD also models communication between software objects must like real world objects
    - Communication occurs between objects. One object may be able to communicate with another via the public methods of communication the other has available to it

Object Oriented Programming – Big Picture

- What is Object Oriented Programming (OOP)?
  - The implementation object oriented design – so we are going to create data types and have instances of those data types interact in our programs.
- Object oriented programming is centered around the concept of **classes** which are structures used to define **abstract data types (class data types)** which package together both data and the methods that operate on that data
  - **Abstract data type (class data type) :** A user defined data type (or one in a language library) which is defined as a collection of data and a set of operations on that data
  - **What is a data type?**
    - Think back to Chapter 2 where you learned about the primitive data types
      - You learned about how to create primitive data types for holding numeric data
      - The data type defined the type of data a primitive variable could hold
      - Example: an int has a size of 4 bytes and can hold integers in a specific range of values
    - A **Variable** is an instance of a primitive data type
      - Variables represent storage locations in a computer's memory – the data type you declare a variable to be determines the amount of memory the variable uses and the way the variable formats and stores data
      - What you use each instance of int variable for is different but each int variable holds the same type/set of data and can have the same functions applied to it
    - A program can also bundle together a group of data items into a **structure data type** (structs).
      - See optional lecture on structs if you are interested
  - `Classes` are used to define abstract data types

Object Oriented Programming – Big Picture

- ***Classes*** are ***definitions*** of Abstract Data Types (Class Data Types) which contain BOTH data members and a set of operations that operate on those data members. They also are designed so that they can be used without needing to know their implementation details
  - Just as you can create many variable instances of primitive data types you can also create many object instances of classes (the definitions/blueprints that define abstract data types)

- **Definitions:**
  - ***Class:*** A definition of structured data type in a programming language that is often used to represent (define) an abstract data type
    - Think of it as a blueprint or definition for an abstract data type, like a struct, it only DEFINES the ADT, you can't use it until you create a variable (object) of that type
  - ***Class member:*** A class member is a component of the class. This is similar to a member of a struct except class members are **BOTH data and the functions that operate on that data.**
  - ***Class object (class instance):*** Instances (specific tangible examples) of classes are called objects. These are instances of the class type created (similar to how a variable is an instance of a primitive data type).
  - ***Client:*** Any software that declares and manipulates objects of a class type is called client of the class

Important point: a DEFINITION of a class data type does not allocate any memory, it is only when a specific instance of a class data type is created
that memory is allocated

Object-Oriented Programming - Book

- **Objects**: Well suited to programming representation
  - Can be specified by two basic characteristics:
    - **State:** How object appears at the moment
      - The state of an object depends on its attributes (DATA MEMBERS)
        - For example a rectangle object would have shape attributes such as length and width as well as location attributes which may be defined as the points its corner positions are at
    - **Behavior:** How object reacts to external inputs (SET OF OPERATIONS THAT ACT ON DATA MEMBERS)
      - The behavior of an object depends on the operations that can be performed on an objects attributes
        - Example: A rectangle can have the area computed

- **Example of an object**: A rectangle
  - **State**: Shape and location
    - Shape specified by length and width
    - Location defined by corner positions
  - **Behavior**: Can be displayed on a screen, can change its size

Object Oriented Programming and Objects

- Simple Representation of a Circle Class
- The Circle class is a formal definition of an ADT
  - An abstract data type contains a set of data items and the operations on that data
    - the data and functions relate to the circle object the abstract data type is representing
- The data items that are a part of the Circle Class are called ***Member Variables*** (can also be called Member Attributes, are called Data Members in your book)
  - This example has one data item radius
- The functions (operations on the object's attributes) that operate on the data are called ***Member Functions***
  - This example has two functions: the **`setRadius()`** and **`calcArea()`** functions
- Notice I used the word member – the variables and functions of the Circle ADT are thought of as ***members*** of the Circle Class in Object Oriented Programming

Note that the member functions of the circle class can ONLY act on instances of Circles (similar to how the length() function of the string class can only act on instances of strings)

Circle
Member Variables: double radius

Member Functions:
void setRadius(double r)
{
radius = r;
}
 	
double calcArea()
{
return 3.141592 * pow(radius,2);
}

Object Oriented Programming and Objects

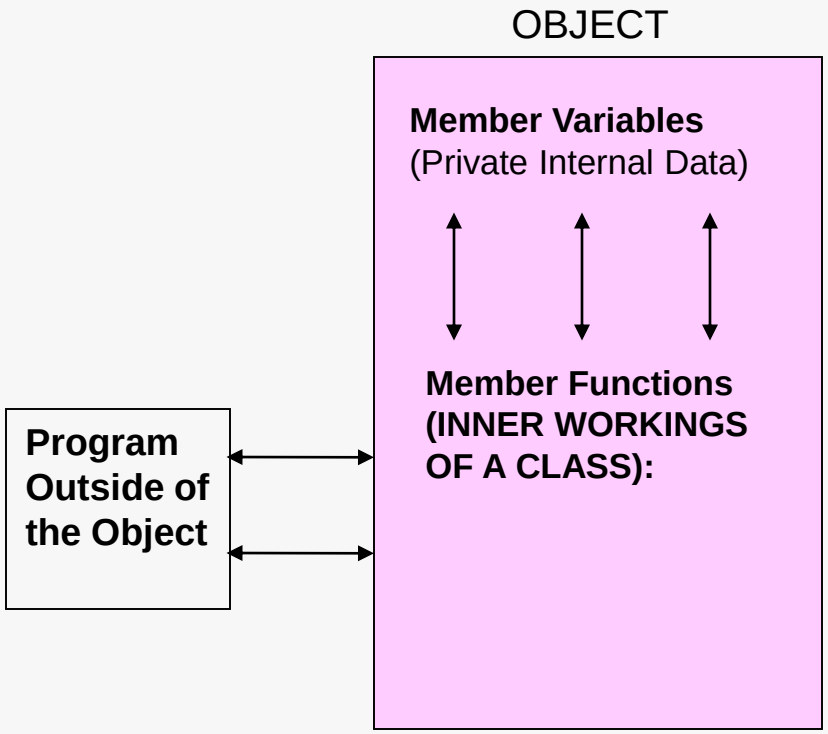
- Simple Representation of a Circle Class
- Notice that radius is not passed into the `calcArea()` function like it would be if we were writing a procedural program to calculate the radius and area of a circle
  - This is because the data and operations on the data of the circle object are all bound together in a single unit.
    - Member functions have direct access to member variables
- This idea of bounding all of an object's data and the functions that operate on that data into a single object is called ***Encapsulation***.
  - The member functions of the circle object have direct access to the member attributes (data)
  - ***However code outside of the class does not necessarily have access to an class's member data.***
  - ***Member Variables can be designed so only member functions of an object can access them***.
  - The ability of objects to restrict parts of the outside program from accessing certain member variables or functions is called ***Data Hiding***.

Circle
Member Variables: double radius

Member Functions:
void setRadius(double r)
{
radius = r;
}
 	
double calcArea()
{
return 3.14 * pow(radius,2);
}

Object Oriented Programming and Objects

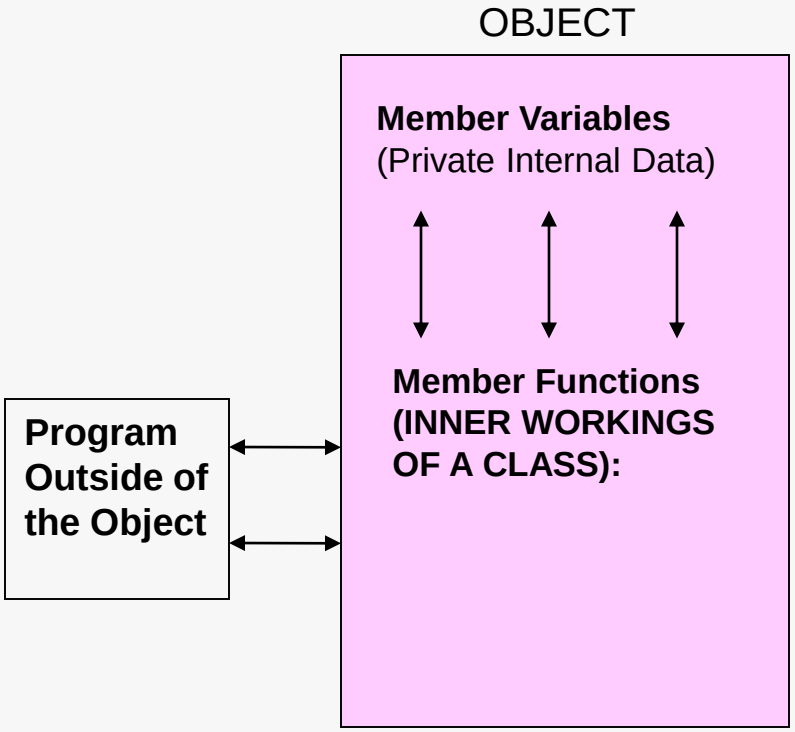
- Object Diagram
  - **When we create an instance (specific tangible example) of the class we create an object**
    - Each object created has its own copies of the class's member variables with values specific to that object
    - The class's member functions are not actually copied when an object is created of a class type but conceptually each member function operates only an object's copy of the member variables
  - Communication between Member Attributes and Member Functions
  - Communication between the Outside program
    - Usually is done through member functions
    - Access to the member attributes is usually restricted so that only members of the class can access them (code outside the class can only access information (usually only methods) in the object the class allows it to)
- This Diagram Illustrates some of the key Concepts of OOP (explained in next slide)



