# Intro. Classes

*Beginning Objected Oriented Programming*

**CIS 15 : Spring 2007**

# Functionalia

HW 5 Out.

Due After Midterm 2.

Today:

- More Classes
- HW 5 Concepts

# Introduction to Classes and OOP

Classes are the basis for *Object Oriented Programming* in C++.
So far, we've been doing *Procedural Programming* in C++.

*Object Oriented Programming* provides better organizational structures for your code -

*classes = data + functionality.*

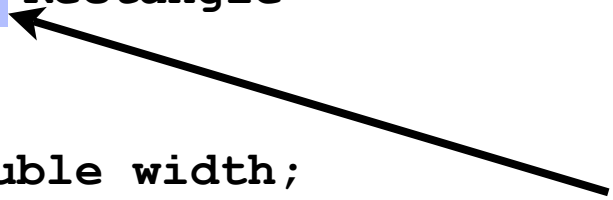
Tenants of **O**bject **O**riented **P**rogramming

1. Encapsulation
2. Data Hiding
3. Polymorphism
4. Inheritance

# Definition of a Class

A ***class*** is defined in a similar manner as a ***struct*** is.

```
class Rectangle
{
   double width;
   double height;
};
```

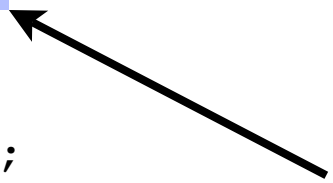


Keyword `class` to indicate that what follows is a Class.

# Definition of a Class

A ***class*** is defined in a similar manner as a ***struct*** is.

```
class Rectangle
{
   double width;
   double height;
};
```

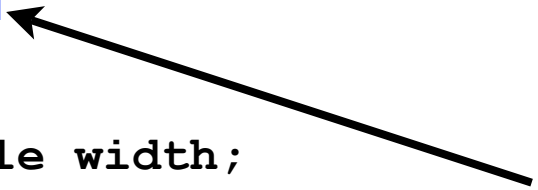


The Name (Tag) of the class follows. This becomes the "*type*" that you declare instances of the class Rectangle with.

# Definition of a Class

A ***class*** is defined in a similar manner as a ***struct*** is.

```
class Rectangle
{
   double width;
   double height;
};
```

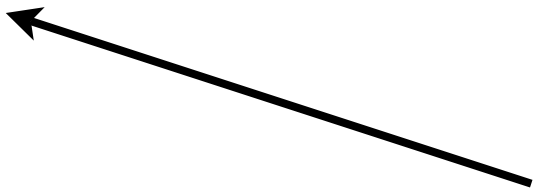


Again, customary to capitalize the first letter of the name of the class.

# Definition of a Class

A ***class*** is defined in a similar manner as a ***struct*** is.

```
class Rectangle
{
   double width;
   double height;
};
```



Don't forget the semi-colon!

# Definition of a Class

A ***class*** is defined in a similar manner as a ***struct*** is.

```
class Rectangle
{
    double width;
    double height;
};
```

Declarations of your data (and functions - as you will shortly see), contained within the curly-braces.

*NOTE: Unlike structs - by default the data and the functions of a class are* ***Private****.*

# Definition of a Class

A ***class*** is defined in a similar manner as a ***struct*** is.

```
class Rectangle
{
    double width;
    double height;
};
```

Declarations of your data (and functions - as you will shortly see), contained within the curly-braces.

*NOTE: Unlike structs - by default the data and the functions of a class are **Private**.*

```
Rectange r1;
r1.width = 5.555;
```

COMPILE ERROR!

# Access Specifiers

Use **`public`**: and **`private`**: labels to specify the access to the different data and functions.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      double getArea();
};
```

Even through it is by default private - it is a good idea to explicitly label the private components of the class.

# Access Specifiers

Set up public member *accessor functions* to control access to the private members of the class.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      void setWidth(double w);
      void setHeight(double h);
      double getArea();
};
```

The interface to manipulating the object.

