COP3503

Inheritance

Inheritance: What Is It?

One of the three pillars of object-oriented programming

Encapsulation	Inheritance	Polymorphism
Grouping data/functions together as objects	Creating new classes that reuse (inherit) functionality from existing classes	Use the same interface (i.e. functions) to access different types of objects.

We've seen this already.

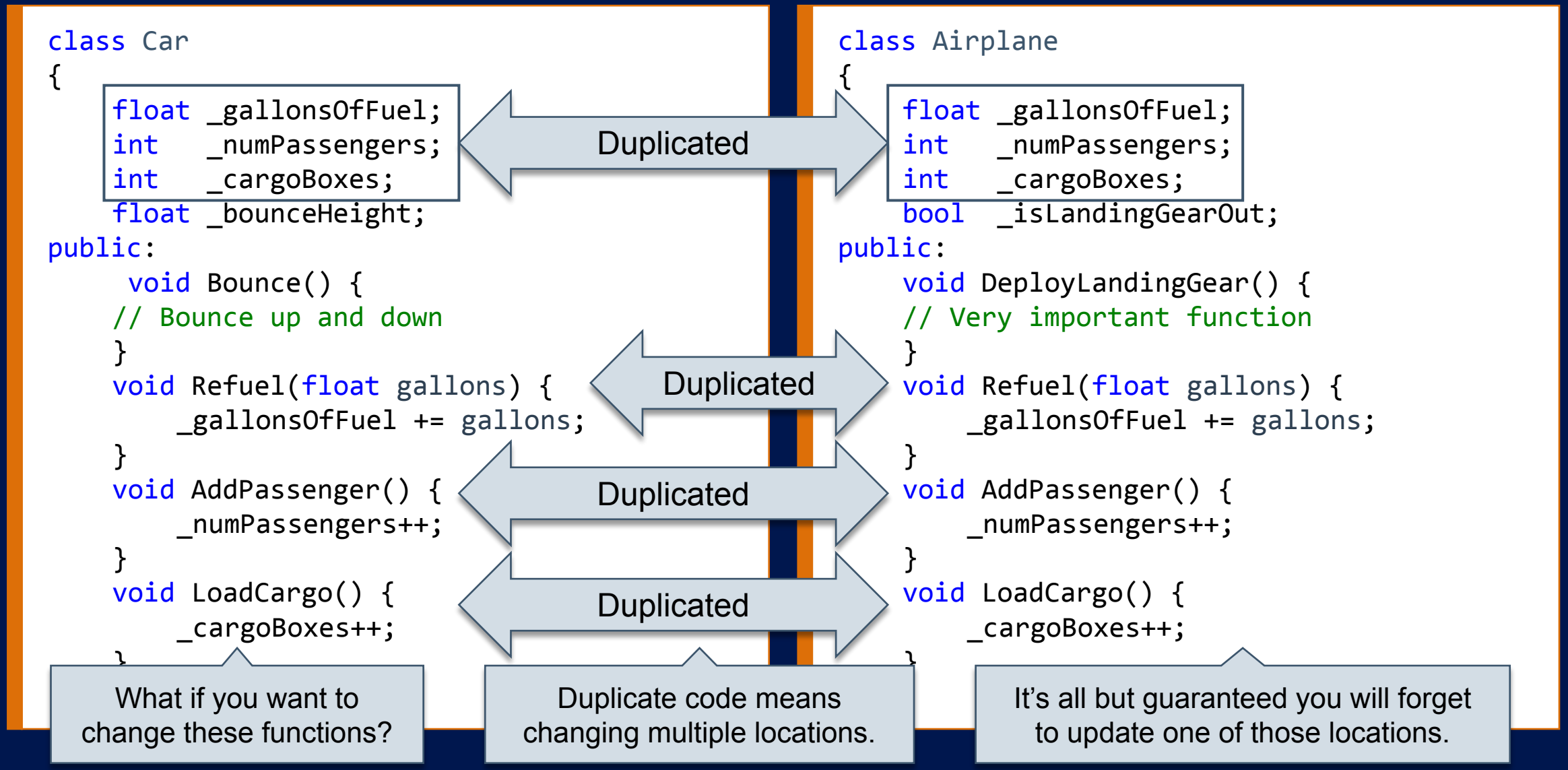
Polymorphism uses inheritance, so we'll cover it another time.