Object Oriented Programming and Objects

Concepts of OOP

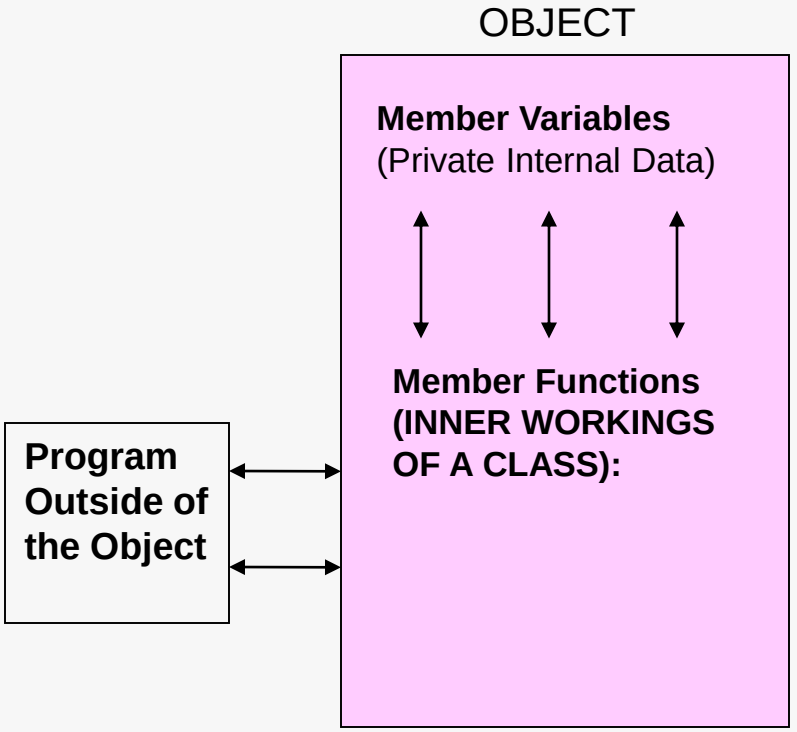
- ***Encapsulation:*** The combining/ bounding all of an object's data and the functions that operate on that data into a single object.
  - Encapsulation allows for **data hiding** and **data abstraction** in programs
  - ***Data Abstraction:*** An object can be used without understanding the details of how it works.
    - (Encapsulation makes this possible by allowing code outside the object to use the objects functions without knowing the details of how the functions work)
    - Car Example
  - ***Data Hiding:*** An object's ability to restrict outside code from accessing their inner workings and data.
    - (Encapsulation makes this possible through data abstraction – Only member functions need to access member attributes and outside code can access member functions without understanding their details or the data behind their details)
    - Projects an object's critical data from outside corruption
    - Hides an objects complexity from the outside program making it easier to use
- ***Object Reusability:*** An object is not a stand alone program – an object is used by programs that need its service. An object can be reused by many different programs in need of it's service
  - 3D Image Rendering Object
    - Written by 3D Imaging Specialist
    - Used by Programmer writing a program for an architectural firm that needs the program they are designing to display 3D images of buildings



Object Oriented Programming and Objects

Concepts of OOP

- Three Basic Concepts of OOP
  - ***Encapsulation***
    - The combining/ bounding all of an object's data and the functions that operate on that data into a single object.
    - This lecture and Chapter 10 in your book focus on encapsulation
- When you add the ability to define relationships between different classes (see Chapters 12 in your book) the next two IMPORTANT OOP concepts come into play
  - (we probably won't get to this, this is covered in CSIS 137 and CSIS 154)
  - ***Inheritance***
    - The ability to create/derive classes of objects from (based on) other classes
  - ***Polymorphism***
    - The ability to determine the behavior of a member function based on which object calls it



Object Based vs. Object Oriented Programming

- We are not truly doing object oriented programming until we add the ability to define relationships between different classes of objects (inheritance and polymorphism)

- What we are doing in this lecture is more accurately referred to as **object based programming**
  - Understanding this object based programming is the key to understanding the true object oriented programming

Building a Class in C++

- A **`class`** is a ***blueprint* or *definition*** for an object (Abstract Data Type)
  - A class is similar to a struct in that it is a data type defined by the programmer
    - A class consists of both member variables and member functions
    - A classes members are by default private
- General Format of a class declaration

```
class ClassName     //Class declaration begins with the keyword class and
                    //is followed by the class name
{                   //the curly braces define the scope of the class
                    //declarations for the class member
                    //attributes and class member functions go
                    //here inside the curly braces

};          //notice the required semicolon after the closing
            //curly brace
```

Building a Class in C++

- We will learn how to implement a class by building one step by step
  - We are going to build the Circle class depicted in previous slides

- **Building a Simple Class**
- **Step 1:** Plan for the High Level Conceptual Object you are creating. This is the phase of programming where you are planning your design you are going to use to write your formal class declaration.
- We are planning the design for a Circle class.
  - This class will define the data (member variables) of a Circle and the functions that can act upon that data. Programs outside of the class (clients) will be able to use the class to create circle objects (instances of the class).

  - The Circle class should have data that describes a Circle
    - Circle data (member variables/attributes):
      radius (hold the circle object's radius)

  - The Circle class should have functions that act on the Circles member variables (perform useful calculations specific to a Circle)
    setRadius (store a value in an object's radius member variable)
    getRadius (return the value in an object's radius member variable)
    (these methods are necessary because we do not want to allow other classes or methods to directly access and change the Circle class's member variables – ***data hiding***)

    calcArea – this method will return the area of a Circle of which is the result of PI * the objects' radius squared

Building a Class in C++

- We will learn how to implement a class by building one step by step
  - We are going to build the Circle class depicted in previous slides

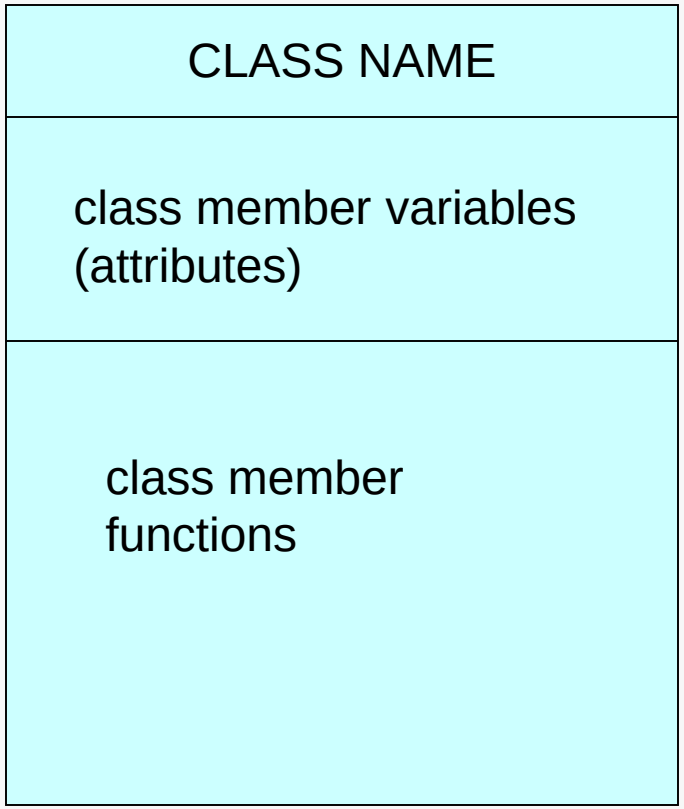
- Building a Simple Class
- **Step 1 Continued:** Plan for the High Level Conceptual Object you are creating. This is the phase of programming where you are planning your design you are going to use to write your formal class declaration.
- When designing a class it's often useful to create a UML diagram

- **UML Diagrams**
  - UML stands for Unified Modeling Language
  - It is used to graphically depict object oriented programs
  - This comes in useful for really big programs with large numbers of objects communicating with each other
    - Objects can also have special relationships with each other that can be communicated graphically though a UML diagram such as Inheritance
  - In the real world problems/programs are so complex designing the UML diagram is a major part of the work
    - Poorly designed object diagrams can lead to a poorly designed program that is not easy to use or re-use…it could also lead to the re-design of a program
    - There is an entire class at CSUN just on Object Oriented Design and UML
  - So we are going to learn just some of the very basic concepts of UML which will seem unnecessary for the simple programs we are designing but once you start designing more complex programs it will make sense why we learned it.
  - Object Oriented Design refers to the planning for the various classes in a program
    - UML Depicts the relationships of classes to each other within a program as well as the way classes communicate with each other

Building a Class in C++

- **Building a Simple Class**
- **Step 1 Continued:** Plan for the High Level Conceptual Object you are creating. This is the phase of programming where you are planning your design you are going to use to write your formal class declaration.

- **UML Diagrams**
  - General Layout for a UML Diagram for a single class
    - In a complicated program with several classes there will be several of these class diagrams with connectors between them showing the relationships between the classes in the program
    - Show Example of Video Store UML Diagram

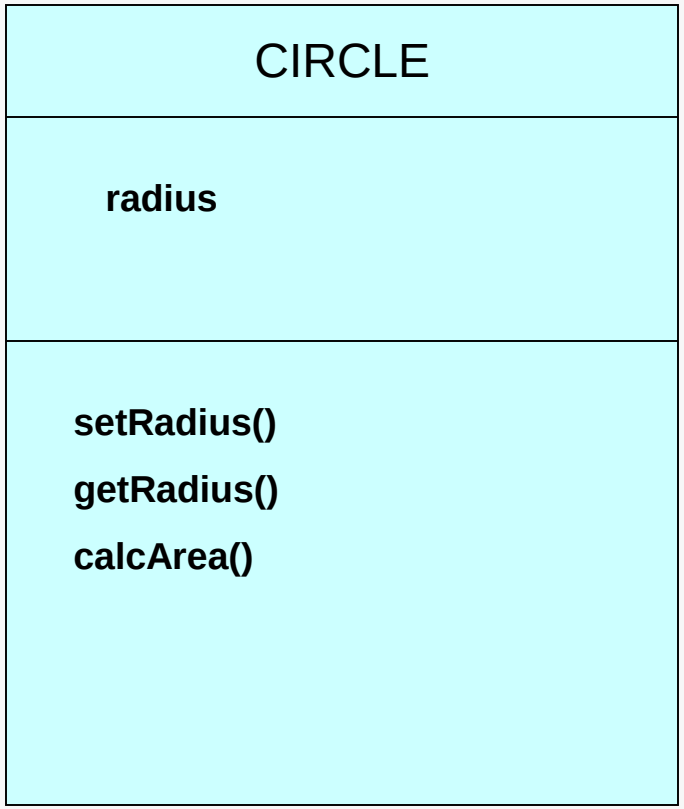


Building a Class in C++

- **Building a Simple Class**
- **Step 1 Continued:** Plan for the High Level Conceptual Object you are creating. This is the phase of programming where you are planning your design you are going to use to write your formal class declaration.