# Access Specifiers

Set up public member *accessor functions* to control access to the private members of the class.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      void setWidth(double w);
      void setHeight(double h);
      double getArea();
};
```

Called *mutators*, or *setter* functions.

Called *accessor*, or *getter* functions.

# Read-Only Member Functions

The keyword `const` can be used to specify that the function will not change any of the data stored in the class.

This is helpful to guarantee that there is no future code inadvertently overwrites something.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};
```

# Defining Member Functions

Once declared in the class, member functions are defined outside of the class definition. Note the use of the '**::**' ***scope operator***.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};

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

void Rectangle::setHeight(double h) {
   height = h;
}
```

***scope operator***
(used with the class name)

# Defining Member Functions

Once declared in the class, member functions are defined outside of the class definition. Note the use of the '**::**' ***scope operator***.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};

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

void Rectangle::setHeight(double h) {
   height = h;
}
```

Private member variables are accessible from the function definition.

# Defining Member Functions

Once declared in the class, member functions are defined outside of the class definition. Note the use of the '**::**' ***scope operator***.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};

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

Use of the **`const`** operator in the function definition to specify a "*constant*" object and read-only access to the class data.

# Defining Member Functions

Once declared in the class, member functions are defined outside of the class definition. Note the use of the '**::**' ***scope operator***.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};

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

Avoids *stale* data, defining Area as a resulting operation on the two member variables **`width`** and **`height`** - as opposed to having declared an **`area`** member variable.

# Defining Member Functions

Once declared in the class, member functions are defined outside of the class definition. Note the use of the '**::**' ***scope operator***.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};
```

Notice that the return type comes on the far right-side of the **`Rectangle::getArea()`** declaration.

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

# Instantiation of the Class

Using the class-name to declare a variable of the class type is called *instantiating* the class. The variable is called an ***object***.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};
```

Definition outside of `main()` and other functions.

---

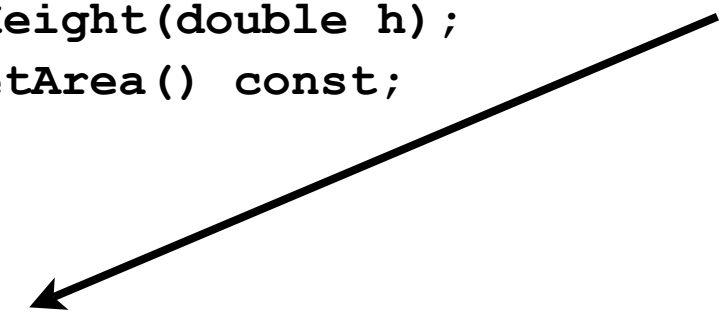
```
Rectangle r1;
r1.setWidth(1.0);
r1.setHeight(4.0);
cout << r1.getArea() << endl;
```

Use of the Class occurs in main() or other functions.

# Instantiation of the Class

Using the class-name to declare a variable of the class type is called *instantiating* the class. The variable is called an ***object***.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};
```



Object `r1` is an instance of the Rectangle class.

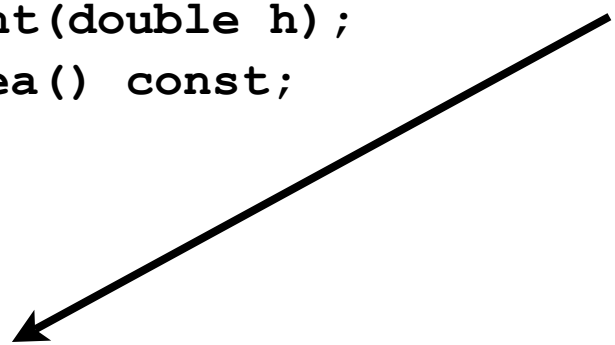
```
Rectangle r1;
r1.setWidth(1.0);
r1.setHeight(4.0);
cout << r1.getArea() << endl;
```

# Instantiation of the Class

Using the class-name to declare a variable of the class type is called *instantiating* the class. The variable is called an ***object***.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};
```

Access the member functions (methods) via ***dot-notation***.



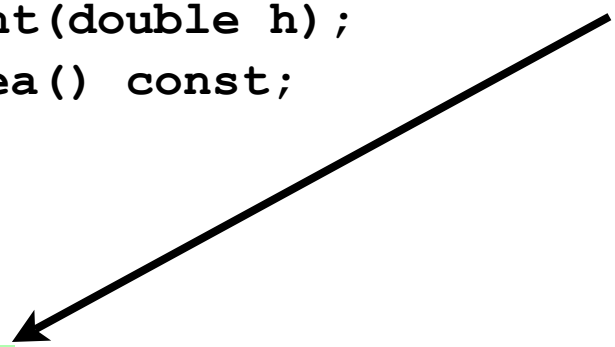
```
Rectangle r1;
r1.setWidth(1.0);
r1.setHeight(4.0);
cout << r1.getArea() << endl;
```

# Instantiation of the Class

Using the class-name to declare a variable of the class type is called *instantiating* the class. The variable is called an ***object***.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};
```

Access the member functions (methods) via ***dot-notation***.



