Classes and Object-Oriented Programming

Chapter 11

Classes

- Classes allow the joining of related functions and data variables

Classes

- Classes allow the joining of related functions and data variables

- Different objects, or instances, of a class has its own copy of the data members

Classes

- A function in a class is generally called
  - member function
  - method

Classes

- Every member function has access to the ‘this’ pointer.

Classes: The ‘this’ Pointer

- The ‘this’ pointer
  - an implicit parameter that points at the object through which the function is called

Classes: The ‘this’ Pointer

- The ‘this’ pointer
  - an implicit parameter that points at the object through which the function is called
    - Since it is part of the object, it has access to everything public, private, and protected

Classes: The ‘this’ Pointer

```
class MyClass {
        public:
                int getData() {
                        return m_data;
                }

                void setData(int data) {
                        m_data = data;
                }
        private:
        int m_data;
};
```

Classes: The ‘this’ Pointer

- getData has no parameters and setData has one

```
int getData() {
	return m_data;
}
void setData(int data) {
	m_data = data;
}
```

Classes: The ‘this’ Pointer

- getData has no parameters and setData has one
  - In reality, they both have the hidden ‘this’ parameter

```
int getData() {
	return m_data;
}
void setData(int data) {
	m_data = data;
}
```

Classes: The ‘this’ Pointer

- Since the ‘this’ pointer is a pointer to the class instance, it can be used like any other pointer

Classes: The ‘this’ Pointer

- Since the ‘this’ pointer is a pointer to the class instance, it can be used like any other pointer

```
this->m_data = 7;
```

Classes: The ‘this’ Pointer

- For example:

```
int getData() {
        return this->m_data;
}
void setData(int data) {
        (*this).m_data = data;
}
```

Constant Member Functions

- In classes, it is often useful to declare a constant function

Constant Member Functions

- In classes, it is often useful to declare a constant function

- A constant function means that the function will make no changes to the class object

Constant Member Functions

- Constant functions are often used for accessors

Constant Member Functions

- **Constant functions are often used for accessors**

```
class MyClass {
        public:
                int getData() const {
                        return m_data;
                }
        ...
};
```

Constant Member Functions

- **Constant functions are often used for accessors**

```
class MyClass {
        public:
                int getData() const {
                        return m_data;
                }
        ...
};
```

Constant functions have the word ‘const’ placed after the parameters

Classes: Static Members

- Sometimes it is necessary that multiple objects of the same class share data and functions

Classes: Static Members

- Sometimes it is necessary that multiple objects of the same class share data and functions

- This can be done through static members

Classes: Static Members

- Static members are shared by all objects of the same class

Classes: Static Members

- Static members are shared by all objects of the same class
  - They are created by placing the keyword ‘static’ in front of functions or variables

Classes: Static Members

- Lets create a class with a static member variable

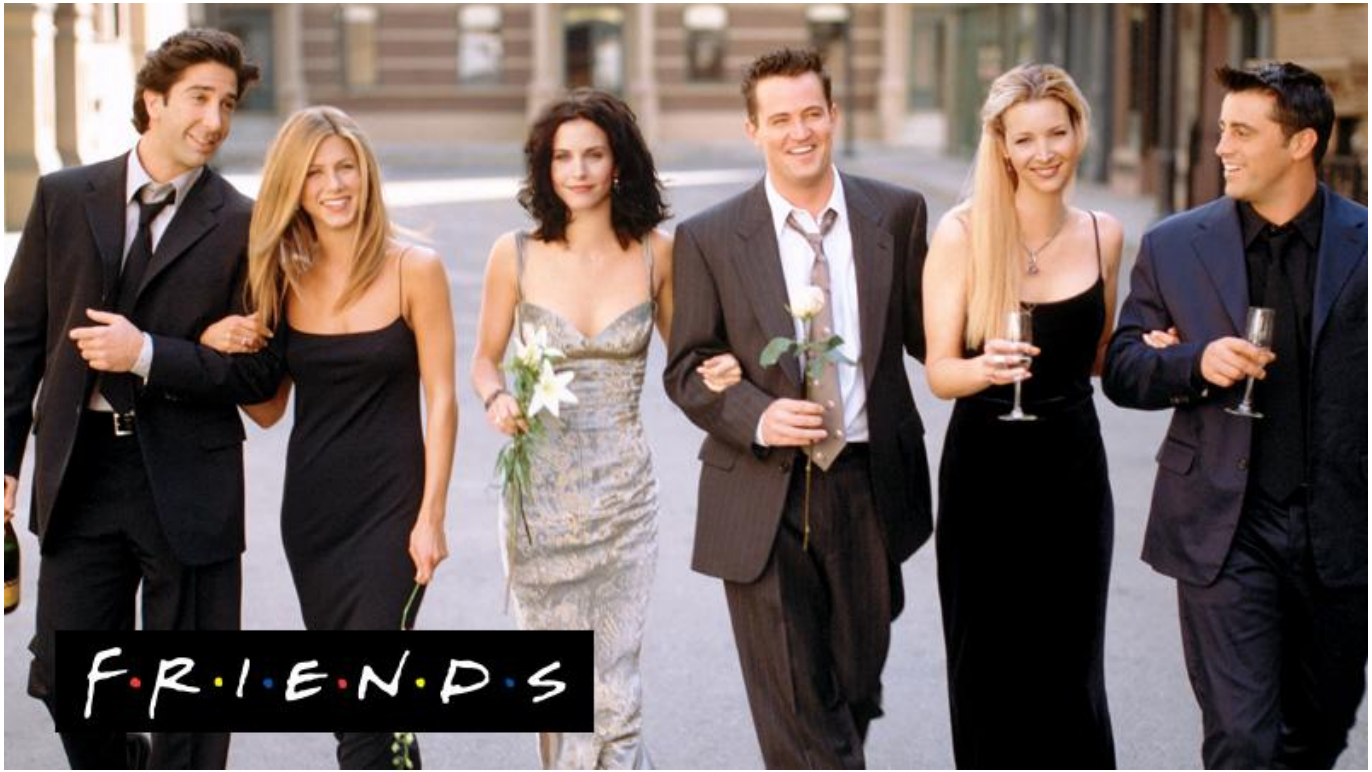
Classes: Static Members

- Lets create a class with a static member variable

- Add a static function

Class: Friends

Class: Friends



Class: Friends

- A friend is a non member function that can access private members

Class: Friends

- A friend is a non member function that can access private members

```
class MyClass{
	public:
		friend void MyClass2::modifyMyClassData();
	private:
		int m_data;
};
class MyClass2{
	public:
		void modifyMyClassData();
};
```

