Foundation of Computer Science: Class

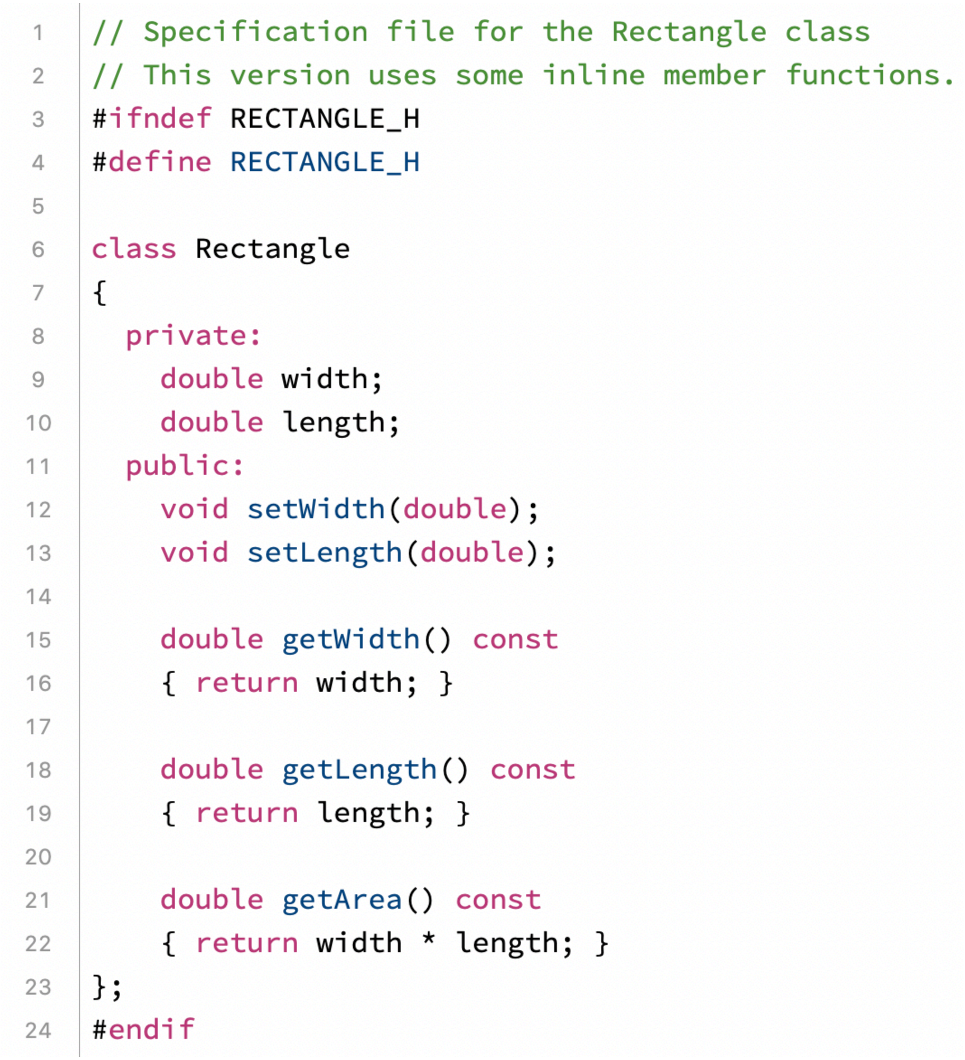
Kafi Rahman

Assistant Professor

Computer Science

Truman State University

Rectangle Class with Inline Member Functions



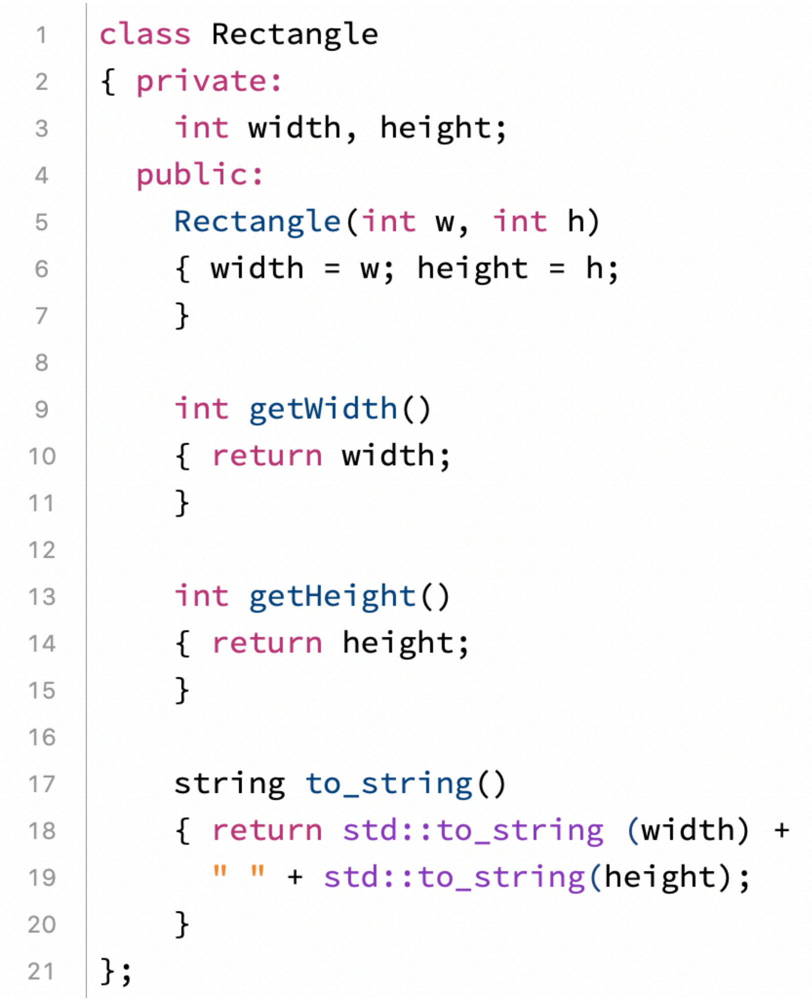
```cpp
// Specification file for the Rectangle class
// This version uses some inline member functions.
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle
{
  private:
    double width;
    double length;
  public:
    void setWidth(double);