- **UML Diagrams**
  - There is actually a more formal set of rules with which to do these diagrams but for now the general idea is to understand how they work
  - So this is not formal detailed UML but more of a light UML and for now gives you an idea of the concepts
  - Looking at this basic UML Design you can see how it would be easy to extend this object to include more member variables (attributes) and additional functions (modification is easy…remember that was a problem in procedural programming)

- **We'll add details to this UML as we go through these examples – this isn't the final one**

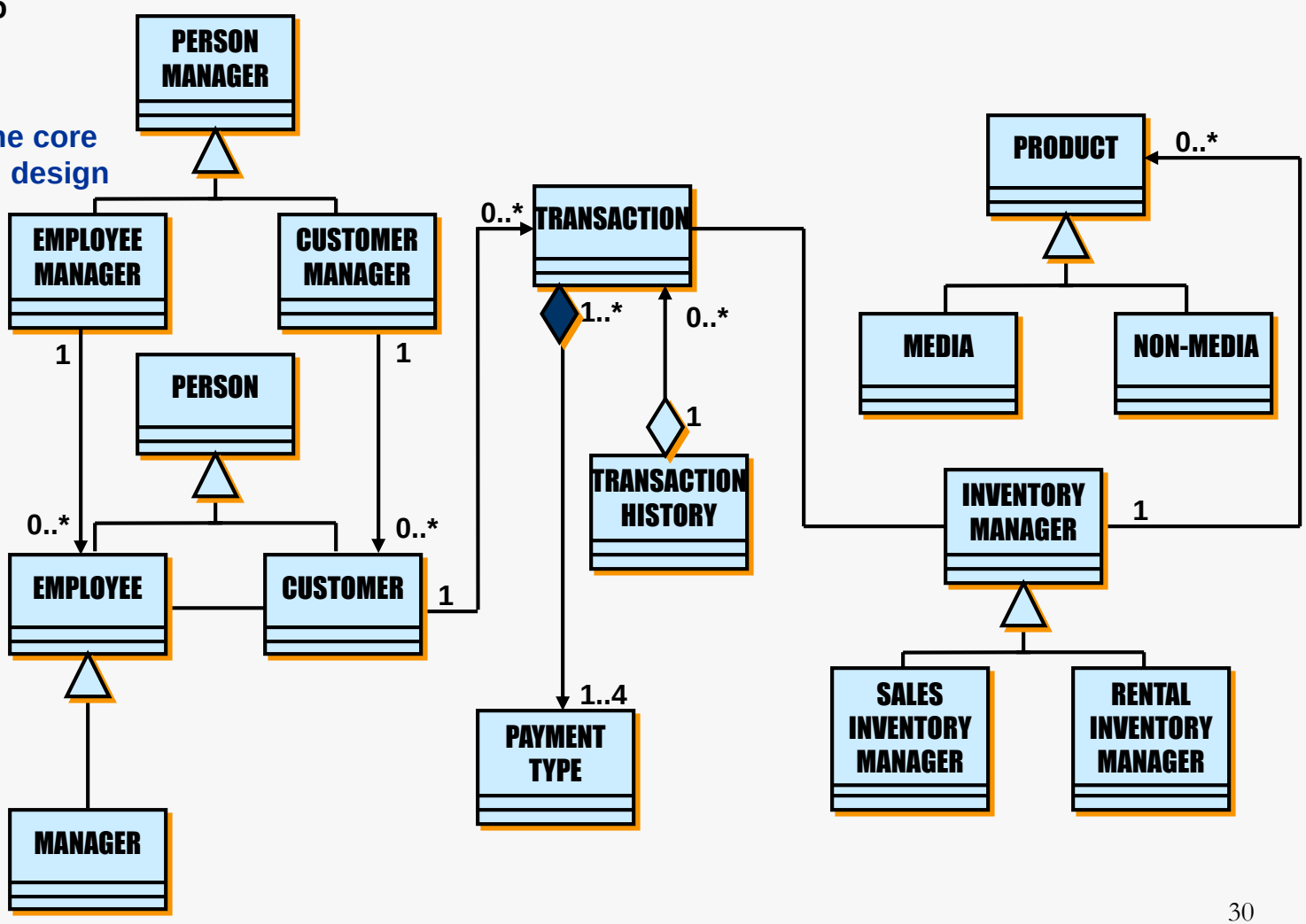


Object Oriented Design
UML EXAMPLE
Video Store

Inheritance: Inheritance allows a new class to extend an existing class. The new class inherits all of the members of the class it extends. Inheritance illustrates an "Is a" relationship between classes.

Designing these relationships is the core of object oriented design

CLASS DIAGRAM: ALL CLASSES



Building a Class in C++

- **Writing the Code for a Simple Class**
- **Step 2:** Using the diagrams and planning you've done for your object start designing the formal class declaration
- General Format for a Class Declaration

```
class Name
{
    private:           //private access specifier
       members;       //private members of the class – There are two types of members:
                      //member variables (attributes) & member functions

   public:            //public access specifier
      members;       //public members of the class – There are two types of
                    //members: attributes (variables) & functions
};
```

- The term **members** refers to the contents of the class, its variables and functions.
- **class** is the keyword telling the C++ compiler that you are defining a class
- Name is the name of your class data type and will be used when other code creates instances of this class

Building a Class in C++

- **Writing the Code for a Simple Class**
- **Step 2:** Using the diagrams and planning you've done for your object start designing the formal class declaration
- General Format for a Class Declaration

```
class Name
{
    private:          //private access specifier
       members;      //private members of the class – There are two types of members:
                     //member variables (attributes) & member functions

    public:           //public access specifier
      members;       //public members of the class – There are two types types of
                     //members: attributes (variables) & functions
};
```

- **public** and **private** are ***access specifiers***
- An ***access specifier*** indicates how a class or class member may be accessed – Some data may be accessed by programs outside the object while other data may only be accessed by other class members (Data Hiding)
  - There are two main types of access specifiers
    - ***Public***
      - A public member variable may be accessed by functions outside of the class and a public member function can be called by functions outside of the class.
      - This means clients can access public members of classes directly (we did this with the dot operator when we used structs and the string class – structs had public members by default).
    - ***Private***
      - A private member variable may ONLY be accessed by a function that is a member of the same class and a private member function may only be called by other functions that are members of the same class.
      - Private members are inaccessible to clients

Building a Class in C++

- **Writing the Code for a Simple Class**
- **Step 2 Continued:** Using the diagrams and planning you've done for your object start designing the formal class declaration

- The Circle Class Declaration – Always start out with your class format and fill in the details as you go

```
class Circle
{
   private:

   public:

};
```

  - Always keep a close eye on syntax and format
    - Notice the colons : after the access specifiers private and public

  - It does not matter whether the private or public members are listed first, they can be listed in any order
    - However, the standard grouping is of private members first followed by public members
      - For this class follow this standard grouping

Building a Class in C++

- **Step 2 Continued:** Using the diagrams and planning you've done for your ADT start designing the formal class declaration

- The Circle Class Declaration – Next add the Member Attributes of the class

```
class Circle
{
   private:
     double radius;   //member variables(attributes) are almost always private

   public:

};
```

  - Note that the attribute of our Circle class has been given a private access identifier

  - This means that the radius variable can only be accessed by class functions inside the Circle class
  - This hides this variable from the code outside this class and prevents code outside of the class from corrupting this variables by changing it in ways they shouldn't purposely or otherwise
    - If a program outside a class attempts to access a private member a compiler error will occur
  - This is an example of how encapsulation (being able to use something without knowing its details) allows data hiding
  - ***It is common practice in object oriented programming to make all of a class's member variables private and only provide access to those variables through functions***
    - ***In this class we will follow this common practice***

Building a Class in C++

- **Step 2 Continued:** Using the diagrams and planning you've done for your object start designing the formal class declaration
- The Circle Class Declaration – Next add the Member Functions of the class

```
class Circle
{
   private:
     double radius;

   public:
      void setRadius(double r)
      {
         if(r > 0)
            radius = r;
         else
            radius = 0;
      }

      double getRadius()
      {
         return radius;
      }

      double calcArea()
      {
         return 3.141592 * pow(getRadius(),2);
      }
};
```

Look closely at the `calcArea()` function it demonstrates that the members of the class can see and access each other, no dot operator needed WITHIN the class….sometimes this can come in very handy in code…member functions can reuse other member functions when processing an instance/object

- Note that these functions look just like the functions we learned to write in Chapter 6 (each function has a return type, a name, and parameters for input data)
  - The exception to this rule is constructors, which is in the Structures Lecture – we'll go over constructors again here also
- `public` indicates that these functions may be accessed seen and called by code outside of this class (clients)…the member functions can be thought of as the way clients (outside code) communicate with the class
  - These functions are created in a way that should allow carefully controlled access to the class
    - For example, it would be a good idea in the `setRadius` function to not allow negative data (in order to not allow the radius attribute to be negative since this would not make sense in the class and would produce meaningless results

Building a Class in C++

- **Step 2 Continued:** Using the diagrams and planning you've done for your object start designing the formal class declaration
- The Circle Class Declaration – Next add the Member Functions of the class
  - **Changing the setRadius function to disallow negative input**

```
class Circle
{
   private:
     double radius;

   public:
      void setRadius(double r)
      {
         if(r > 0)
            radius = r;
         else
            radius = 0;
      }

      double getRadius()
      {
         return radius;
      }

      double calcArea()
      {
         return 3.141592 * pow(getRadius(),2);
      }
};
```

  - `setRadius` now sets the radius attribute of the class to the number input by the user if it is positive, otherwise sets it to zero

Building a Class in C++

- **Step 2 Continued:** Using the diagrams and planning you've done for your object start designing the formal class declaration
- **The Circle Class Declaration – Completed**

```
class Circle
{
   private:
     double radius;

   public:
      void setRadius(double r)
      {
         if(r > 0)
            radius = r;
         else
            radius = 0;
      }

      double getRadius()
      {
         return radius;
      }

      double calcArea()
      {
         return 3.141592 * pow(getRadius(),2);
      }
};
```

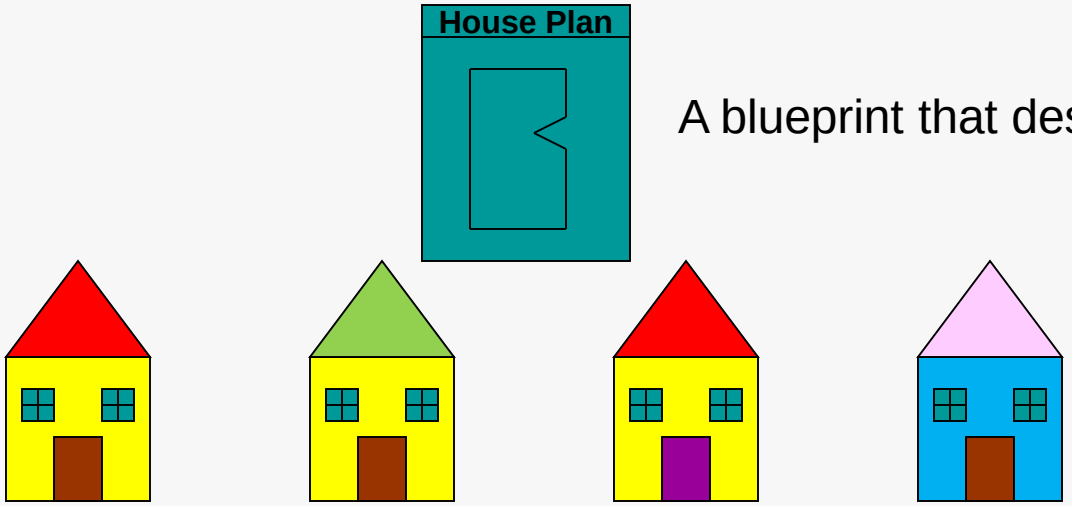
Building a Class in C++

- What if the **`private`** and **`public`** words had been omitted in the class declaration?
  - Everything would have defaulted to private
    - This would not have been very useful because except in special circumstances because no functions outside of the class would ever use the class