Inheritance Example

- What if you wanted to create a class for a Car and an Airplane?
- Things you might do with each of those:
  - `Refuel()`
  - `AddPassenger()`
  - `LoadCargo()`
- An Airplane may do something else like deploy landing gear (which a Car doesn't have)
- A Car may (hypothetically) do something like bounce up and down on a custom suspension system

You really don't want your airplanes doing this…

Duplicate Functionality == Duplicate Code



Inheritance to the Rescue!

```
// Shared code is put into a "generic" class
class Vehicle
{
    float _gallonsOfFuel;
    int   _numPassengers;
    int   _cargoBoxes;

public:
    void Refuel(float gallons) {
        _gallonsOfFuel += gallons;
    }
    void AddPassenger() {
        _numPassengers++;
    }
    void LoadCargo() {
        _cargoBoxes++;
    }
};
```

```
// Unique functionality goes in
// "specialized" classes
class Car : public Vehicle
{
    float _bounceHeight;

public:
    void Bounce() {}

};

class Airplane : public Vehicle
{
    bool _isLandingGearOut;

public:
    void DeployLandingGear() {}
};
```

This indicates we want to inherit the code from the Vehicle class.

Inheriting Lets Us Reuse Functionality

```
Vehicle someVehicle;

// Call vehicle functions
someVehicle.AddPassenger();
someVehicle.LoadCargo();
someVehicle.Refuel(2.6f);
```

An instance of **Car** is more capable than an instance of **Vehicle**.

```
Car someCar;

// Because Car inherits Vehicle...
someCar.AddPassenger();
someCar.LoadCargo();
someCar.Refuel(6.8f);
someCar.Bounce(); // #awesome
```

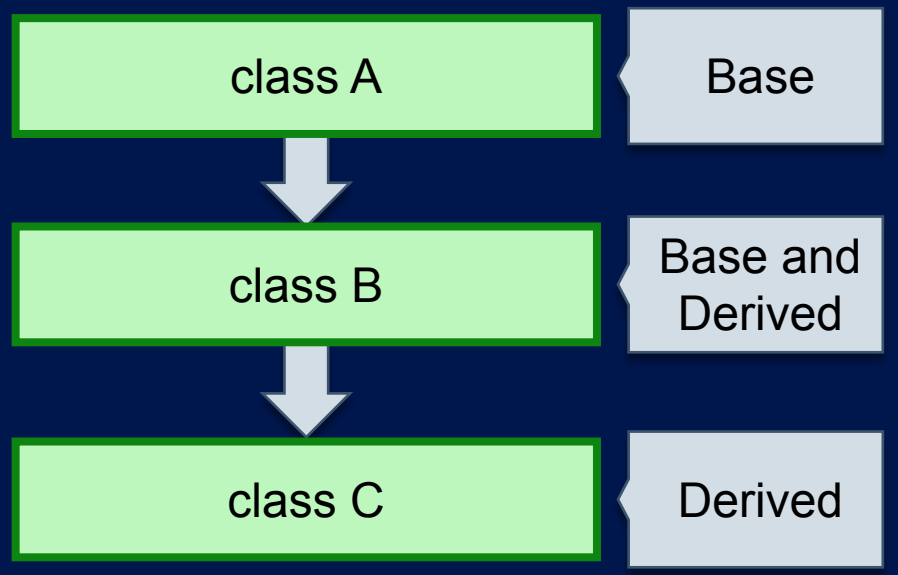
```
class Vehicle
{
    float _gallonsOfFuel;
    int   _numPassengers;
    int   _cargoBoxes;

public:
    void Refuel(float gallons);
    void AddPassenger();
    void LoadCargo();
};

class Car : public Vehicle
{
    float _bounceHeight;
public:
    void Bounce();
};
```

Inheritance Class Terminology

- Two classes are required for inheritance.
- **BASE class**:
  The original, with member variables and/or functions to pass on
- **DERIVED class**:
  Inherits the members and functions
- These are sometimes referred to as **PARENT** (base) class and **CHILD** (derived) class.
- You can derive a class from anything, even a class which itself is derived from another.
  - A class **can be both** derived **and** base (Just like a person could be both a child and a parent).