    void setLength(double);

    double getWidth() const
    { return width; }

    double getLength() const
    { return length; }

    double getArea() const
    { return width * length; }
};
#endif
```

How to Compare Two Objects

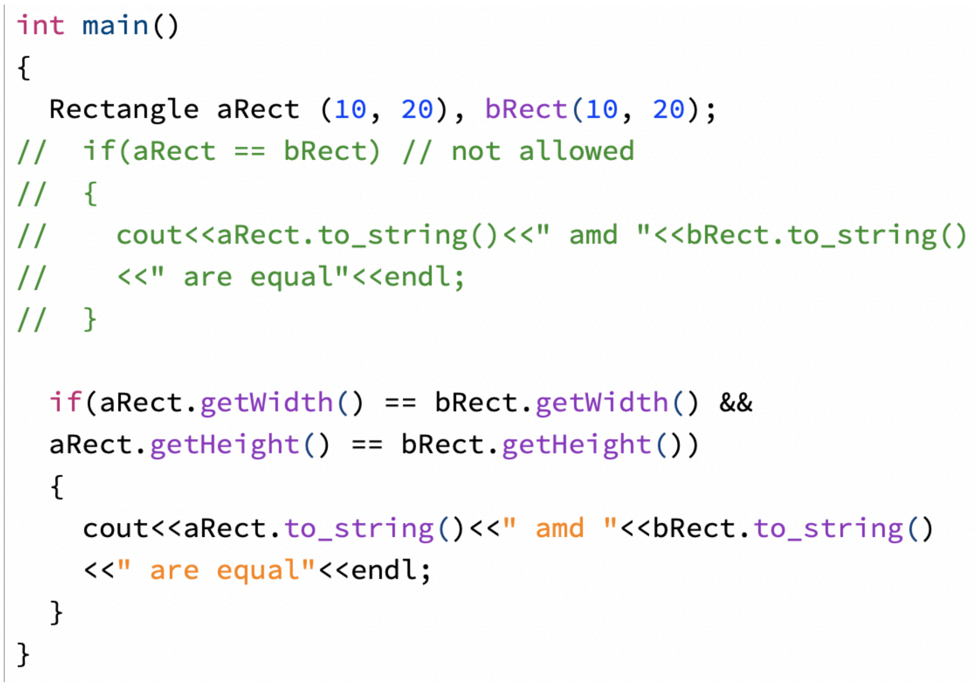
- Given two objects of the Rectangle class, how would we compare them to determine whether they are equal?

```
Rectangle aRect(10, 20);
Rectangle bRect(10, 20);

if(aRect == bRect) // is this supported?
{
  cout<<"Equal"<<endl;
}
```