- Structs can also declare members to be public or private but we normally do not use access specifiers with structs
  - This is ***because by default all members of structures are*** **`public`** and that is usually what is wanted when a **`struct`** is used

Using a Class in C++ - Objects

- **Concept:** Objects are instances of classes. They are created with a definition statement after a class has been declared
  - Like a **`struct`** declaration, the Circle class declaration we wrote does not create variables (objects) of the class type. Class type variables (objects) are created using ordinary declarations
    - `Circle circ1;`
    - `Circle circ2;`
  - A class declaration by itself does not create an object but is merely a description or definition of an object
  - Think of a class declaration as being similar to a blueprint for a house
    - The blueprint itself is not a house but is a detailed description of a house
    - When a blueprint is used to build an actual house we could say we are building an instance of the house described by the blueprint
    - Several houses could be built from the same blueprint
  - **A class declaration serves a similar purpose. We can use it to create one or more objects, which are instances of classes – all the objects have the same general characteristics but can have different specifications of each characteristic**
    - like the different colored roofs, each house has a roof with a color but the specific color can be different



A blueprint that describes a house

Instances of the house described by the blueprint

Using a Class in C++ (Creating Objects)

- Writing code that creates an instance of the class and uses it in a program
- **Step 3:** Using the class by creating an instance of it

- Let's see how the Circle class could be used in a main method by creating Circle objects (instances of the circle class)
  - Class Objects are created with simple definition statements, just like variables. The following statements defines `circ1` and `circ2` to be two members of the Circle class:

```
Circle circ1;
Circle circ2;
```

  - Defining a class object is called the ***instantiation*** of a class

  - **The objects `circ1` and `circ2` are two distinct instances of the `Circle` class with different memory assigned to hold the values of their member variables**

Using a Class in C++ (Creating Objects)

- Writing code that creates an instance of the class and uses it in a program

- **Step 3:** Using the class by creating an instance of it (a circle object) and accessing that object's member functions

  - After Circle Objects are created, the code outside the class can then access the circle object's members (**only the public ones**) – so through this public member function the outside code (client) can see and manipulate the object's member variables
    - The members of a class object are accessed with the ***dot operator***
      - Think of the ***dot operator*** as a built in class operation for member selection (**only public member selection is allowed**)
        - Since in a class only the functions are public that means the dot operator is going to be used to access an object's member functions and perform operations on an object's member variables

    - The following statements call the `setRadius` member function of circ1 and circ2

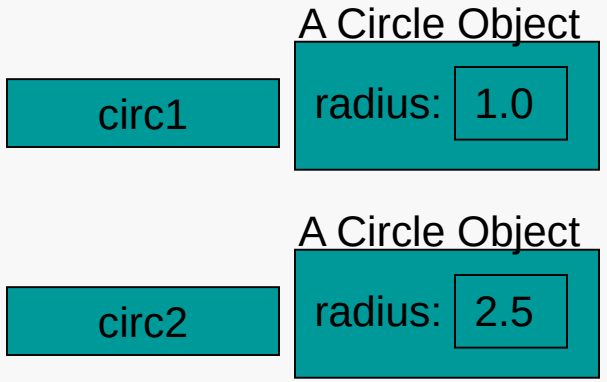
```
circ1.setRadius(1);
circ2.setRadius(2.5);
```

Using a Class in C++ (Creating Objects)

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

class Circle
{
   private:
     double radius;

   public:
      void setRadius(double r)
      {
         if(r > 0)
            radius = r;
         else
            radius = 0;
      }

      double getRadius(double r)
      {
         return radius;
      }


      double calcArea()
      {
         return 3.141592 * pow(getRadius(),2);
      }
};

int main()
{
   Circle circ1;
   Circle circ2;

   circ1.setRadius(1);
   circ2.setRadius(2.5);

   return 0;
}
```

circ1, circ 2 variables

(slightly different than Java, the variables while they still refer to the location of the object in memory are not reference variables, you will see later objects are by default pass by value in C++, to pass by reference you will need the &)



What if we want to print out the radius members
Of each object (circ1 and circ2)?

- This won't work because radius is private
  - `cout << circ1.radius;` //illegal

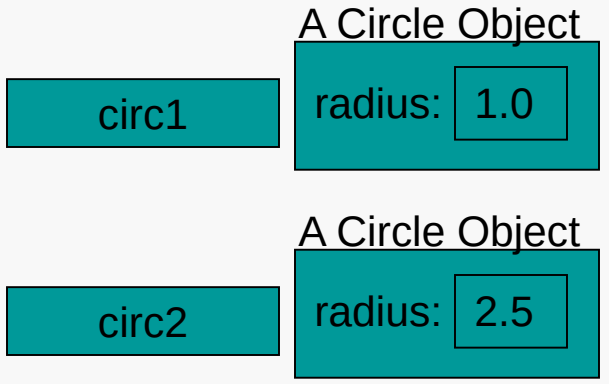
- We need to use the **`getRadius`** function which allows us to retrieve the class's private member variable radius – so we can retrieve it but not change/manipulate it

EXAMPLE #1

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

 class Circle
 {
    private:
      double radius;

    public:
       void setRadius(double r)
       {
          if(r > 0)
             radius = r;
          else
             radius = 0;
       }

       double getRadius()
       {
         return radius;
       }

       double calcArea()
       {
          return 3.141592 * pow(getRadius(),2);
       }
 };

 int main()
 {
    Circle circ1;
    Circle circ2;

    circ1.setRadius(1);
    circ2.setRadius(2.5);

    cout << "The radius of circ1 is " << circ1.getRadius() << endl;
    cout << "The radius of circ2 is " << circ2.getRadius() << endl;
    system("PAUSE");
    return 0;
 }
```

circ1, circ 2 variables

(slightly different than Java, the variables while they still refer to the location of the object in memory are not reference variables, you will see later objects are by default pass by value in C++, to pass by reference you will need the &)



NOTE: we are eventually going to divide our object oriented projects into THREE files – don't stop reading the lecture here, you are required to use the three files we learn later in the lecture.

Using a Class in C++ (Creating Objects)

- Now we can add code to the main function to calculate and print out the area of each circle (circ1 and circ2 using the calcArea() method)

- **EXAMPLE 2: ObjectOrientedCircle2.cpp**

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

class Circle
{
  //Circle declaration statements here same as before
  //omitted on this slide to save space
};

int main()
{
   Circle circ1;
   Circle circ2;

   circ1.setRadius(1);
   circ2.setRadius(2.5);

   cout << "The radius of circ1 is " << circ1.getRadius() << endl;
   cout << "The radius of circ2 is " << circ2.getRadius() << endl;

   cout << "The area of circ1 is " << circ1.calcArea() << endl;
   cout << "The area of circ2 is " << circ2.calcArea() << endl;

   system("PAUSE");
   return 0;
}
```

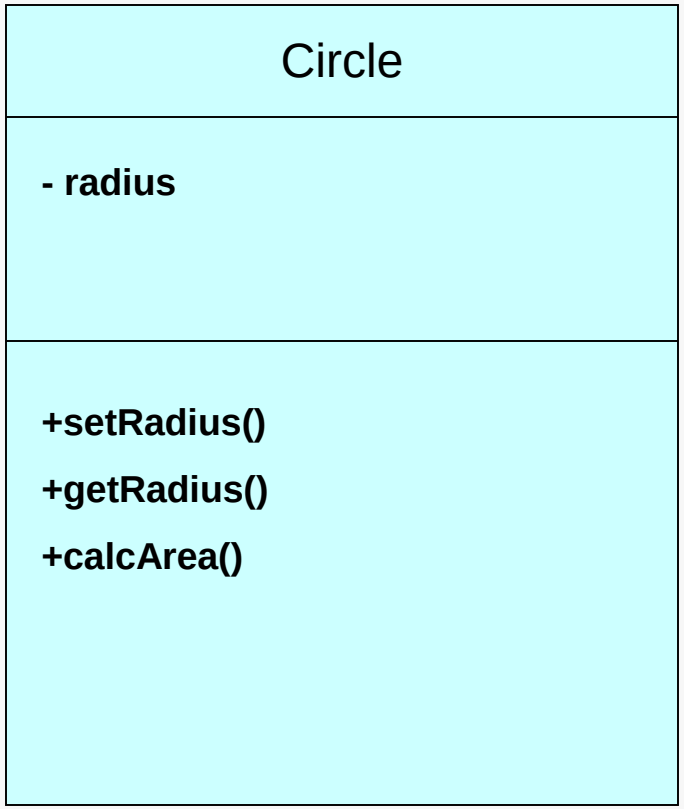
Using a Class in C++ (Creating Objects)

- Writing code that creates an instance of the class and uses it in a program
- **Step 3:** Using the class by creating an instance of it (a circle object) and accessing that object's member functions

- Notice that member functions do not need the dot operator to reference member variables ***or member functions*** ***within the same class*** (look at the `calcArea` member function). They are able to access `radius` as if were a regular variable.
  - ***Important idea, member functions of a class can re-use other member functions of the same class (USE THIS FOR HW #7)***
  - Only code **outside** the class needs the dot operator to use objects created from the class to access that object's public members.

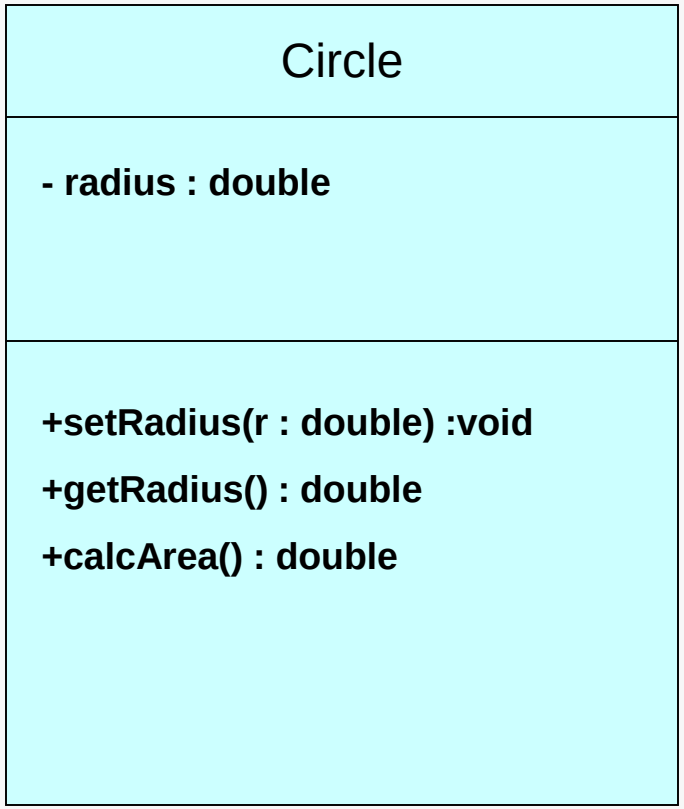
Using a Class in C++ - More UML

- Showing Access Specification in UML Diagrams
  - The basic UML diagram we completed before coding the Circle class listed all of the members of a class but did not specify the access specification of the members (which members were public and which private)
  - In a UML diagram you have the option of showing the access specifier of each member
    - A "-" before a class member indicates it is a private member
    - A "+" before a class member indicates it is a public member



Using a Class in C++ - More UML

- Showing Data Type and Parameter in UML Diagrams
  - In a UML diagram you have the option of indicating the data types of member variables, member functions, and parameter variables
    - 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
    - The Return type of a function can be indicated by placing a colon followed by the return data type after the function's ()
    - Parameter variables and their data types may be listed inside a functions's parenthesis
- **This is a completed UML diagram that includes all of the details for the circle class**



Defining Member Functions

- So far all of the functions we have written have been in the class declaration itself
  - When a class function is defined within the class declaration it is called an ***inline function***
    - ***Inline functions*** provide a convenient way to contain function information within a class declaration ***but they can only be used when a function body is short*** (usually no longer than a single line is a good guideline)
      - We will go over why this is in a few slides – ***IMPORTANT:*** ***In this class I don't want you to use inline functions even if the function is short (points off!)***
  - When a function body is longer than a single line we can NOT use inline functions (meaning we cannot define our functions inside our class definition)
    - ***We also shouldn't use inline functions because we want our implementation details of our class to be separate from how it works so instead:***
      - The **function prototype** **should appear in the class declaration** and the **function definition** **should be placed outside the class declaration either following it or in a separate file**
  - So Class construction has two components:
    - **Declaration (specification) section**: Declares member variables and inline functions or function prototypes for the class member functions
    - **Implementation section**: Defines functions whose prototypes were declared in declaration section
      - This can be in another file (more common way which we will see in a minute) or in the same file as the class declaration following the declaration section
      - Note we are beginning to separate the what and how to use a class from the details of how its implemented, remember the user of the class (the client) only needs to know the what and the how, not the implementation details

Class Construction

FIGURE 12.2 *Format of a Class Definition*