We typically don't go past parent and child to describe inheritance relationships.

Inheritance Syntax

```
class DerivedClass : public BaseClass
{
    // Whatever class code you want in here
};
```

`public` inheritance is the most common.

There is also **`protected`** and **`private`** inheritance, more on those later.

Language	Syntax
C#	`class Derived : Base`
Java	`class Derived extends Base`
Python	`class Derived(Base):`

Most languages don't have "types" of inheritance like C++, but the concept is the same (and public inheritance is the most common).

What Does “Inheriting” Actually Do?

- A **derived** class “gets” all of the data and functionality from the base class.
- All **public** member variables and functions.
- All **protected** member variables and functions (we’ll look at protected more later).
- **Private data stays private**—technically the derived class **has** them…
  it just can’t access them (more on this later).
  - You need public/protected accessibility in the base class for that.
- The new derived class can use all of the inherited functionality—it “is-a” base class.

The “Is-A” Relationship

- A derived class commonly has a “**Is-A**” relationship with the base class.
  - A Car IS-A Vehicle (Car has Vehicle functionality and more).
  - A Hero IS-A Person (Hero has Person functionality and more).
- If the statement **<DerivedClass> IS-A <BaseClass>** doesn’t sound right, inheritance may not be a good idea.

```
class Vehicle {};

// A car IS A vehicle—no problem!
class Car : public Vehicle {};

// This says a person IS A Car...
class Person : public Car {};
```

```
// A person is a car (and by extension, a Vehicle?)
Person p;
p.LoadCargo(); // Like, a backpack, or...?
p.Refuel(2.74f); // Eating lunch? #proteinshakebro
```

You should consider why you’re deriving a new class, and only do so if it makes sense!

Accessibility Keywords Revisited

Public

Any code inside a class or outside the class can access this

Private

- **Only** the defining class can access this
- The “Hands off, this is mine!” keyword

Protected

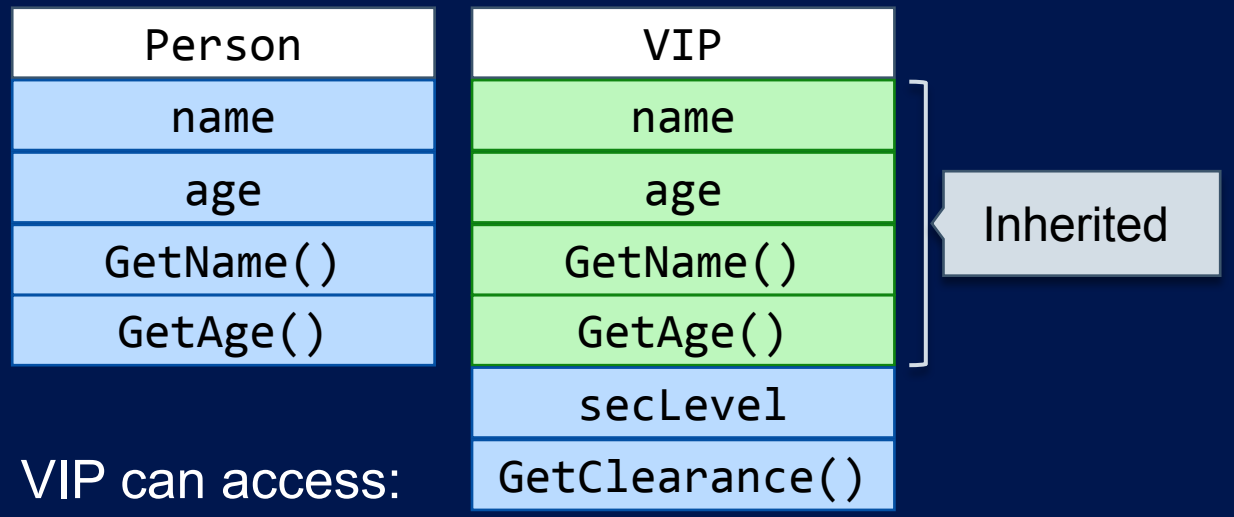
- A base class **and** any classes derived from it can access this
- The “Members Only” keyword