How to Compare Two Objects of a Class



```
class Rectangle
{ private:
    int width, height;
  public:
    Rectangle(int w, int h)
    { width = w; height = h;
    }

    int getWidth()
    { return width;
    }

    int getHeight()
    { return height;
    }

    string to_string()
    { return std::to_string (width) +
       " " + std::to_string(height);
    }
};
```



```
int main()
{
  Rectangle aRect (10, 20), bRect(10, 20);
//  if(aRect == bRect) // not allowed
//  {
//    cout<<aRect.to_string()<<" amd "<<bRect.to_string()
//    <<" are equal"<<endl;
//  }

  if(aRect.getWidth() == bRect.getWidth() &&
  aRect.getHeight() == bRect.getHeight())
  {
    cout<<aRect.to_string()<<" amd "<<bRect.to_string()
    <<" are equal"<<endl;
  }
}
```

13.15

The Unified Modeling Language

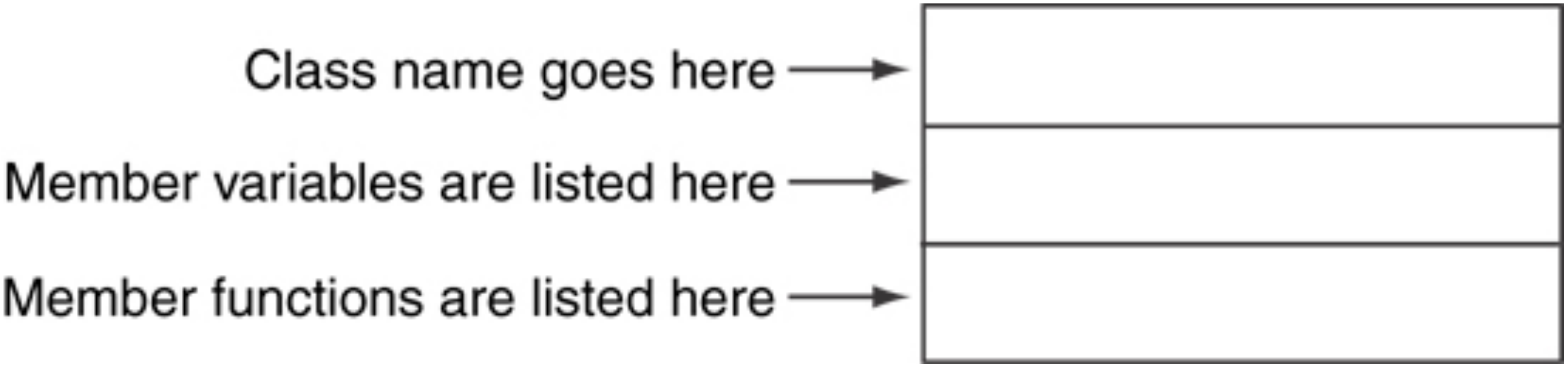
The Unified Modeling Language

- UML stands for Unified Modeling Language.

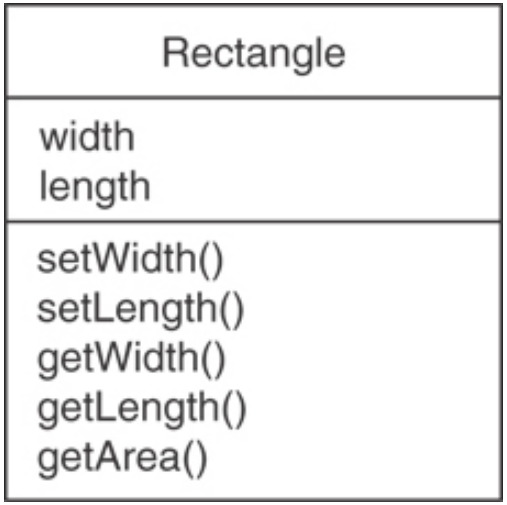
- The UML provides a set of standard diagrams for graphically depicting object-oriented systems