Memberwise Assignment

- Primitive types have the ability to be assigned to one another at any point

Memberwise Assignment

- Primitive types have the ability to be assigned to one another at any point

```
int a = 1;
int b = 2;
a = b;
```

Memberwise Assignment

- Primitive types have the ability to be assigned to one another at any point

```
int a = 1;
int b = 2;
a = b;
```

- This can also be done with classes

Memberwise Assignment

- Primitive types have the ability to be assigned to one another at any point

```
int a = 1;
int b = 2;
a = b;
```

- This can also be done with classes

Memberwise Assignment

```
class MyClass{...};

MyClass A;
MyClass B;

A = B;
```

or

```
MyClass A;
MyClass B = A;
```

Memberwise Assignment

```
class MyClass{...};

MyClass A;
MyClass B;

A = B;
```

or

```
MyClass A;
MyClass B = A;
```

This does not work for all classes!

Memberwise Assignment

```
class MyClass{...};

MyClass A;
MyClass B;

A = B;
```

This does not work for all classes!

or

```
MyClass A;
MyClass B = A;
```

What is actually happening here?

Memberwise Assignment

- What about this?

```
class MyClass{...};


MyClass A;
...
MyClass B(A);
```

Copy Constructors

- A copy constructor is called whenever a new object is created and initialized with data from another object of the same class

Copy Constructors

- A copy constructor is called whenever a new object is created and initialized with data from another object of the same class

```
MyClass A;
MyClass B(A);
```

Copy Constructors

- A copy constructor is called whenever a new object is created and initialized with data from another object of the same class

```
MyClass A;
MyClass B(A);
```

Object B is initialized with the data from A

Copy Constructors

- A default copy constructor is automatically created if the programmer does not specify one

```
MyClass A;
MyClass B(A);
```

OR

```
MyClass A;
MyClass B = A;
```

Copy Constructors

- A default copy constructor is automatically created if the programmer does not specify one

```
MyClass A;
MyClass B(A);
```

OR

```
MyClass A;
MyClass B = A;
```

The compiler smart enough to use the copy constructor instead of the normal constructor followed by an assignment

Copy Constructors

- A default copy constructor copies all data members from one class to another

Copy Constructors

- A default copy constructor copies all data members from one class to another
  - Are there any potential dangers?

Copy Constructors

- A default copy constructor copies all data members from one class to another
  - Are there any potential dangers?
    - Pointers
    - Dynamic Arrays

Copy Constructors

- A default copy constructor copies all data members from one class to another
  - Are there any potential dangers?
    - Pointers
    - Dynamic Arrays

What does a default copy constructor do if we have those data members?

Copy Constructors

- To prevent issues like that, the programmer can create his/her own copy constructor

Copy Constructors

- To prevent issues like that, the programmer can create his/her own copy constructor

```
MyClass::MyClass(MyClass& rObj)
{
        ...
}
```

Copy Constructors

- To prevent issues like that, the programmer can create his/her own copy constructor

```
MyClass::MyClass(MyClass& rObj)
{
        ...
}
```

Now, the programmer has the power to do whatever is deemed correct for any special members

Copy Constructors

- As part of a copy constructor, the constructor has access all data members of the passed in object

Copy Constructors

- As part of a copy constructor, the constructor has access all data members of the passed in object

```
MyClass::MyClass(MyClass& rObj)
{
        m_x = rObj.m_x;
        ...
}
```

Copy Constructors

- As part of a copy constructor, the constructor has access all data members of the passed in object

```
MyClass::MyClass(MyClass& rObj)
{
        m_x = rObj.m_x;
        ...
}
```

What about initialization lists?

Copy Constructors

- How could we handle pointers and dynamic arrays in a class copy constructor?

Copy Constructors

- Const can, and possibly should, be used in copy constructors

Copy Constructors

- Const can, and possibly should, be used in copy constructors

```
MyClass::MyClass(const MyClass& rObj)
{
        ...
}
```

Copy Constructors

- Const can, and possibly should, be used in copy constructors

```
MyClass::MyClass(const MyClass& rObj)
{
        ...
}
```

What does this tell us about the input parameter?

Copy Constructors

- Const can, and possibly should, be used in copy constructors

```
MyClass::MyClass(const MyClass& rObj)
{
        ...
}
```

What does this tell us about the input parameter?

Can we make the copy constructor a const function? ie MyClass(const MyClass& rObj) const

Operator Overloading

- Not only can we overload the copy constructor, but we can overload other functions as well

Operator Overloading

- Not only can we overload the copy constructor, but we can overload other functions as well
  - = operator
  - < operator
  - > operator
  - == operator
  - ...

Assignment Operator

- What if we modify our example a bit

```
MyClass A;
MyClass B = A;  // The copy constructor
MyClass C;


C = A;          // What function does this call?
```

Assignment Operator

- Overloading the assignment operator gives us similar control to the copy constructor

Assignment Operator

- Overloading the assignment operator gives us similar control to the copy constructor

```
void MyClass::operator=(const MyClass& rRight)
{
    ...
}
```

Assignment Operator

- For overloading most operators there is a pattern

Return type | Function name | Parameter for object ont he right side of the operator

```
void MyClass::operator=(const MyClass& rRight)
{
    ...
}
```

Assignment Operator

- For overloading most operators there is a pattern

Return type — Function name — Parameter for object ont he right side of the operator

```
void MyClass::operator=(const MyClass& rRight)
{
    ...
}
```

How might we fill this out for a class with pointers or dynamic memory?

Assignment Operator

- For primitives, the assignment operator supports chaining, what about ours?