```
// class declaration section
class classname
{
   data members    //    instance variables
     and
   function members    //    inline and prototypes
};
// class implementation section
function definitions
```

Defining Member Functions

- **Example:** Defining function prototypes in the class declaration and defining the functions following the declaration section (in the implementation section)

  - This is an example using the circle class we created even though the functions in that class are short enough to be inline functions
    - Look ahead two slides to ***Example 3***

  - Inside the class ***declaration (also called the specification)*** the functions are be replaced by the following prototypes:

```
void setRadius(double);
double getRadius();
double calcArea();
```

Defining Member Functions

- **Continued:** Example of Defining function prototypes in the class declaration and defining the functions following the declaration section (in the implementation section)
  - Following the class declaration a **function** ***implementation section*** is added containing the following function definitions:

```
/*IMPLEMENTATION SECTION:  Contains function definitions*/
void Circle::setRadius(double r)
{
   if(r > 0)
      radius = r;
   else
      radius = 0;
}

double Circle::getRadius()
{
   return radius;
}

double Circle::calcArea()
{
   return 3.141592 * pow(radius,2);
}
```

- These look like ordinary functions except for one thing:
  - They contain the class name and a double colon (`::`) before the function name (but after the function type)
  - The :: is the ***scope resolution operator*** and is needed to indicate that these are class member functions and to tell the compiler which class they belong to
  - This :: operator is required in this exact format in the ***IMPLEMENTATION SECTION*** of a class

- **EXAMPLE 3: ObjOrCircle_implementSec.cpp**

```
//preprocessor directives go here, were omitted to save space for this slide
class Circle
{
   private:
     double radius;

   public:                  //function prototypes
     void setRadius(double);
     double getRadius();
     double calcArea();
};

 /*IMPLEMENTATION SECTION:  Contains function definitions*/
 void Circle::setRadius(double r)
 {
    if(r > 0)
       radius = r;
    else
       radius = 0;
 }

double Circle::getRadius()
{
   return radius;
}

double Circle::calcArea()
{
   return 3.141592 * pow(radius,2);
}

int main()  //note the spacing here is bad, changed for these slides, add whitespace for readability
{
   Circle circ1;
   Circle circ2;
   circ1.setRadius(1);
   circ2.setRadius(2.5);
   cout << "The area of circ1 is " << circ1.calcArea() << endl;
   cout << "The area of circ2 is " << circ2.calcArea() << endl;
   system("PAUSE");
   return 0;
}
```

Note these are still in the same file as the declaration but we will change this

Defining Member Functions – Inline Functions

- When designing a class (in the outside world) you will need to decide which member functions to write as inline functions and which ones to define outside of the class in the implementation section
  - ***REMEMBER IN THIS CLASS WE ARE NOT USING INLINE FUNCTIONS (points off!)***

- Inline functions are handled completely differently by the compiler than regular functions are
  - Understanding this difference will help you decide when to use which type of function in the real world

Defining Member Functions – Inline Functions

- ***Regular Functions:*** how they are handled by the compiler
  - When a regular function is called a number of special items including but not limited to the address to return to when the function is finished executing and the values of the function arguments must be stored in a special section of memory called the **stack**.
    - In addition, any local variables created in the function need to be stored on the stack as well as a location for the function's return value
    - All of this overhead that sets the stage for a regular function call takes CPU time
      - Although this CPU time is miniscule, it can add up if the function is called many times such as in a loop

- ***Inline functions:*** how they are called by the compiler
  - An inline function is not called in the same way as a regular function
  - Instead a process called ***inline expansion*** is used where the compiler replaces every call to the function with the actual code of the function itself
    - This means that if the function is called from multiple places in the program the entire body of its code will be inserted into the program increasing the size of the program itself
    - This is why only a function with very few lines of code should be written as an inline function
      - In fact, if the inline function is too large to make the inline expansion practical, the compiler will ignore the request to handle the function this way
      - However when a member function is small it can improve performance to write it as an inline function because of the reduced overhead of not having to make actual function calls

- ***IMPORTANT!!: For practice in this class, treat ALL functions as regular functions and use prototypes in the class declaration and have an implementation section (which we will implement as a separate file – shown in upcoming slides)***
  - Know however, how an inline function would be used in the real world

Defining Member Functions

- **Get and Set Functions**
  - It is common practice to make all of a class's member variables private and to provide public functions for accessing and changing those fields
    - This ensures that an object is in control of all of the changes being made to its member variables
      - Remember the example in the Circle class where the `setRadius` method made sure the value the user tried to set the `radius` to was positive
    - A function that gets a value from a class's attribute (member variable) but does not change it is known as an **Get Function**
    - A function that stores a value in an attribute (member variable) or changes the value of an attribute (member variable) is known as a **Set Function**
    - In the Circle Class
      - **Get Method:** `getRadius`
      - **Set Method:** `setRadius`
  - **In most cases all class members variables (attributes) should have a get and set method since in most cases they will be private**
    - **For this class make all data members of a class private and have each data member have a get and set method**
      - (unless there is a very good reason a member should never be changed in which case you wouldn't want a set method or there is a very good reason a member shouldn't be inspected, in which case you wouldn't want a get function)

Another Example: A Rectangle Class

- **Step 1:** Plan for the High Level Conceptual Object you are creating. This is the phase of programming where you are planning your design you are going to use to write your formal class definition.

- We are planning the design for a Rectangle class.
  - This class will define the data (member variables) of a Rectangle and the functions that can act upon that data. Programs outside of the class will be able to use the class to create rectangle objects (instances of the class).

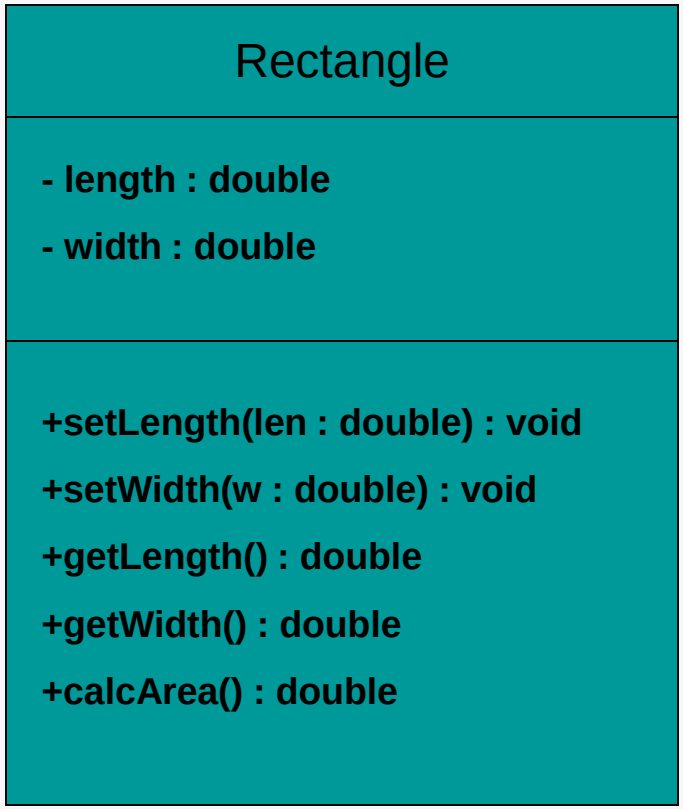
  - The rectangle class should have data about a rectangle and methods that can act upon that data.
    - Rectangle data (member variables):
      length (hold the rectangle object's length)
      width (hold the rectangle object's width)

  - The rectangle class should have functions that act on the rectangle's member variables (perform useful calculations specific to a rectangle)
    setLength (store a value in an object's length field)
    setWidth (store a value in an object's width field)
    getLength (return the value in an object's length field)
    getWidth (return the value in an object's width field)
    (these methods are necessary because we do not want to allow other classes or methods to directly access and change the rectangle class attributes – **data hiding**)

    calcArea – this method will return the area of the rectangle which is the result of the object's width multiplied by its length

Another Example: A Rectangle Class

- ***Step 1:*** Plan for the High Level Conceptual Object you are creating. This is the phase of programming where you are planning your design you are going to use to write your formal class definition.
- **UML Diagram for Rectangle Class**
  - Note this is another example of a completed UML diagram…this is CLOSE to what you will need to do for HW #7 – you will need to add the constructor function (constructors are coming up)



Another Example: A Rectangle Class

- **Step 2:** Using the diagrams and planning you've done for your object to write the class declaration (member variables and function prototypes)

- Example filename: (***Example 4***)