```
Rectangle r1;
r1.setWidth(1.0);
r1.setHeight(4.0);
cout << r1.getArea() << endl;
```

Expected output?

# Instantiation of the Class

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};
```

```
Rectangle r1;
r1.setWidth(1.0);
r1.setHeight(4.0);
cout << r1.getArea() << endl;
```

4.0

# Instantiation of the Class

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};
```

```
Rectangle r2;
cout << r2.getArea() << endl;
```

Expected output?

# Instantiation of the Class

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};
```

```
r2.width = ?
r2.height = ?
```

```
Rectangle r2;
cout << r2.getArea() << endl;
```

# Pointer to a Class

Like other variables, and structs, an object can have a ***pointer*** that references it indirectly.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};


Rectangle r3;
Rectangle * rPtr.

rPtr = &r3;
rPtr->setWidth(0.2);
```

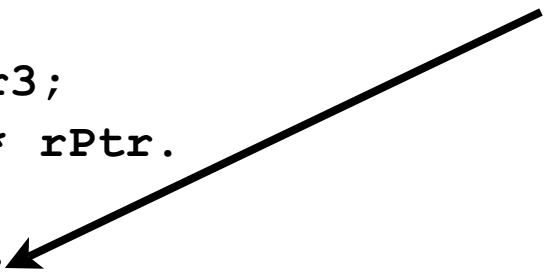
# Pointer to a Class

Like other variables, and structs, an object can have a ***pointer*** that references it indirectly.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};
```

```
Rectangle r3;
Rectangle * rPtr.

rPtr = &r3;
rPtr->setWidth(0.2);
```



Use of the & operator to set the pointer to hold the memory address of the object.

# Pointer to a Class

Like other variables, and structs, an object can have a ***pointer*** that references it indirectly.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};
```

```
Rectangle r3;
Rectangle * rPtr.

rPtr = &r3;
rPtr->setWidth(0.2);
```

Use of the -> reference operator to access the public members of the class.

# Dynamically Allocated Objects

Objects (like structs) can be dynamically allocated as well.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};
```

```
Rectangle * rPtr.
rPtr = new Rectangle;
```

**new** operator allows for the creation of a dynamically allocated Rectangle object.

```
rPtr->setWidth(0.2);
```

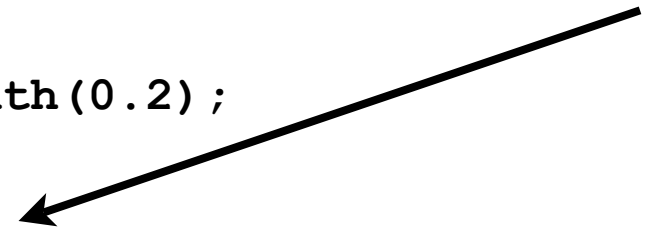
# Dynamically Allocated Objects

What gets dynamically allocated must eventually be freed.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};
```

```
Rectangle * rPtr.
rPtr = new Rectangle;
```

```
rPtr->setWidth(0.2);
...
delete rPtr;
```



`delete` operator frees the memory of the dynamically allocated Rectangle object.

# Private Data

Having the data hidden and private allows for more intelligent and safer handling of the object. Additionally it allows for better code evolution.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};

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

Keep on getting errors!
Negative widths.

Need to update the `setWidth(...)` function.

# Private Data

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};

void Rectangle::setWidth(double w) {
   if(w >= 0)
      width = w;
   else {
      cout << "Width is invalid!" << endl;
      exit(EXIT_FAILURE); // quits the program
   }
}
```

Does the rest of the program need to change?