Assignment Operator

- For primitives, the assignment operator supports chaining, what about ours?

```
int a = 1;
int b = 2;
int c = 3;

a = b = c;
```

Assignment Operator

- For primitives, the assignment operator supports chaining, what about ours?

```
int a = 1;
int b = 2;
int c = 3;


a = b = c;
```

How would we need to modify our assignment operator to support chaining?

Assignment Operator

- We need to modify the return type

Assignment Operator

- We need to modify the return type

```
MyClass MyClass::operator=(const MyClass& rRight)
{
    …
        return *this;
}
```

Assignment Operator

- We need to modify the return type

```
MyClass MyClass::operator=(const MyClass& rRight)
{
    …
        return *this;
}
```

How many times might the Class object be constructed when chaining a = b = c?

Assignment Operator

- We need to modify the return type

```
MyClass MyClass::operator=(const MyClass& rRight)
{
    …
        return *this;
}
```

How many times might the Class object be constructed when chaining a = b = c?

Can we improve this?

Assignment Operator

- We need to modify the return type again

```
MyClass& MyClass::operator=(const MyClass& rRight)
{
    …
        return *this;
}
```

Other Overloadable Operators

- There are many other overloadable operators

Other Overloadable Operators

+	-	*	/	%	^	&
|	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	|=
<<	>>	>>=	<<=	==	!=	<=
>=	&&	||	++	--	->*	,
->	[]	()	new	delete		

Arithmetic And Relational Operators

- The arithmetic and relational operators can also be overloaded

Arithmetic And Relational Operators

- The arithmetic and relational operators can also be overloaded
  - operator+
  - operator-
  - operator<
  - operator==

Arithmetic And Relational Operators

- The arithmetic and relational operators can also be overloaded
  - SomeClass operator+(SomeClass left, SomeClass right);
  - SomeClass operator-(SomeClass left, SomeClass right)
  - bool operator<(SomeClass left, SomeClass right);
  - bool operator==(SomeClass left, SomeClass right);

Arithmetic And Relational Operators

- It is even possible to overload these outside of a class

Arithmetic And Relational Operators

- It is even possible to overload these outside of a class
  - How might we do that?

Arithmetic And Relational Operators

- It is even possible to overload these outside of a class
  - How might we do that?

```
friend bool operator==(SomeClass left, SomeClass right)
{
    ...
}
```

Arithmetic And Relational Operators

- The usage of overloaded operators can be used automatically or manually

```
friend bool operator==(SomeClass left, SomeClass right) {
    … }
```

Arithmetic And Relational Operators

- The usage of overloaded operators can be used automatically or manually

```
friend bool operator==(SomeClass left, SomeClass right) {
    … }
SomeClass A, B;

if (A == B) …

if (operator==(A,B)) ...
```

Arithmetic And Relational Operators

- How might we use the operator== in a Student class?

Arithmetic And Relational Operators

- How might we use the operator== in a Student class?

- What about the operator<?

Other Operators

- Overloading the [] Operator is also important

Other Operators

- Overloading the [] Operator is also important
  - What if we created a list, array, or other container or objects?

Other Operators

- Overloading the [] Operator is also important
  - What if we created a list, array, or other container or objects?

SomeClass& operator[](int) const;

Other Operators

- Overloading the [] Operator is also important
  - What if we created a list, array, or other container or objects?

SomeClass& operator[](int) const;

Lets create a University class that contains many Student objects.

Type Conversion Operators

- Sometimes it is useful to convert a class into a different type

Type Conversion Operators

- Sometimes it is useful to convert a class into a different type
  - For example
    - We have a non trivial object that we want to convert to a const char*

Type Conversion Operators

- What other type conversions could we make?

Type Conversion Operators

- What other type conversions could we make?
  - double
  - int
  - ...

Type Conversion Operators

- What other type conversions could we make?
  - double
  - int
  - …

```
operator double() const;
operator int() const;
...
```

Type Conversion Operators

- What other type conversions could we make?
  - double
  - int
  - …

What is the difference between type conversion operators and functions like ‘getAsDouble()’?

```
operator double() const;
operator int() const;
...
```

Type Conversion Operators

- Could we convert one class to another?

Convert Constructors

- We can also go the opposite direction
  - converting something else into a class

Convert Constructors

- Convert Constructors
  - provide a means for the compiler to create objects using different types/objects

Convert Constructors

- A simple example

```
class MyString{
        public:
                MyString(std::string input){
                        …
                }


        private:
        UInt8* m_pArray;
};
```

Convert Constructors

- A simple example

```
class MyString{
        public:
                MyString(std::string input){
                        …
                }


        private:
        UInt8* m_pArray;
};
```

The argument is of a different type than the class. Here we convert a standard string into a MyString class.

Convert Constructors

- A convert constructor can be called automatically

Convert Constructors

- A convert constructor can be called automatically

```
class Foo{
public:
        Foo(int value)
          : m_value(value)
        {}
        int m_value;
};

Foo obj = 4;
```

Aggregation and Composition

- In some instances, it is important that a class of one type contains a class of another type

Aggregation and Composition

- If a class owns the class of another type, then this is called Class Aggregation

Aggregation and Composition

- If a class owns the class of another type, then this is called Class Aggregation

- Class Composition is similar to aggregation, except that the lifetime of the owned class coincides with the lifetime of the owner class

Aggregation and Composition

- Class Composition is similar to aggregation, except that the lifetime of the owned class coincides with the lifetime of the owner class

Aggregation and Composition

- A good example that shows the difference is as follows

Aggregation and Composition

```
class Date{ ...};

class Student {
        Student(Date birthday){...}

        Date getBirthday() {...}
        void setCurrentAddress(Address& rAddress){...}
        void removeCurrentAddress() {...}

        Date m_birthdate;
        Address* m_pCurrentAddress;
};
```

Aggregation and Composition

```
class Date{ ...};

class Student {
        Student(Date birthday){...}

        Date getBirthday() {...}
        void setCurrentAddress(Address& rAddress){...}
        void removeCurrentAddress() {...}

        Date m_birthdate;
        Address* m_pCurrentAddress;
};
```