UML Class Diagram

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


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

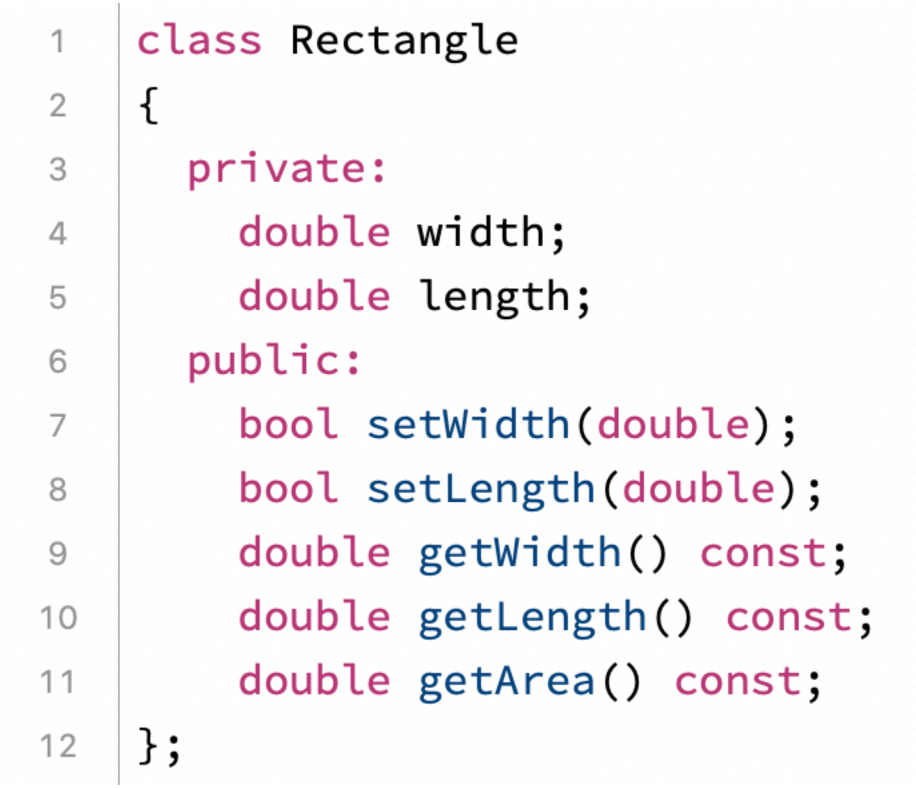
Example: A Rectangle Class



Rectangle

width
length

setWidth()
setLength()
getWidth()
getLength()
getArea()



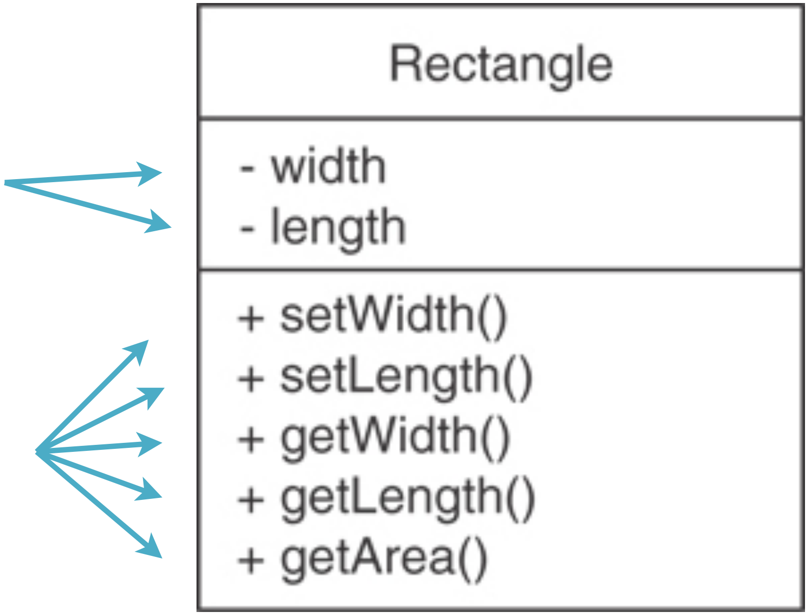
```cpp
class Rectangle
{
  private:
    double width;
    double length;
  public:
    bool setWidth(double);
    bool setLength(double);
    double getWidth() const;
    double getLength() const;
    double getArea() const;
};
```

UML Access Specification Notation

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

These member variables are private.

These member functions are public.