Applicable to data/functions in a class, and to indicate the type of inheritance

If you aren’t using inheritance at all, you don’t need to use protected.

```cpp
class Person
{
    string _name;
protected:
    int _age;
public:
    string GetName() {
        return _name;
    }
    int GetAge() {
        return _age;
    }
};
```

```cpp
// Public inheritance - most common
class VIP : public Person
{
    // Security Clearance Level
    int _secLevel;
public:
    int GetClearanceLevel() {
        return _secLevel;
    }
};
```



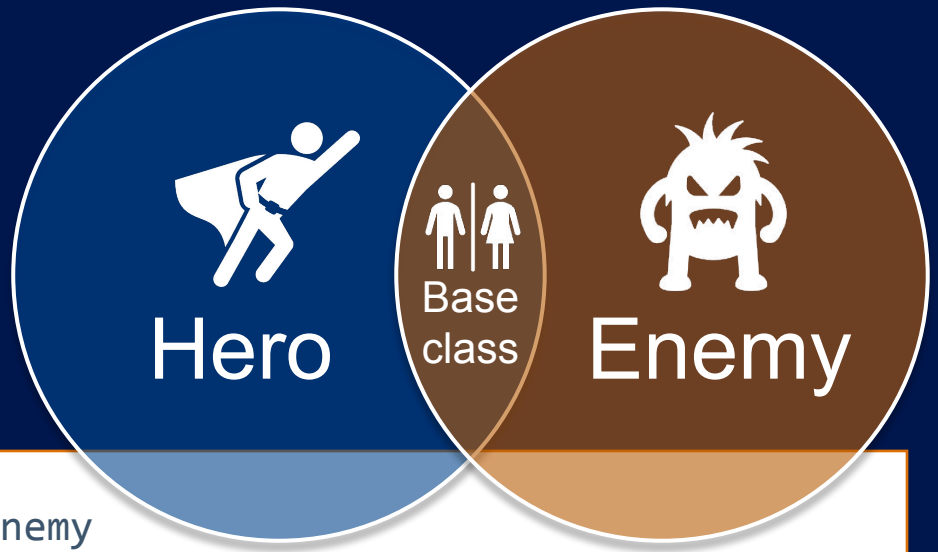
- VIP can access:
  - `GetName()` and `GetAge()`: They're public.
  - `age`: It's protected, but VIP inherits this class.
- VIP can **not** access `name` (it's private).

```cpp
void VIP::Print()
{
    cout << _name; // Error, _name is private
    cout << GetName(); // Okay, public function
    cout << _age; // Okay, protected is accessible
    cout << _secLevel; // Okay, it "owns" this
}
```

Game Example

Heroes and Enemies

It may be worth extracting duplicate code into a base class.




```
class Hero
{
    // Accumulated stuff
    int _experiencePoints;
    int _level;
    int _gold;

    string _name;
    int _hitpoints;
    int _strength;
public:
    // Accessors, mutators
};
```

Duplicated

```
class Enemy
{
    // Reward for defeating this enemy
    int _expValue;
    int _goldValue;

    string _name;
    int _hitpoints;
    int _strength;
public:
    // Accessors, mutators
};
```