The m_pCurrentAddress is added and removed during the lifetime of the student. Aggregation

Aggregation and Composition

```
class Date{ ...};

class Student {
        Student(Date birthday){...}

        Date getBirthday() {...}
        void setCurrentAddress(Address& rAddress){...}
        void removeCurrentAddress() {...}

        Date m_birthdate;
        Address* m_pCurrentAddress;
};
```

The m_birthdate object will be created when a student is created, and destroyed when the student is destroyed...Composition

Inheritance

- As in the real world, inheritance plays a big role

Inheritance

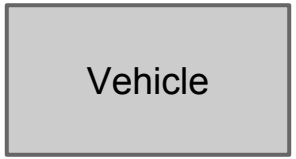
- As in the real world, inheritance plays a big role
  - Inheritance deals with specialized versions of things

Inheritance

- As in the real world, inheritance plays a big role
  - Inheritance deals with specialized versions of things

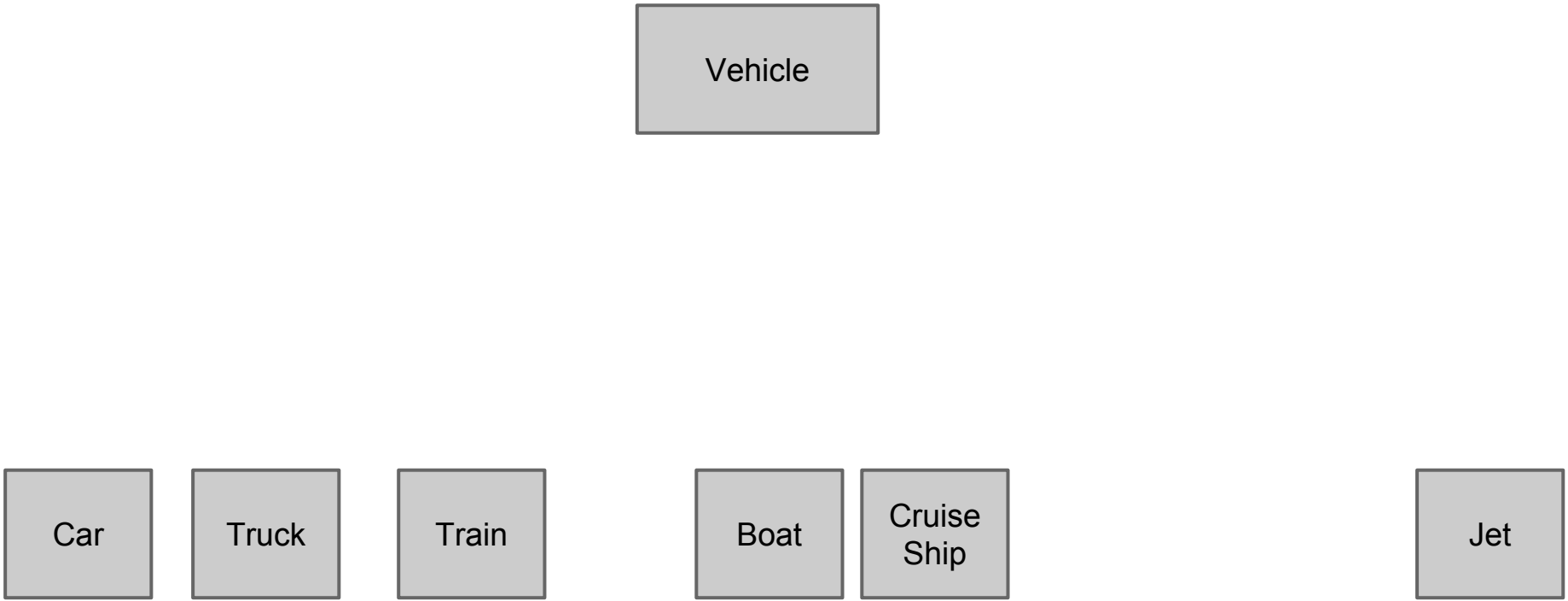
Inheritance

What type of vehicles are there?