Rectangle

- width
- length

+ setWidth()
+ setLength()
+ getWidth()
+ getLength()
+ getArea()

UML Data Type Notation

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

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

UML Parameter Type Notation

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

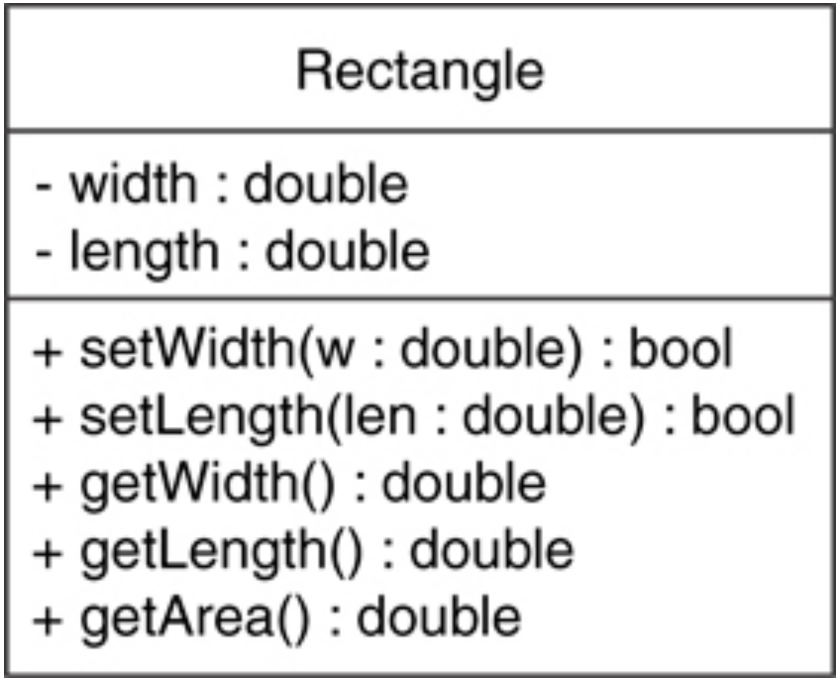
```
+ setWidth(w : double)
```

UML Function Return Type Notation

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

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

The Rectangle Class



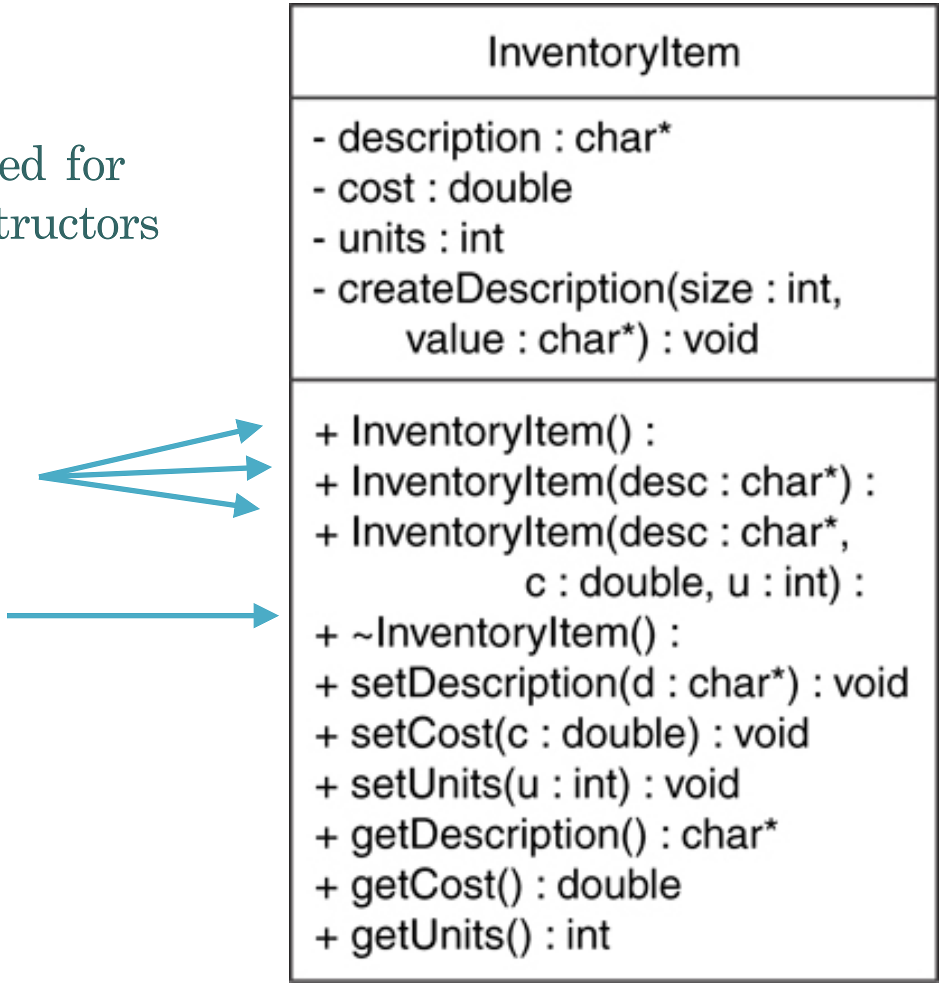
Rectangle

- width : double
- length : double

+ setWidth(w : double) : bool