```
/*CLASS DECLARATION – Member Variables & Function
   prototypes*/
class Rectangle
{
   private:                    //make the attributes private
     double length;
     double width;

   public:                    //function prototypes
     void setLength(double);
     void setWidth(double);
     double getLength() const;
     double getWidth() const;
     double calcArea() const;
};
```

Another Example: A Rectangle Class

- Notice that in my class declaration I have added the keyword **`const`** after the following member function prototypes

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

- By declaring a classes member variables private, a class can offer a reliable product because it knows external access (by the client) to private data is impossible

- Often it is acceptable to let the client inspect (but not modify) private member variables through certain member functions
  - These functions are often called observer functions
  - Because these functions are not intended to modify the private data they inspect they are declared with the keyword **`const`** following the parameter list

- Within the body of a **`const`** C++ member function, a compile time error occurs if any statement tries to modify a private data member

- ***Although not required by the language, it is good practice to declare as `const` the member functions that do not modify private data*** ***(we use good design and practices in this class – or we lose points)***
  - This increases readability

we will see more benefits of `const` in Advanced C++ (CSIS 137) – get in the practice of using them – this is just the first reason to use `const`

Another Example: A Rectangle Class (Example 4)

- **Step 3:** Using the diagrams and planning you've done for your object to write the implementation section (function definitions)

```
/*IMPLEMENTATION SECTION - MEMBER FUNCTION DEFINITONS*/
/*****************************************************************************
                        Rectangle::setLength
This function sets the value of the member variable length.  If
the argument passed to the funcion is greater than zero it is copied into length.
If the argument passed to the function is negative, 1.0 is assigned to length
*****************************************************************************/
void Rectangle::setLength(double len)
{
   //do not allow for a negative length
   if(len >=0)
      length = len;
   else
   {
      length = 1.0;
      //NOTE:  using cout in a member function is BAD DESIGN, we'll fix this later
      cout << "Invalid length.  Using a default value of 1.0.\n";
   }
}
/*****************************************************************************
                        Rectangle::setWidth
This function sets the value of the member variable width.  If
the argument passed to the funcion is greater than zero it is copied into width.
If the argument passed to the function is negative, 1.0 is assigned to width
*****************************************************************************/
void Rectangle::setWidth(double w)
{
   //do not allow for a negative width
   if(w >=0)
      width = w;
   else
   {
      width = 1.0;
     //NOTE:  using cout in a member function is BAD DESIGN, we'll fix this later
      cout << "Invalid width.  Using a default value of 1.0.\n";
   }
}
```

Another Example: A Rectangle Class

- **Step 3 Continued:** **Using the diagrams and planning you've done for your object to write the implementation section (function definitions)**

```
/*IMPLEMENTATION SECTION CONTINUED - MEMBER FUNCTION DEFINITONS*/
/*************************************************************************
                        Rectangle::getLength
This function returns the value that is in the private member length
*************************************************************************/
double Rectangle::getLength()const
{
  return length;
}

/*************************************************************************
                        Rectangle::getWidth
This function returns the value that is in the private member width
*************************************************************************/
double Rectangle::getWidth()const
{
  return width;
}

/*************************************************************************
                        Rectangle::calcArea
This function calculates and returns the area of the rectangle
*************************************************************************/
double Rectangle::calcArea()const
{
  return length * width;
}
```

Another Example: A Rectangle Class

- **Step 3:** Using the class by creating an instance of it (a circle object) and accessing that object's member functions

```
/**************************************************************************
                                main - The Client
**************************************************************************/
int main()
{
   Rectangle box; //declare a rectangle object
   double boxLength, boxWidth;

   //get box length and width from the user using prompts
   cout << "This program will calculate the area of a rectangle." << endl;
   cout << "What is the length of the rectangle?";
   cin  >> boxLength;
   cout << "What is the width of the rectangle?";
   cin  >> boxWidth;

   //call the member functions to set the box dimensions
   box.setLength(boxLength);
   box.setWidth(boxWidth);

   //call box's member functions to get the box information to display
   cout <<"\nHere is the Rectangle's data:\n";
   cout << fixed << setprecision(2);
   cout <<"Length: " << box.getLength() << endl;
   cout <<"Width: " << box.getWidth() << endl;
   cout <<"Area: " << box.calcArea() << endl;

   system("PAUSE");
   return 0;
}
```

Rectangle Class - Avoiding Stale Data

- Avoiding Stale Data
  - In the Rectangle class the **`calcArea`** Function returns the results of calculations
  - You might be wondering why the area of the rectangle is not stored as an member variable like the length and width are
    - The reason is because this attribute could eventually become stale
      - What would happen if the area had been calculated and stored in an area attribute in the class and then later the length or width attributes were changed?
      - Since the area attribute would not be automatically updated every time length or width changed it could potentially become incorrect
      - When the value of an item is dependent on other data and is not automatically updated when that other data is changed it is said that that item has become stale
      - If area were stored in as an attribute the value of this attribute would become incorrect as soon as either the length or width fields were changed. (based on the way the Rectangle class has been designed)

C++ Class Design

- **Concept:** Usually class declarations (specifications) are stored in their own ***header files (.h)*** and the implementation section (function definitions) are store in their own ***implementation .cpp files*** (C++ source files)
  - All of the programs we have written so far have the class declaration, the function definitions (implementation section), and the application program (main in our case) all in one file
  - The more conventional way of designing a C++ program is to store the class declaration section (the specification) and the implementation section in their own separate files
    - In addition, since there will be many different clients using the class (classes should be reusable) each client program should be saved in a different file
  - ***An abstract data type has two parts:*** a specification (declaration) and an implementation
    1. ***The class declaration (***specification) describes the behavior of the ADT without reference to it's implementation
       - In other words the class declaration describes what the class does without going into detail about how it works (data abstraction)
       - A client can use a class looking only at its declaration
    2. ***The class implementation*** creates an abstraction barrier by hiding the concrete data representation as well as the code for the operations

C++ Class Design

- Typically C++ program components are stored in the following fashion:
  1. Class declarations are stored in their own header files. A header file that contains a class declaration is called a ***class specification (or declaration) file.***
     - In principle the specification/declaration file should not reveal any of the implementation details to the user of the class, so ***the file should specify what each member function does but NOT how***
     - The name of the class specification file is usually the same as the name of the class with a .h extensions
     - For example, the Rectangle class would be declared in the file **`Rectangle.h`**
  2. The member function definitions for a class are stored in a separate .cpp file which is called the ***class implementation file***.
     - ***This is where the HOW is defined – client programs do not need access and should not have access***
     - The class implementation file also usually has the same name of the class with the .cpp extension
     - The implementation file will need an **`#include`** preprocessor directive to properly link it with its corresponding header file
     - The Rectangle class's member functions would be defined in the file **`Rectangle.cpp`**
  3. Any client program that uses the class should include the class's header (class specification) file using a **`#include`**. The class's .cpp file should be complied and linked with the main program. This process can be automated with a **`project`** in Visual C++. Visual C++ projects are set up to work this way.

Rectangle Example Revision

- The Rectangle Class & Program created earlier will now be revised in two ways

1. It will the **`setLength`** and **`setWidth`** functions to only set the length and width if values the user input are positive. Instead of changing incorrect data to 1.0 the functions return **`bools`**.
   - If a valid arguments is received, the argument is stored in the proper member variable (attribute) and the function returns **`true`**
   - If an invalid argument is received the member variable (attribute) is left unchanged and **`false`** is returned
     - This allows the client program to test the returned **`bool`** value and determine how to proceed in the case of invalid data
     - This safeguards data input as well as ***leaves control to the client program making it more reusable*** ***(USE THIS METHOD! – DESIGN MATTERS – it does effect final grades on programmming exams)***

2. It will use separate files for the Rectangle class specification file (class declaration), the class implementation file (implementation section with function definitions), and the client program that uses the class

Rectangle Example Revision

- **Revised setLength and setWidth functions (Example 5)**

```
/*****************************************************************************
                        Rectangle::setLength
If the argument passed to the setLength function is zero or greater it is copied
into the member variable length and true is returned.  If the argument is negative,
the value of length remains unchanged and false is returned
*****************************************************************************/
bool Rectangle::setLength(double len)
{
   bool validData;  //local variable

   if(len >=0)  //if length is positive data is valid
   {
      length = len;  //set the member variable
      validData = true;
   }
   else  //else data is invalid
      validData = false;

   return validData;  //return to calling program whether or not data is valid
}
/*****************************************************************************
                        Rectangle::setWidth
If the argument passed to the setWidth function is zero or greater it is copied
into the member variable length and true is returned.  If the argument is negative,
the value of width remains unchanged and false is returned
*****************************************************************************/
bool Rectangle::setWidth(double w)
{
   bool validData;  //local variable

   if(w >=0)  //if length is positive data is valid
   {
      width = w;  //set the member variable
      validData = true;
   }
   else  //else data is invalid
      validData = false;

   return validData;  //return to calling program whether or not data is valid
}
```

Rectangle Example Revision

- Revised Main to use new setLength and setWidth functions (Example 5)

```
int main()
{
   Rectangle box; //declare a rectangle object
   double boxLength, boxWidth;

   //get box length and width from the user using prompts
   cout << "This program will calculate the area of a rectangle." << endl;
   cout << "What is the length of the rectangle?";
   cin  >> boxLength;
   cout << "What is the width of the rectangle?";
   cin  >> boxWidth;

   //call the member functions to set the box dimensions,
   //check to see if the member functions return false to see if data is invalid
   while(!box.setLength(boxLength))  //while box.setLength returns false
   {
     cout << "\nInvalid box length entered.  Please enter a positive box length\n";
     cout << "What is the length of the rectangle?";
     cin  >> boxLength;
   }

   cout << "\nWhat is the width of the rectangle?";
   cin  >> boxWidth;
    while(!box.setWidth(boxWidth))
   {
      cout << "\nInvalid box width entered.  Please enter a positive box width\n";
      cout << "What is the width of the rectangle?";
      cin  >> boxWidth;
   }