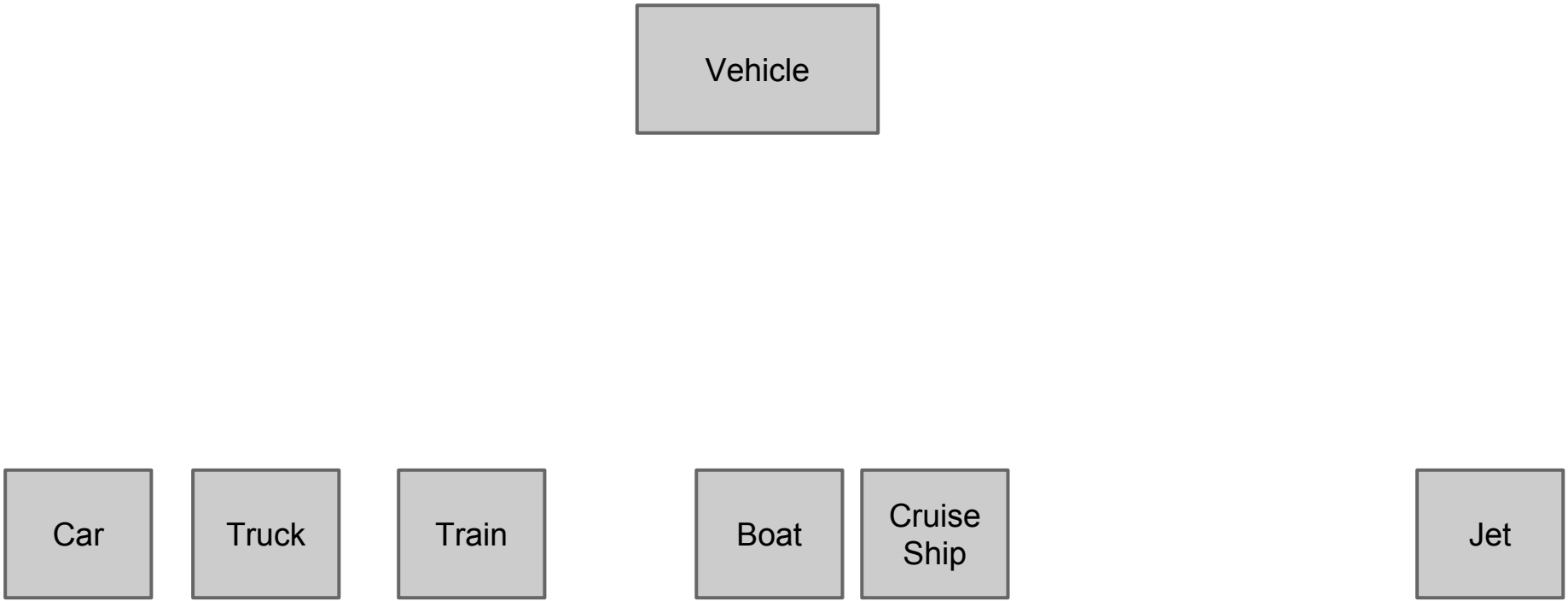
Inheritance

What type of vehicles are there?



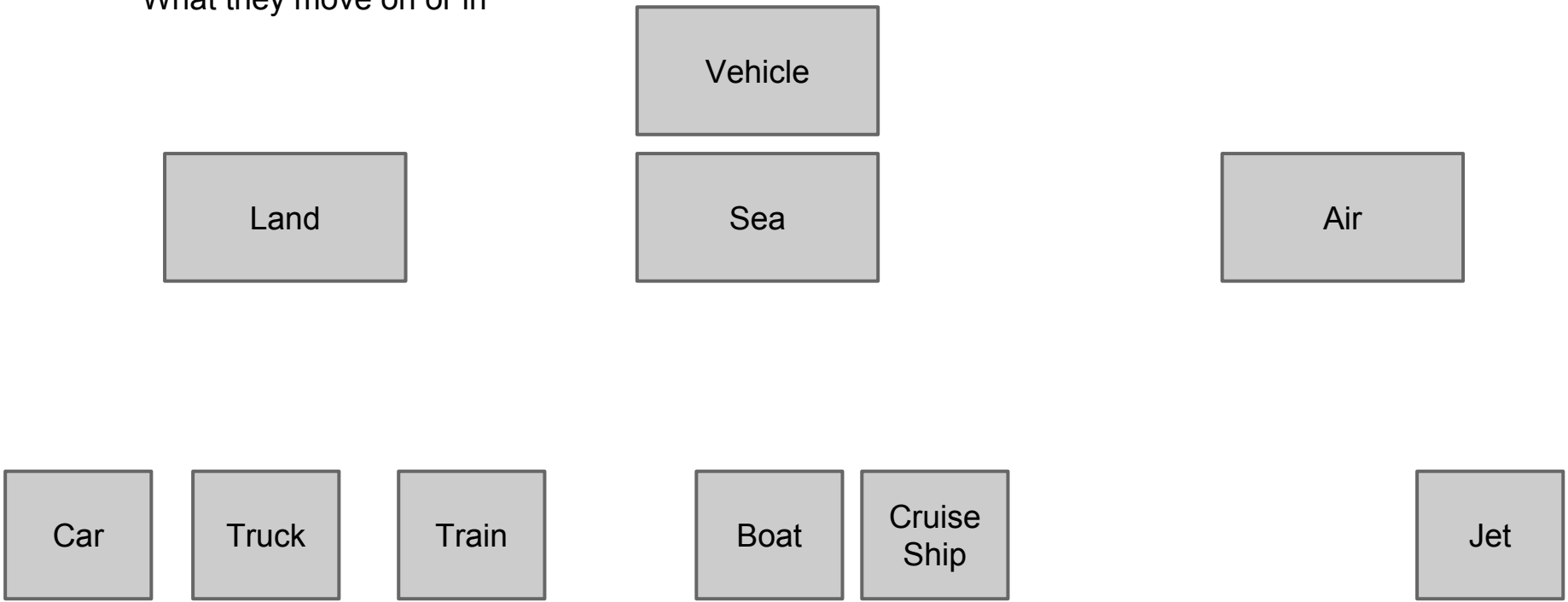
Inheritance

How could we characterize these?