+ setLength(len : double) : bool
+ getWidth() : double
+ getLength() : double
+ getArea() : double

Showing Constructors and Destructors

No return type listed for constructors or destructors



Constructors

Destructor

InventoryItem

- - description : char*
- - cost : double
- - units : int
- - createDescription(size : int, value : char*) : void

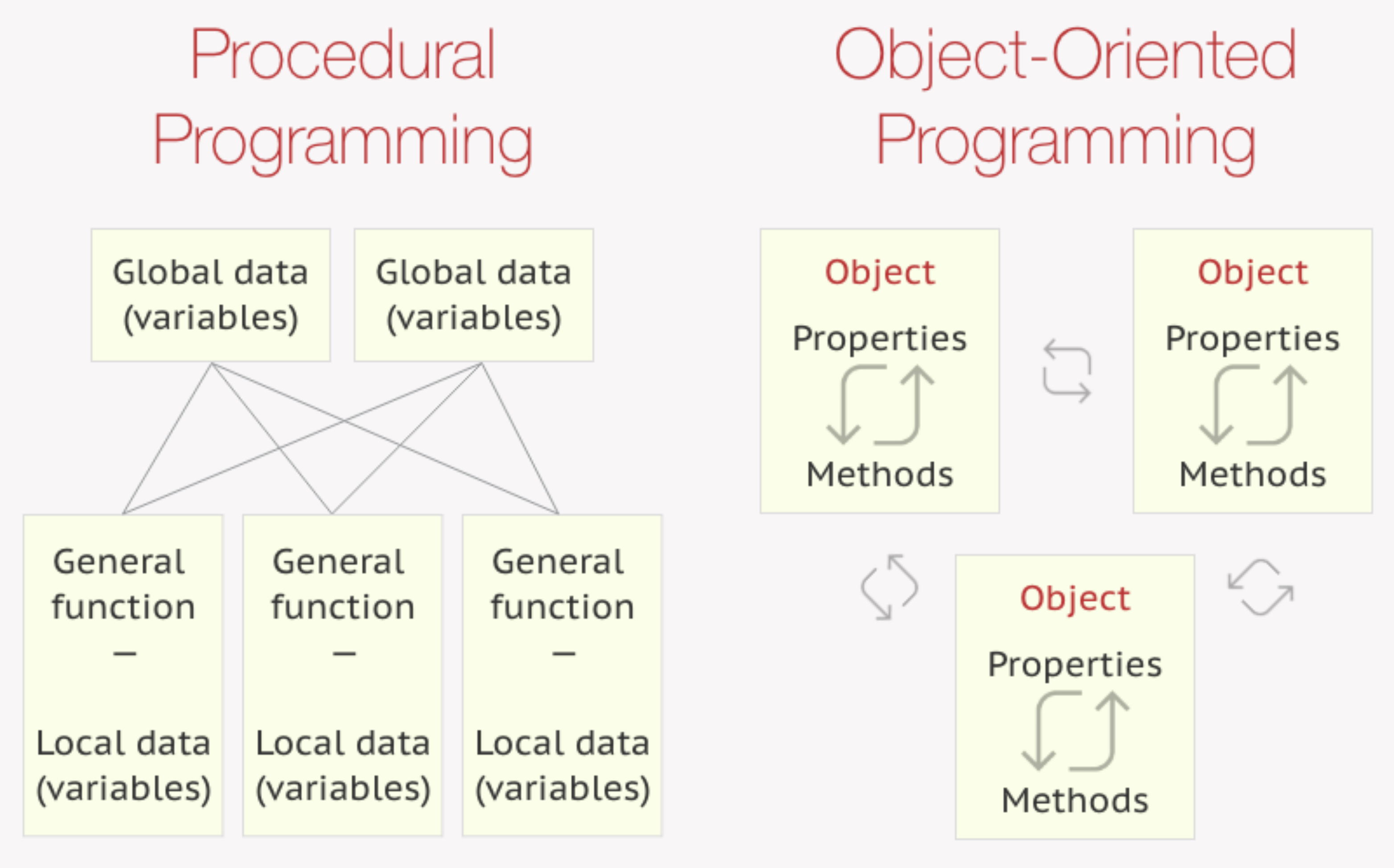
- + InventoryItem() :
- + InventoryItem(desc : char*) :
- + InventoryItem(desc : char*, c : double, u : int) :
- + ~InventoryItem() :
- + setDescription(d : char*) : void
- + setCost(c : double) : void
- + setUnits(u : int) : void
- + getDescription() : char*
- + getCost() : double
- + getUnits() : int