   //call box's member functions to get the box information to display
   cout <<"\nHere is the Rectangle's data:\n";
   cout << fixed << setprecision(2);
   cout <<"Length: " << box.getLength() << endl;
   cout <<"Width:  " << box.getWidth() << endl;
   cout <<"Area:   " << box.calcArea() << endl;
   return 0;
}
```

Rectangle Example Revision

- Now we want to revise to use the three separate files for the Rectangle class (specification, implementation, client)
  - The three separate files will all be included in our one project and will compile separately and then link together to form the final executable

- ***How to do this using Visual C++***
  - Create your project in the same way you have been in every other program you did in this class
  - We will add each of the three files to the project: the class specification file, the class implementation file, and a file for the client program that uses the class (I will call this client program rectangleApp.cpp)
  - Make sure when you're adding the files to put the proper extensions on them
    - `Rectangle.h` (the class specification file)
    - `Rectangle.cpp` (the class implementation file
    - `RectangleApp.cpp` (the program that uses the class)

Rectangle Example Revision (Example 5)

- Now we want to revise to use the three separate files for the Rectangle class
  - **The class specification file (header file) rectangle.h**

```
#ifndef RECTANGLE_H
#define RECTANGLE_H

//Rectangle Class Declaration
class Rectangle
{
   private:                  //make the attributes private
     double length;
     double width;

   public:                  //function prototypes
     bool setLength(double);
     bool setWidth(double);
     double getLength()const;
     double getWidth()const;
     double calcArea()const;
};

#endif
```

  - The **#ifndef** directive shown is called an include guard and prevents a header file from accidentally being included more than once when the all of the files in the program are linked together.
  - The **#ifndef** directive checks for an instance of the constant RECTANGLE_H. If the constant has not been defined it is immediately defined with the **#define** directive and the header file is included. If the constant has already been defined everything between the **#ifndef** and **#endif** directives is skipped.
  - **You will want to add more descriptions as to what each function does (remember in most cases this is ALL the client program will see – the implementation is kept separate from how to use the class) – make sure you are describing what and not how**

Rectangle Example Revision – Example 5

- Now we want to revise to use the three separate files for the Rectangle class
  - **The class implementation file rectangle.cpp**

```
/*File rectangle.cpp -- Function Implementation File*/
//Contains function definitions for Rectangle class functions

#include "Rectangle.h"
#include <iostream>
using namespace std;

/*RECTANGLE MEMBER FUNCTION DEFINITONS*/
/*************************************************************************
                        Rectangle::setLength
If the argument passed to the setLength function is zero or greater it is copied
into the member variable length and true is returned.  If the argument is negative,
the value of length remains unchanged and false is returned
*************************************************************************/
bool Rectangle::setLength(double len)
{
   bool validData;  //local variable

   if(len >=0)  //if length is positive data is valid
   {
      length = len;  //set the member variable
      validData = true;
   }
   else  //else data is invalid
      validData = false;

   return validData;  //return to calling program whether or not data is valid
}
```

  - NOTE THIS IS ONLY A PART OF THE CLASS IMPLEMENTATION FILE, THE FUNCTION DEFINITIONS HAVE NOT CHANGED SO THEY ARE NOT INCLUDED HERE
  - Note the line `#include "Rectangle.h"`. This allows this file to use the class declaration we created in the rectangle.h file
  - Notice the use of the scope resolution operator so the compiler knows what class the function belongs to
    - Different classes can have functions with the same name

Rectangle Example Revision

- Now we want to revise to use the three separate files for the Rectangle class
  - **The client program that uses the class rectangleApp.cpp**

```
#include <iostream>
#include <iomanip>
#include "Rectangle.h"
using namespace std;

/*************************************************************************
                              main
*************************************************************************/
int main()
{

  //Nothing inside main has changed so it is not included here
}
```

  - **NOTE THIS IS ONLY A PART OF THE PROGRAM FILE, THE DETAILS INSIDE MAIN HAVE NOT CHANGED SO THEY ARE NOT INCLUDED HERE**

  - Note the line `#include "Rectangle.h"` . This allows this file to use the class declaration we created in the rectangle.h file
  - How does rectangleApp.cpp know about rectangle.cpp?
    - They are both inside the same project and are linked during the compilation process
      - Two object code files are created by compiling rectangle.cpp and rectangleApp.cpp
      - These two object code files are then linked to make the executable file that you see rectangleApp.exe

Class Design Considerations

- **Including your own header files**
  - The program rectangleApp uses the following **`#include`** directive to include the contents of the rectangle.h header file

```
#include "Rectangle.h"
```

  - The name of the header file is enclosed in double quotes(" ") instead of in the angled brackets (< >) like the C++ system file headers are such as iostream
    - The angled brackets indicate that the file to include is located in the compiler's **include file directory**, the directory where all of the standard C++ header files are located
    - To include a header file that you have written yourself such as a class specification file, you enclose the name of the file in double quotes instead of the angular brackets
      - **This indicates that the file is located in the current project directory**

- **Note:** You can also use this idea with procedural programs and put all your functions in a header file and reuse them in other programs

Class Design Considerations

- **Performing Input/Output in a Class Object (In general BAD DESIGN)**
  - An important class design issue is the use of **`cin`** and **`cout`** member functions
    - **In general it is considered good design to have class member functions AVOID using `cin` and `cout` (DO NOT PUT THEM IN CLASS MEMBER FUNCTIONS)**
      - This is so anyone who writes a program and uses this class will not be locked in to the way the class performs input and output
      - Unless a class is specifically designed to perform Input/Output operations like user input and output are best left to the person designing the application
    - Classes should provide member functions for retrieving data values without displaying them to the screen
    - Classes should provide member functions that store data into private member variables without using **`cin`**
    - That's another reason why we changed the **`setWidth`** and **`setLength`** methods in our revised rectangle class
  - **NOTE:** There are **some** instances where it is appropriate for a class to perform Input/Output.
    - One example is a class that is designed to display an menu on the screen and get the user's selection

Class Constructors

- ***Constructors***
  - A ***constructor*** is a special member function that is used to construct, or set up and initialize, an instance of a class data type
  - A ***constructor*** looks like a regular function except:
    1. Its name is the same as the class name
    2. It has no return type (remember with regular functions a return type must be specified even if that return type is void)
  - Constructor member functions are ***automatically called*** when an instance of a class (an object) is created
    - A constructor function is a special function automatically called to "construct" an object when it is instantiated
    - Example:
      - `Rectangle box;` //calls the constructor member function of the Rectangle class
        - What the programmer usually does is define the constructor in a way to initialize the member variables of the instance to default values…so after setting aside memory for box's length and width the constructor might run and set length and width to default values of zero
        - If we don't define our own constructor C++ defines a DO NOTHING default constructor
        - It's always better to define your own constructor
      - ***Example Rectangle constructor***:

```
Rectangle()
{
  length = 0;
  width = 0;
}
```

Important: **constructors are ONLY called (automatically when an instance is created) they can *NOT* be called explicitly**

This is *NOT* correct! `box.Rectangle()`

  - Constructor functions have the same name as the class
    - This is how the compiler knows which functions are constructors
  - ***A constructor does NOT have a return type not even*** `void`

Class Constructors

- ***Constructors - Format***
  - For the kinds of programs we are writing the constructors should be **`public`**
    - Make sure and place them in the public section of your class declaration since this will not be done automatically
      - When we used a constructor with a structure it was public by default, this is not the case with a class (with a class everything is private by default)
    - A constructor can be an inline function or its prototype can be defined in the class declaration and its definition in the implementation section
      - We are going to use the later method (placing the constructor prototype in the class declaration and its definition in the implementation section)

Class Constructors

- ***Constructors - Format***
  - When a constructor is defined in the implementation section it has the following format:
    - `className:: className(parameter list);`
      - The name appears twice because the function name for a constructor must be the same as the name of the class and the class name and the scope resolution operator must precede the function name.
      - Just like with a regular function if the constructor accepts no arguments its parameter list will be empty
  - Must have no return type (not even **`void`**)
    - This is because a constructor is always called when an object is instantiated and never returns anything to the object

Class Constructors

- Constructors are automatically implicitly called when an object is instantiated and their normal purpose is to initialize member data
  - Although the normal purpose of a constructor is to initialize member data or perform other setup of the object, it can to anything a normal function can do except return a value to the calling function
  - The Demo program has its constructor print a message instead of initializing data members
    - This message shows us when the constructor is being called in the program

- Example: (***Example 6***) The Demo program shows us when the constructor is being called in the program

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

/*CLASS DECLARATION - Normally would be in a .h file*/
//Since the class has no private members the private access specifier can be left out
class Demo
{
   public:
      Demo();
};

/*IMPLEMENTATION SECTION - Normallly would be in a .cpp implementation file*/

Demo::Demo()  //NOTICE NO RETURN TYPE IS SPECIFIED - because constructors never have a
       //return type
{
   cout << "Now the Demo Constructor is running\n"; //prints when constructor is executed
}

/*Client File – the program that uses the class,normally a separate .cpp file*/
int main()
{
   cout << "This message is displayed before the Demo object is created.\n";
   Demo demoObj;  //This defines the demo object and calls the constructor
   cout << "This message is displayed after the Demo object is created.\n";
   system("PAUSE");
   return 0;
}
```

- **Remember when a new object is created without passing any arguments to the constructor no parenthesis follow the object name:** Demo demoObj;
- **The constructor in this program accepts no arguments, a constructor that accepts no arguments is called a default constructor**

Class Constructors

- Adding a Constructor to the Rectangle Example
- Rectangle class declaration

```
//Rectangle Class Declaration
class Rectangle
{
   private:            //make the attributes private
     double length;
     double width;

   public:              //function prototypes
     bool setLength(double);
     bool setWidth(double);
     double getLength()const;
     double getWidth()const;
     double calcArea()const;
};
```

- IF NO constructor has been included as in the above example:
  - Compiler assigns a **do-nothing default constructor** equivalent to:

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

    - This constructor expects no parameters, and has an empty body
    - This is called the compiler supplied default constructor
      - Is not very useful but does exist if no other constructor is declared – ***in C++ is it more important than ever to create your own constructors? Why?***

Adding a Constructor to the Rectangle Class Declaration

FIGURE 12.4 *Constructor Format*

```
classname::className(parameter list)
{
    function body
}
```

Let's add a constructor to the rectangle class that allows the program using the class to initialize the length and width of the rectangle object when the Rectangle object is created.

This type of constructor is useful when each object a program may create of a type (like Rectangle) will most likely have different initialization values. This constructor allows the client application program to set the initialization values.