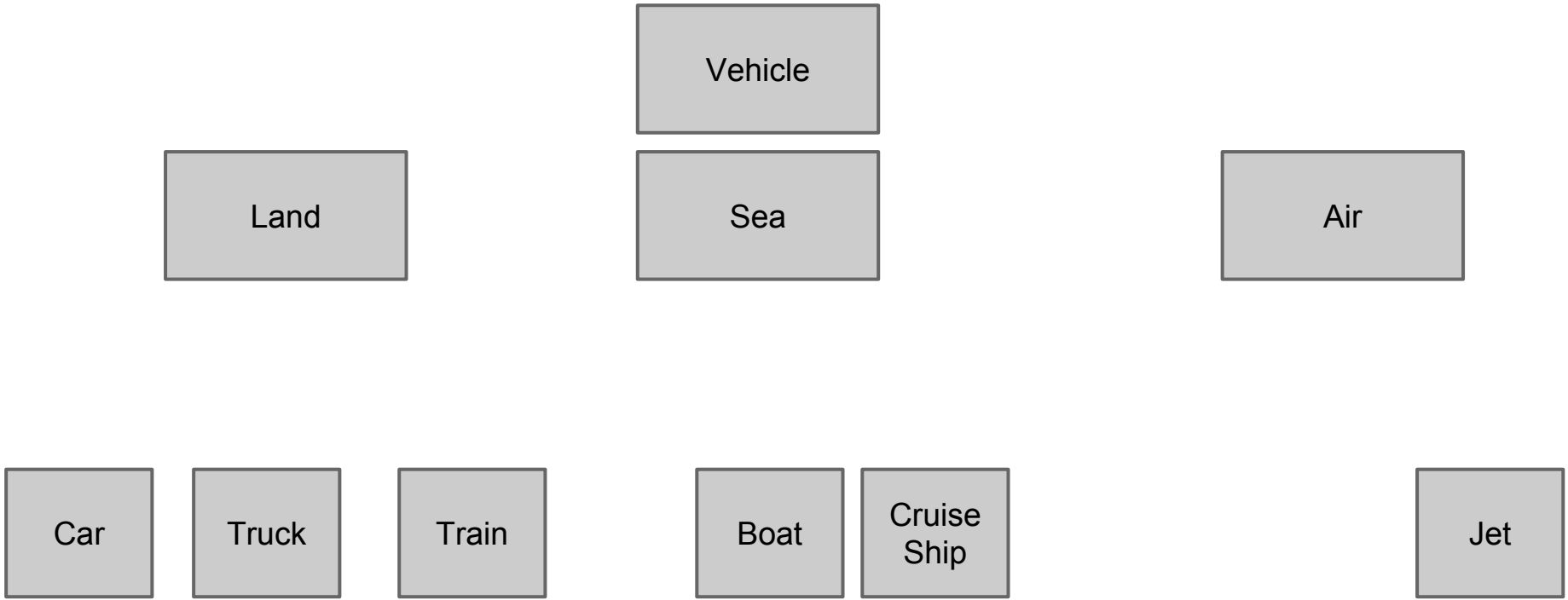
Inheritance

How could we characterize these?
What they move on or in



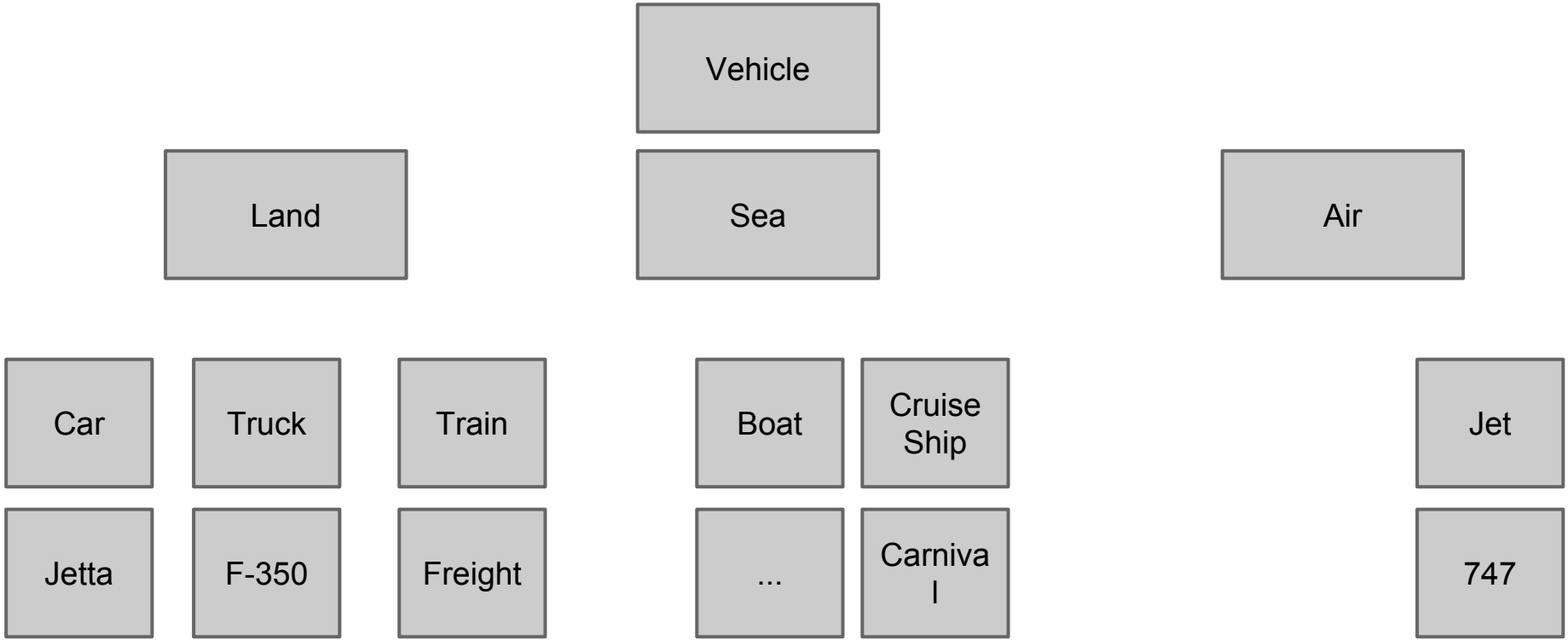
Inheritance

Could we get more specific?



Inheritance

Could we get more specific?

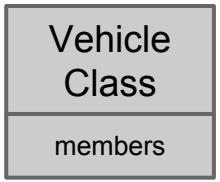


Inheritance

- Often times, inheritance follows an ‘is-a’ relationship
  - A Jetta is a Volkswagen
  - A rectangle is a shape
  - ...

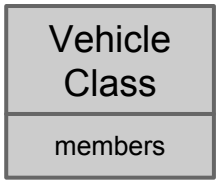
Inheritance

- To implement inheritance in C++ we create what is called a ‘base class’



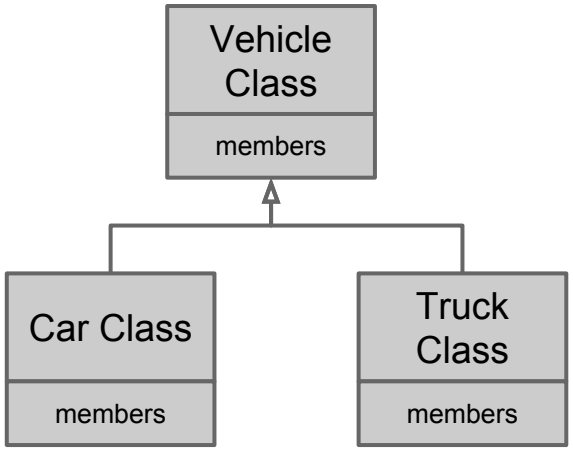
Inheritance

- To implement inheritance in C++ we create what is called a ‘base class’
  - The base class acts as a parent