Procedural and Object-Oriented Programming

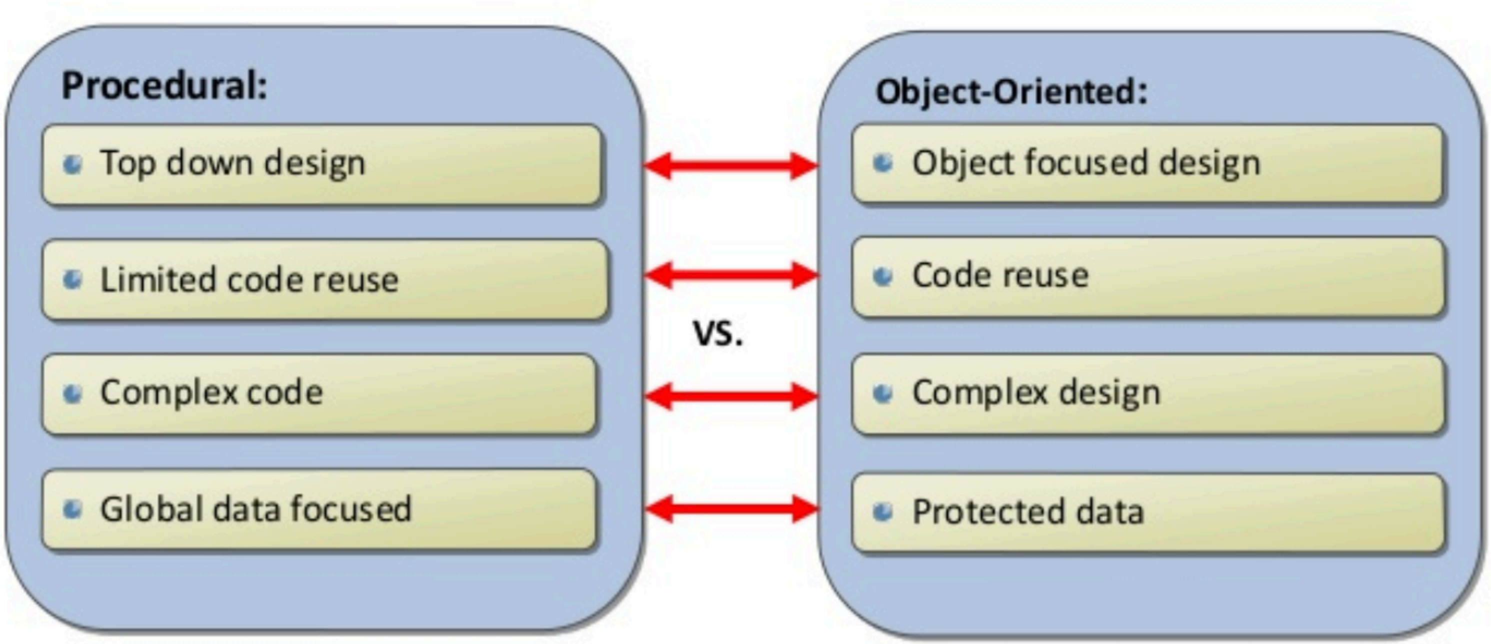
- Procedural Oriented programming (POP) focuses on the process/actions that occur in a program
  - They do not have proper way of hiding data, hence, they are less secure

- Object-Oriented programming (OOP) is based on the data and the functions that operate on those data.
  - Objects are instances of ADTs that represent the data and its functions
  - OOP provides data hiding mechanism

Procedural and Object-Oriented Programming



Procedural and Object-Oriented Programming



Procedural:

- Top down design
- Limited code reuse
- Complex code
- Global data focused

vs.