# Private Data

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};

void Rectangle::setWidth(double w) {
   if(w >= 0)
      width = w;
   else {
      cout << "Width is invalid!" << endl;
      width = 0; // error handling
   }
}
```

Less heavy handed.
But needs to be documented!

# Inline Member Functions

Very small and simple functions can be declared ***inline*** to improve code clarity and possible performance enhancements.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      void setWidth(double w) {
         if(w >= 0)
            width = w;
      }
      void setHeight(double h) {
         if(h >= 0)
            height = h;
      }
      double getArea() const;
};
```

# Inline Member Functions

The actual call of the function gets replaced with a copy inline code. Instead of the normal use of the call stack and jump in the program code.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      void setWidth(double w) {
         if(w >= 0)
            width = w;
      }
      void setHeight(double h) {
         if(h >= 0)
            height = h;
      }
      double getArea() const;
};
```

# Inline Member Functions

The actual call of the function gets replaced with a copy inline code. Instead of the normal use of the call stack and jump in the program code.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      void setWidth(double w) {
         if(w >= 0)
            width = w;
      }
      void setHeight(double h) {
         if(h >= 0)
            height = h;
      }
      double getArea() const;
};
```

Inline functions are a *request* to the compiler.

*Code size grows considerably with many inline function calls.*

# Constructors

A ***constructor*** is a special member function that gets called when the class is instantiated.

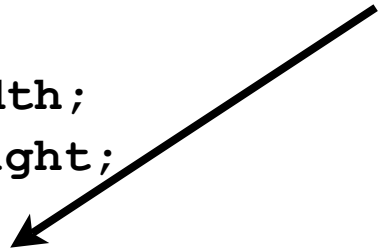
```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      Rectangle();
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};
```

Name of the constructor is the same as a the class name.

# Constructors

A ***constructor*** is a special member function that gets called when the class is instantiated.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      Rectangle();
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};
```



Note the absence of a return type!

# Constructors

A ***constructor*** is a special member function that gets called when the class is instantiated.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      Rectangle();
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};

Rectangle::Rectangle() {
   width = 0;
   height = 0;
}
```

Defined like other member functions.

Common use: initialization of object data.

# Constructors

A ***constructor*** is a special member function that gets called when the class is instantiated.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      Rectangle();
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};

Rectangle::Rectangle() {
   width = 0;
   height = 0;
}
```

```
Rectangle r1;

cout << r1.getArea() << endl;
```

What is printed?

# Constructors

A ***constructor*** is a special member function that gets called when the class is instantiated.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      Rectangle();
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};

Rectangle::Rectangle() {
   width = 0;
   height = 0;
}
```

```
Rectangle r1;

cout << r1.getArea() << endl;
```

0

# Constructors

A constructor is called also when an object is dynamically allocated.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      Rectangle();
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};

Rectangle::Rectangle() {
   width = 0;
   height = 0;
}
```

```
Rectangle * rPtr;
rPtr = new Rectangle;

cout << rPtr->getArea() << endl;
```

# Constructors

Constructors also can have function parameters (useful in initializing member data).

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      Rectangle(double w, double h);
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};

Rectangle::Rectangle(double w, double h) {
   width = h;
   height = h;
}
```

# Constructors

Use of parameters in instantiating the object looks similar to a function call.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      Rectangle(double w, double h);
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};

Rectangle::Rectangle(double w, double h) {
   width = h;
   height = h;
}
```

```
Rectangle r1(0.1, 10.0);

cout << r1.getArea() << endl;
```

What is printed?

# Constructors

Use of parameters in instantiating the object looks similar to a function call.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      Rectangle(double w, double h);
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};

Rectangle::Rectangle(double w, double h) {
   width = h;
   height = h;
}
```

```
Rectangle r1(0.1, 10.0);

cout << r1.getArea() << endl;
```

1.0

# Constructors

Default values can be set in the definition of the Constructor to create a ***default constructor***.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      Rectangle(double w, double h);
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};

Rectangle::Rectangle(double w = 1.0, double h = 1.0) {
   width = h;
   height = h;
}
```

# Constructors

Default values can be set in the definition of the Constructor to create a ***default constructor***.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      Rectangle(double w, double h);
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};

Rectangle::Rectangle(double w = 1.0, double h = 1.0) {
   width = h;
   height = h;
}
```

```
Rectangle r1;

cout << r1.getArea() << endl;
```

What is printed?

# Constructors

Default values can be set in the definition of the Constructor to create a ***default constructor***.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      Rectangle(double w, double h);
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};

Rectangle::Rectangle(double w = 1.0, double h = 1.0) {
   width = h;
   height = h;
}
```

```
Rectangle r1;

cout << r1.getArea() << endl;
```

1.0

# Passing parameters in Dynamic Objects

Dynamically Allocated Objects can also have parameters passed to the Constructor.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      Rectangle(double w, double h);
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};

Rectangle::Rectangle(double w = 1.0, double h = 1.0) {
   width = h;
   height = h;
}
```

```
Rectangle * rPtr;
rPtr = new Rectangle(2.0, 1.0);

cout << rPtr->getArea() << endl;
```

What is printed?