Adding a Constructor to the Rectangle Class Declaration

- Adding a Constructor to the Rectangle Example
- Rectangle class declaration
  - Example (***Example 7*** – Rectangle.h)

```
//Rectangle Class Declaration
class Rectangle
{
   private:              //make the attributes private
     double length;
     double width;

   public:                //function prototypes
     Rectangle(double, double);
     bool setLength(double);
     bool setWidth(double);
     double getLength()const;
     double getWidth()const;
     double calcArea()const;
};
```

Adding a Constructor to the Rectangle Class Declaration

- Adding a Constructor to the Rectangle Example
- Example (***Example 7***)
- Adding the Rectangle Constructor function definition to the class implementation file

```
/*RECTANGLE MEMBER FUNCTION DEFINITONS*/
/*****************************************************************
                        Rectangle::Rectangle(double, double)
This constructor takes the two arguments passed to it when an object is
instantiated and assigns those values to the length and width members.  If the
arguments passed in are negative, the constructor sets them to zero
*****************************************************************/
Rectangle::Rectangle(double len, double w)
{
  if(len >= 0)
    length = len;
  else
    length = 0;

  if(w >=0)
    width = w;
  else
    width = 0;
}
```

Adding a Constructor to the Rectangle Class Declaration

- Adding a Constructor to the Rectangle Example
- Using the constructor in the client program (in main)

```
#include <iostream>
#include <iomanip>
#include "Rectangle.h"
using namespace std;

/*************************************************************************
                              main
*************************************************************************/
int main()
{
   Rectangle box(4.1,6.0); //declare a rectangle object - call constructor that
                           //takes two doubles and set length to 4.1 and width to 6.0

   //call box's member functions to get the box information to display
   cout <<"\nHere is the Rectangle's data:\n";
   cout << fixed << setprecision(2);
   cout <<"Length: " << box.getLength() << endl;
   cout <<"Width:  " << box.getWidth() << endl;
   cout <<"Area:   " << box.calcArea() << endl;

   system("PAUSE");
   return 0;
}
```

Important: **You do *NOT* get rid of the set methods in the rectangle class. Why not? Is our Rectangle class ONLY for THIS client? What is a client program wants to change the length or width of the rectangle later? This is an easy way to lose A LOT of points on the final**

Adding a Constructor to the Rectangle Class

- WHAT HAPPENS IF WE CALL THE CONSTRUCTOR WITH NO ARGUMENTS AFTER WE HAVE CREATED THIS NEW CONSTRUCTOR THAT ACCEPTS TWO ARGUMENTS?

```
#include <iostream>
#include <iomanip>
#include "Rectangle.h"
using namespace std;

/*****************************************************************************
                              main
*****************************************************************************/
int main()
{
   Rectangle box(4.1,6.0); //declare a rectangle object box - call constructor that
                           //takes two doubles and set length to 4.1 and width to 6.0

   Rectangle box2;  //What constructor is called?

   //call box's member functions to get the box information to display
   cout <<"\nHere is the Rectangle's data:\n";
   cout << fixed << setprecision(2);
   cout <<"Length: " << box.getLength() << endl;
   cout <<"Width:  " << box.getWidth() << endl;
   cout <<"Area:   " << box.calcArea() << endl;

  //call box2's member functions to get the box2 information to display - What prints?
   cout <<"\nHere is the Rectangle's data:\n";
   cout << fixed << setprecision(2);
   cout <<"Length: " << box2.getLength() << endl;
   cout <<"Width:  " << box2.getWidth() << endl;
   cout <<"Area:   " << box2.calcArea() << endl;

   system("PAUSE");
   return 0;
}
```

ANSWER: COMPILER ERROR: rectangleApp.cpp no matching function for call to `Rectangle::Rectangle()

Adding an Overloaded Constructor to the Rectangle Class

- Let's add a no-arg constructor (a no arg constructor is any constructor that takes no arguments, a default constructor is a no arg constructor provided by C++) to the rectangle class so if another program instantiates a rectangle class without arguments for length and width the default constructor is called and default values are assigned to length and width
- Example (Rectangle – ***Example 8***)
- Revised class declaration:

```
#ifndef RECTANGLE_H
#define RECTANGLE_H

//Rectangle Class Declaration
class Rectangle
{
   private:                                  //make the attributes private
     double length;
     double width;

   public:                            //function prototypes
     Rectangle();                    //default constructor
     Rectangle(double, double); //constructor that takes two parameters
     bool setLength(double);
     bool setWidth(double);
     double getLength() const;
     double getWidth() const;
     double calcArea() const;
};

#endif
```

Adding an Overloaded Constructor to the Rectangle Class

- Revised Constructor definitions in the class implementation file:

```
/*RECTANGLE MEMBER FUNCTION DEFINITONS*/
/*****************************************************************************
                        Rectangle::Rectangle()
This constructor takes zero arguments.  It sets length and width to 1.0
*****************************************************************************/
Rectangle::Rectangle()
{
  length = 1.0;
  width = 1.0;
}

/*****************************************************************************
                        Rectangle::Rectangle(double, double)
This constructor takes the two arguments passed to it when an object is instantiated
and assigns those values to the length and width members.  If the arguments passed
in are negative, the constructor sets them to zero
*****************************************************************************/
Rectangle::Rectangle(double len, double w)
{
  if(len >= 0)
    length = len;
  else
    length = 0;

  if(w >=0)
    width = w;
  else
    width = 0;
}
```

Adding an Overloaded Constructor to the Rectangle Class

- Revised Client Program (Program that uses the class)

```
/*************************************************************************
                              main
*************************************************************************/
int main()
{
   Rectangle box(4.1,6.0); //declare a rectangle object - call constructor that
                           //takes two doubles and set length to 4.1 and width to 6.0

   Rectangle box2;  //The default Constructor is now called


   //call box's member functions to get the box information to display
   cout <<"\nHere is the Rectangle BOX'S data:\n";
   cout << fixed << setprecision(2);
   cout <<"Length: " << box.getLength() << endl;
   cout <<"Width:  " << box.getWidth() << endl;
   cout <<"Area:   " << box.calcArea() << endl;

   //call box2's member functions to get the box2 information to display - What prints?
   cout <<"\nHere is the Rectangle BOX2'S data:\n";
   cout << fixed << setprecision(2);
   cout <<"Length: " << box2.getLength() << endl;
   cout <<"Width:  " << box2.getWidth() << endl;
   cout <<"Area:   " << box2.calcArea() << endl;

   return 0;
}
```

Important Hint on HW #7 Part 2

- The getAve function should USE the getTotal function by calling it, remember member functions can call and use each other

```
double Stats::getAve()
{
    double tot = getTotal()
    //rest of code here
}
```

Destructors

- **Concept:** A ***destructor*** is a member function that is automatically called when an object is destroyed

- A destructor is a counterpart of the constructor function
  - Has same name as the class
    - Preceded with a tilde (**~**)
  - **`Rectangle`** class destructor: **`~Rectangle()`**

- Destructors are automatically called when an object is destroyed.
  - In the same way that constructors can be used to set things up when an object is created destructors are used to perform shutdown type procedures when an object goes out of existence

- More points on Destructors:
  - Destructors have no return type
  - Destructors ***cannot accept arguments so they never have a parameter list*** (the list is always empty **`~Rectangle();`**
  - Because Destructors can accept no arguments there can only be one Destructor

- Destructors are most useful when working with objects that have dynamically allocated members
  - We'll see an example of this in the pointers lecture

Arrays of Objects

- Declaring array of objects same as declaring array of any C++ built-in type
  - **Example:** `Circ theCirc[4];`
    - Creates three objects named `theCirc[0]` through `theCirc[3]`
    - This example calls the DEFAULT constructor for each Circle element of the array
      - That means for this example we better have a default constructor

- Member functions for `theCirc` array objects are called using:

```
objectName.functionName()
```

- **Example: Declaring an array of Circle objects using our Circle class**
  - **(Example 9)**

```
#include <iostream>
#include "circle.h"
using namespace std;

const int numCircles = 4;  //for our circle array subscript

int main()
{
    Circle theCirc[numCircles];  //declare an array of 4 Circle Objects, called theCirc

    //use a loop to initialize the radius of each object
    for(int index = 0; index < numCircles; index++)
    {
       double r;
       cout << "Enter the radius for circle #" << index << ":";
       cin >> r;
       theCirc[index].setRadius(r);  //sets the current index's radius
                                     //remember the index is the array subscript
    }

    cout << endl;

    //now print out the radius of each circle in the array using a loop
    for(int index = 0; index < numCircles; index++)
    {
       cout << "Circle #" << index << " has a radius of "
            << theCirc[index].getRadius() << endl;
    }
   system("PAUSE");
   return 0;
}
```

Passing Objects as Function Arguments

- **Passing an entire object as a Function Argument**

```
showRect(box);

void showRect(Rectangle r)
{
   cout << r.getLength() << endl;
   cout << r.getWidth() << endl;
   cout << r.getArea() << endl;
}
```

  - Once the function is called **`r's`** member variable **`length`** contains a **copy** of **`box's`** member variable **`length`**, and **`r.width`** contains a copy of **`box's`** member variable **`width`**
  - By default class instances (objects) are passed to functions ***by value*** as shown here – THIS IS DIFFERENT THAN JAVA!!!

- See ***Example 15***

Passing Objects as Function Arguments– Pass by Reference

- ***Pass by value*** has an advantage for a object if the function does not need to have access (be able to modify) to the members of the original object argument (you always want to protect your data and make functions reusable and independent)
  - ***NOTE:*** anything I do to the `r Rectangle` inside `showRect` does not happen to `box` (let's try it)
  - If a function needs to have access to the original object the object should be ***passed by reference***
  - Remember passing by reference passes the address of the argument (in this case a object) giving the function access to the actual object (instance)
  - ***HOWEVER think of all the space wasted copying a object with a large number of members in it***
- ***Example 16***
  - ***Now I'll pass the rectangle to the function by reference so any changes that happen to the rectangle in the function also happens to the argument (rectangle passed into the function)***

Constant Reference Parameters

- ***Disadvantages of Passing an Object by Value***
  - If a object has a lot of members passing an object by value can slow down a program's execution time
    - This is because when an argument is passed by value a copy of it is created
      - For an object argument passed by value each of its member variables must passed so a copy of each member variable must be created

- ***When a argument is Passed by Reference a copy of it is not created***
  - Only a reference that points to the original argument is passed
  - The ***disadvantage of pass by reference*** however is that the function has access to the original argument (all of a structure's members in this case) and can alter the argument's value (all member values)
    - This violates the principle of least privilege

- **Solution: Pass the argument by a *constant reference***

Constant Reference Parameters

- **Solution:** Pass the argument by a ***constant reference***
  - When an argument is passed by ***constant reference*** **this means that only a reference to the original variable is passed to the function *BUT* the argument cannot be changed by the function – *so we minimize memory use and at the same time protect our data as well as keep our function more reusable and independent***
  - To declare that a function parameter will receive a constant reference the keyword **const** must be placed in the parameter list of the function prototype and the function header
  - Revised **`showRect function`**

Whenever you make an parameter of a function `const`, the function cannot change that argument

```
void showRect(const Rectangle&)  //function prototype
void showRect(const Rectangle& r ) //function header
```

- ***Example 17***

When possible I usually pass objects by constant reference in C++ to save memory and keep my functions independent and reusable as well as protect data