Object-Oriented:

- Object focused design
- Code reuse
- Complex design
- Protected data

Checkpoint Exercises

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

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

  - A) RetailItem::void setPrice(double p)
  - B) void RetailItem::setPrice(double p)

Checkpoint Exercises (cont)

- 13.3 An object's private member variables are accessed from outside the object by
  - A) public member functions
  - B) any function
  - C) the dot operator
  - D) the scope resolution operator

- 13.4 Assume that RetailItem is the name of a class, and the class has a void member function named setPrice, which accepts a double argument. If soap is an instance of the RetailItem class, which of the following statements properly uses the soap object to call the setPrice member function?
  - A) RetailItem::setPrice(1.49);
  - B) soap::setPrice(1.49);
  - C) soap.setPrice(1.49);
  - D) soap:setPrice(1.49);

Readings from Chapter 13

- Read the following sections
  - 13.1, 13.2, 13.3, 13.4, 13.5, 13.6, 13.7, 13.8, 13.9, 13.10, 13.11, 13.12
  - Skip 13.13, 13.14
  - Good to know 13.15, 13.16

- Checkpoint exercises
  - 13.1, 13.2, 13.3, 13.4, 13.5
  - 13.6, 13.7, 13.8, 13.9, 13.11
  - 13.12 -- 13.20 (all of them)
  - Skip 13.27 -- 13.33 (all of them)

Readings from Chapter 13

- Review Questions
  - Short Answer
    - 1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 13, 14
  - Fill in the Blank
    - Try all of them
  - Algorithm Workbench
    - 43, 44, 45, 46, 47, 48
  - True or False
    - Try all of them
  - Find the Errors
    - Try all of them (they are fun!!)
  - Programming Challenges
    - 1, 2, 3, 5, 6, 8, 10 (very interesting), 14

Chapter 14:

More About Classes

14.1

Instance and Static Members

Instance and Static Members

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

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

- `static` member function: can be used to access `static` member variable;
  - static member functions can be called just by using the class name!

static member variable

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

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

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

Static member declared here.

Static member defined here.

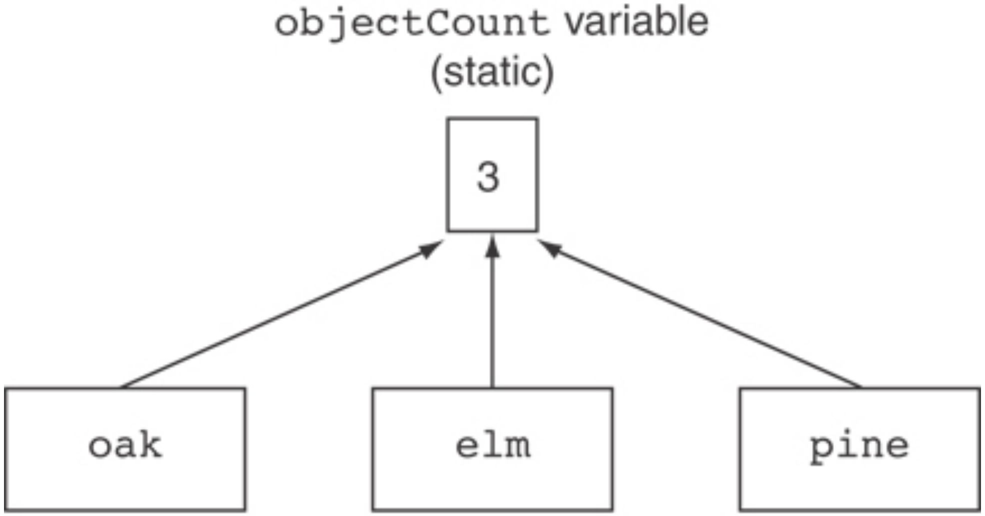
Program 14-1

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

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

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

Program Output

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

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



Instances of the `Tree` class

`static` member function

- Declared with `static` before return type:

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

- Static member functions can only access static member data

- Can be called independent of objects:

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

Modified Version of Tree.h

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

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

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

```
// we can call the static function
// by using the class name
cout << "There are " << Tree::getObjectCount()
     << " objects.\n";
```