# Passing parameters in Dynamic Objects

Dynamically Allocated Objects can also have parameters passed to the Constructor.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      Rectangle(double w, double h);
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};

Rectangle::Rectangle(double w = 1.0, double h = 1.0) {
   width = h;
   height = h;
}
```

```
Rectangle * rPtr;
rPtr = new Rectangle(2.0, 1.0);

cout << rPtr->getArea() << endl;
```

2.0

# Destructors

***Destructors*** are member functions that are called when the object is de-allocated.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      Rectangle(double w, double h);
      ~Rectangle();
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};
```

Same as a constructor except declaration uses a ~ to indicate that it is a Destructor.

# Destructors

***Destructors*** are member functions that are called when the object is de-allocated.

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      Rectangle(double w, double h);
      ~Rectangle();
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};

Rectangle::~Rectangle() {
   cout << "Bye bye!" << endl;
}
```

Definition is the same. Note no parameters are ever used.

# Destructors

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      Rectangle(double w, double h);
      ~Rectangle();
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};

Rectangle::~Rectangle() {
   cout << "Bye bye!" << endl;
}

int main() {
   Rectangle r1(0.1, 10.0);

   cout << r1.getArea() << endl;
}
```

What is printed?

Destructor is called at the end of `main()`.

# Destructors

```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      Rectangle(double w, double h);
      ~Rectangle();
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};

Rectangle::~Rectangle() {
   cout << "Bye bye!" << endl;
}

int main() {
   Rectangle r1(0.1, 10.0);

   cout << r1.getArea() << endl;
}
```

1.0
Bye bye!

Destructor is called at the end of `main()`.

# Destructors

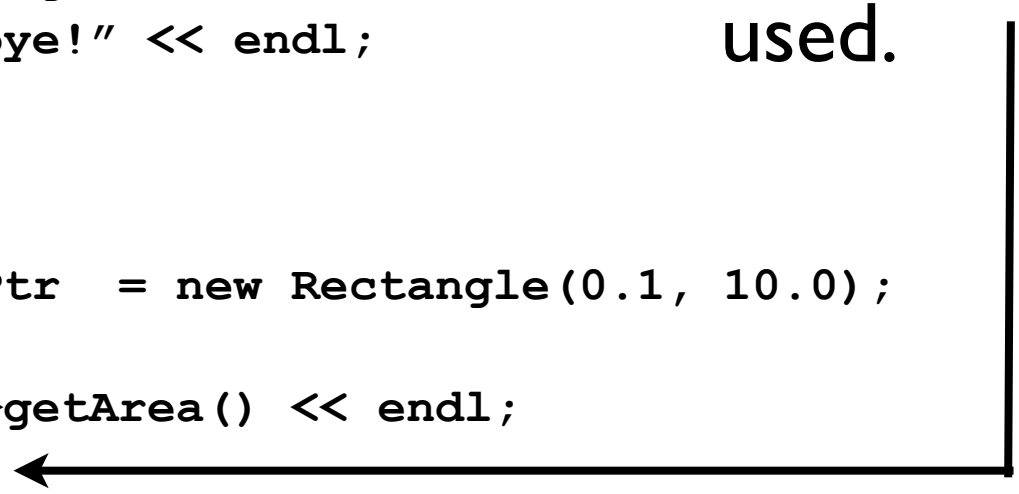
```
class Rectangle
{
   private:
      double width;
      double height;
   public:
      Rectangle(double w, double h);
      ~Rectangle();
      void setWidth(double w);
      void setHeight(double h);
      double getArea() const;
};

Rectangle::~Rectangle() {
   cout << "Bye bye!" << endl;
}

int main() {
   Rectangle * rPtr  = new Rectangle(0.1, 10.0);

   cout << rPtr->getArea() << endl;
   delete rPtr;
}
```

Destructors of dynamically allocated objects is called when the `delete` operator is used.



# Design your own class.

```
class Cat
{
```

```
};
```

**3 Member Variables**
**Constructor**
**Destructor**
**Accessor/Mutator Functions.**