Inheritance

- Next, we create one or many classes that inherit from the base class



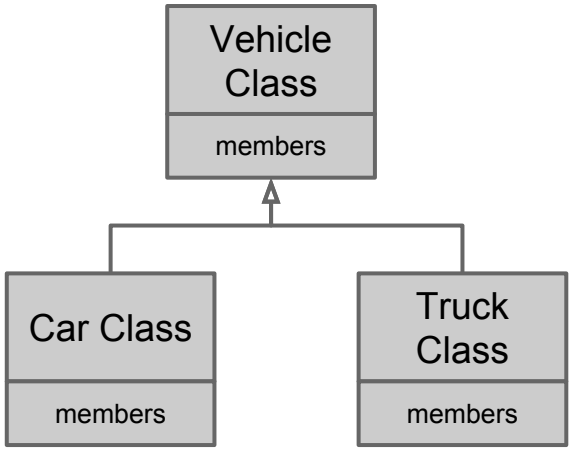
Inheritance

- Next, we create one or many classes that inherit from the base class

Derived Class

Base Class

```
class Car : public Vehicle
{
        ...
};
```



Inheritance

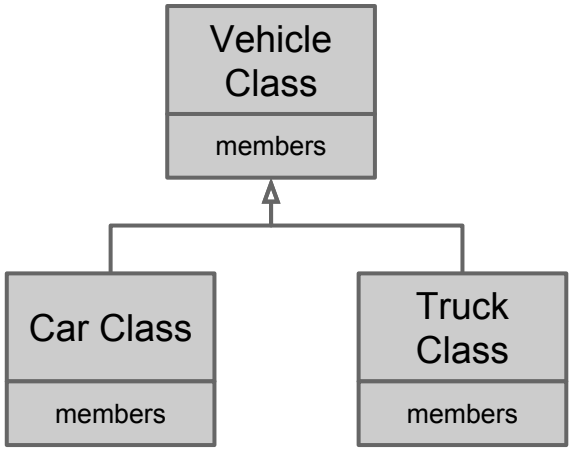
- Next, we create one or many classes that inherit from the base class

Derived Class

Base Class

```
class Car : public Vehicle
{
        ...
};
```

When the Car inherits from Vehicle, it gets all of Vehicles public and protect members and functions. Privates still exist but are not accessible from the derived class.



Inheritance

- Lets create an inheritance example

Protected Members & Class Access

- Class access is important
  - Any data members or functions that should be protected must not be accessible by anything that might modify them

Protected Members & Class Access

- Class access
  - Public
    - Anything/anyone can call, access, or modify

Protected Members & Class Access

- Class access
  - Public
    - Anything/anyone can call, access, or modify
  - Private
    - Only the owning class (or friends) can call, access, or modify

Protected Members & Class Access

- Class access
  - Public
    - Anything/anyone can call, access, or modify
  - Private
    - Only the owning class (or friends) can call, access, or modify
  - Protected
    - The owning AND derived classes can call, access, or modify

Protected Members & Class Access

- Protected and private base class access specifications can also be used

Protected Members & Class Access

- Protected and private base class access specifications can also be used

Class A : protected Letter {}

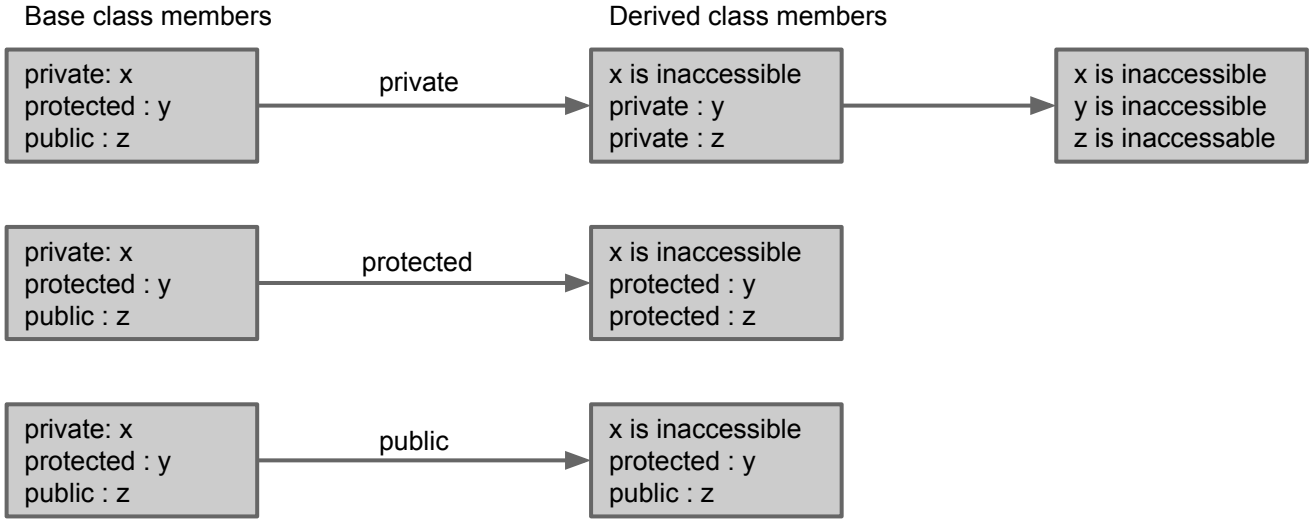
Class B : private Letter{}

Protected Members & Class Access

- Base class access is a little different than the normal access restrictions

Protected Members & Class Access

- Base class access is a little different than the normal access restrictions



Constructors, Destructors, and Inheritance

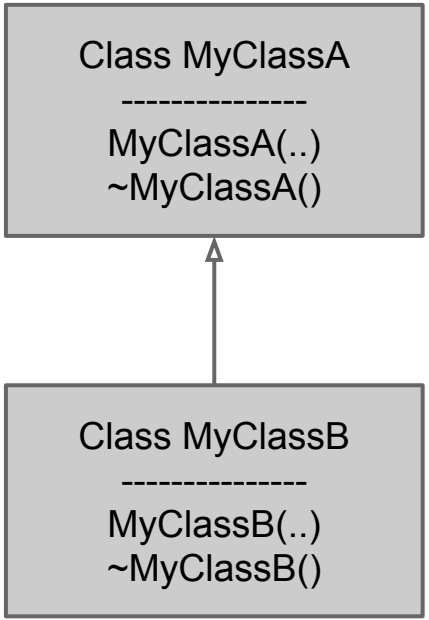
- With inheritance, the base class constructor gets called before the derived class constructor

Constructors, Destructors, and Inheritance

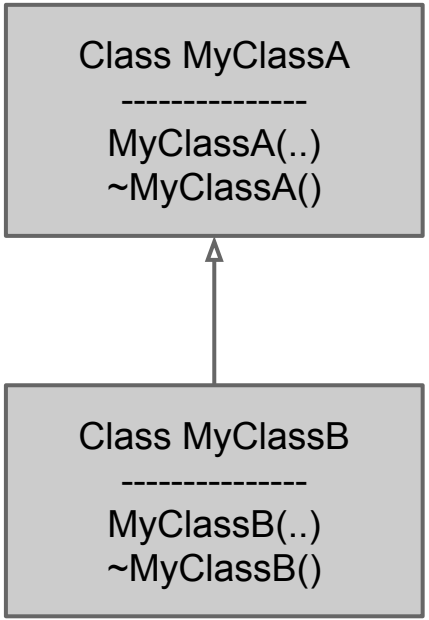
- With inheritance, the base class constructor gets called before the derived class constructor

- When the object is destructed, the derived destructor is called before the base class

Constructors, Destructors, and Inheritance



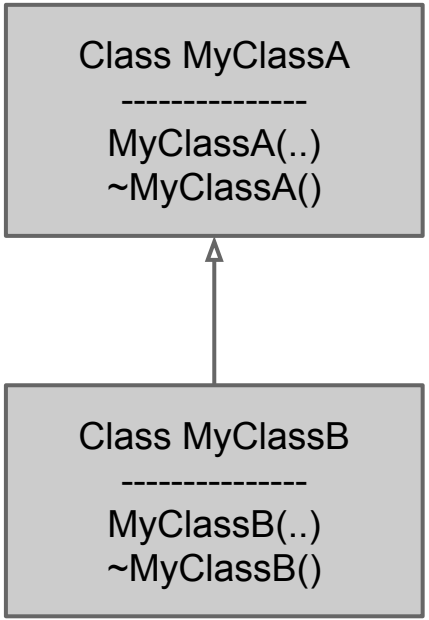
Constructors, Destructors, and Inheritance



On construction:

MyClassA();
MyClassB();

Constructors, Destructors, and Inheritance



On construction:

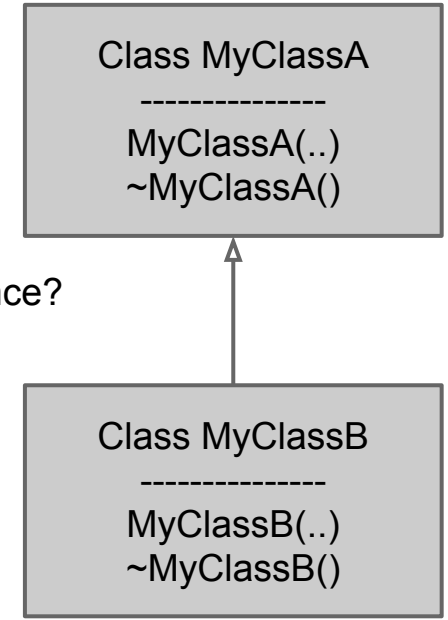
MyClassA();
MyClassB();

On destruction:

~MyClassB();
~MyClassA();

Constructors, Destructors, and Inheritance

What happens with multiple inheritance?



On construction:

MyClassA();
MyClassB();

On destruction:

~MyClassB();
~MyClassA();

Constructors, Destructors, and Inheritance

- Often, it is useful for the derived class to pass an argument to the base class

Constructors, Destructors, and Inheritance

- Often, it is useful for the derived class to pass an argument to the base class
  - This can be done through the initialization list

Constructors, Destructors, and Inheritance

- Often, it is useful for the derived class to pass an argument to the base class
  - This can be done through the initialization list

```
MyClassB()
  : MyClassA(TYPE_B)
{
   ...
}
```

Overriding Base Class Functions

- Base classes are a great place to put functions that should be shared amongst all derived classes

Overriding Base Class Functions

- Base classes are a great place to put functions that should be shared amongst all derived classes
  - Unfortunately, it is sometimes necessary for a derived class to change, or override, a base class function

Overriding Base Class Functions

- Override base class functions is as easy as redefining the same function in the derived class

Overriding Base Class Functions

- Override base class functions is as easy as redefining the same function in the derived class

```
Class A{
        int myFunction();
}

Class B : public A{
        int myFunction();
}
```

Overriding Base Class Functions

- Override base class functions is as easy as redefining the same function in the derived class

```
Class A{
        int myFunction();
}

Class B : public A{
        int myFunction();
}
```

The function name, return type, and parameter list should match!

Overriding Base Class Functions

- When overriding base class function, funny things happen when a base class pointer points to a derived class

Overriding Base Class Functions

- Even when a base class function is overridden, it is still accessible through different means

Overriding Base Class Functions

- Even when a base class function is overridden, it is still accessible through different means

```
MyClassA::myFunction();
```

Overriding Base Class Functions

- Even when a base class function is overridden, it is still accessible through different means

```
MyClassA::myFunction();
```

This can even be done from inside of MyClassB::myFunction();

Overriding Base Class Functions

- Even when a base class function is overridden, it is still accessible through different means

MyClassA::myFunction();

This can even be done from inside of MyClassB::myFunction();

This is very useful for overloaded equality operators!

Overriding Base Class Functions

- Overriding
  - Can only be done via inheritance
  - redefines something that was already defined
- Overloading

Overriding Base Class Functions

- Overriding
  - Can only be done via inheritance
  - redefines something that was already defined
- Overloading
  - Defining different functions within the same class with the same name and different parameters
  - Can be global