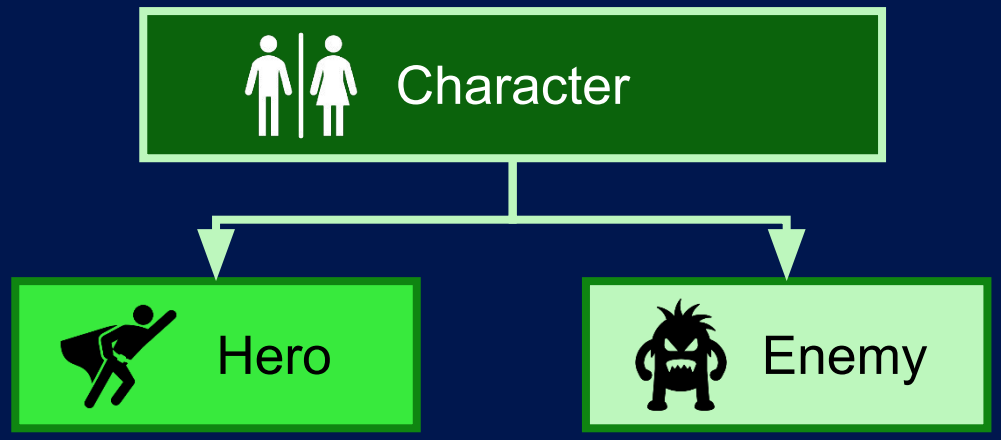
Any time there is “significant” overlap you may want to use a base class.

That can be subjective—there’s no single standard for it.

Game Example

Heroes and Enemies

You may hear this referred to as the **inheritance tree** or **inheritance hierarchy**.



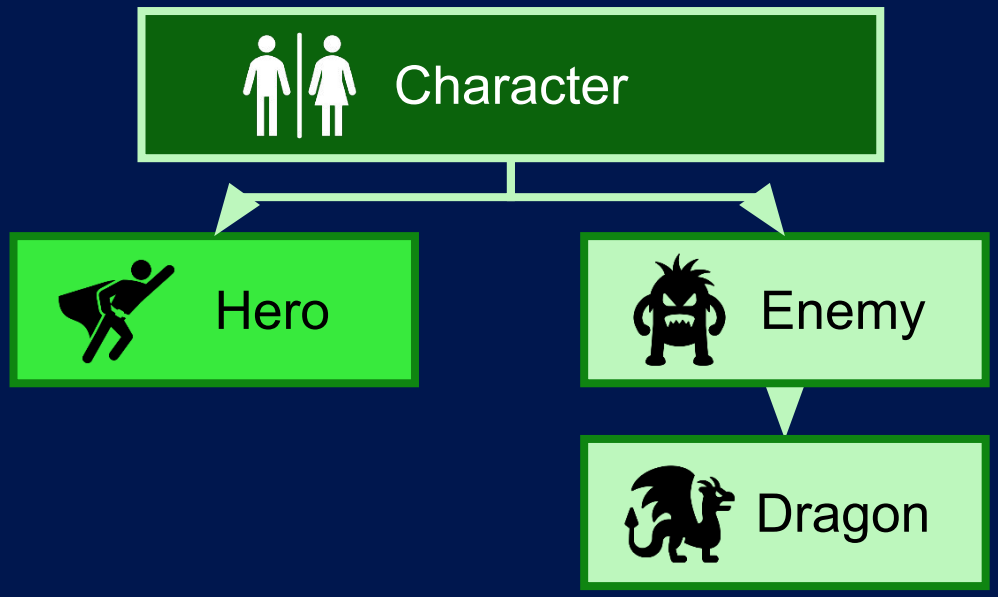

```
class Character
{
    string _name;
    int _hitpoints;
    int _strength;
public:
    // Accessors, mutators, etc
};
```

```
class Hero : public Character
{
    // Accumulated stuff
    int _experiencePoints;
    int _level;
    int _gold;
};
```

```
class Enemy : public Character
{
    // Reward for defeating this enemy
    int _expValue;
    int _goldValue;
};
```

Expanding and Using the Hierarchy

```
class Enemy : public Character
{
    int _expValue;
    int _goldValue;
};
```



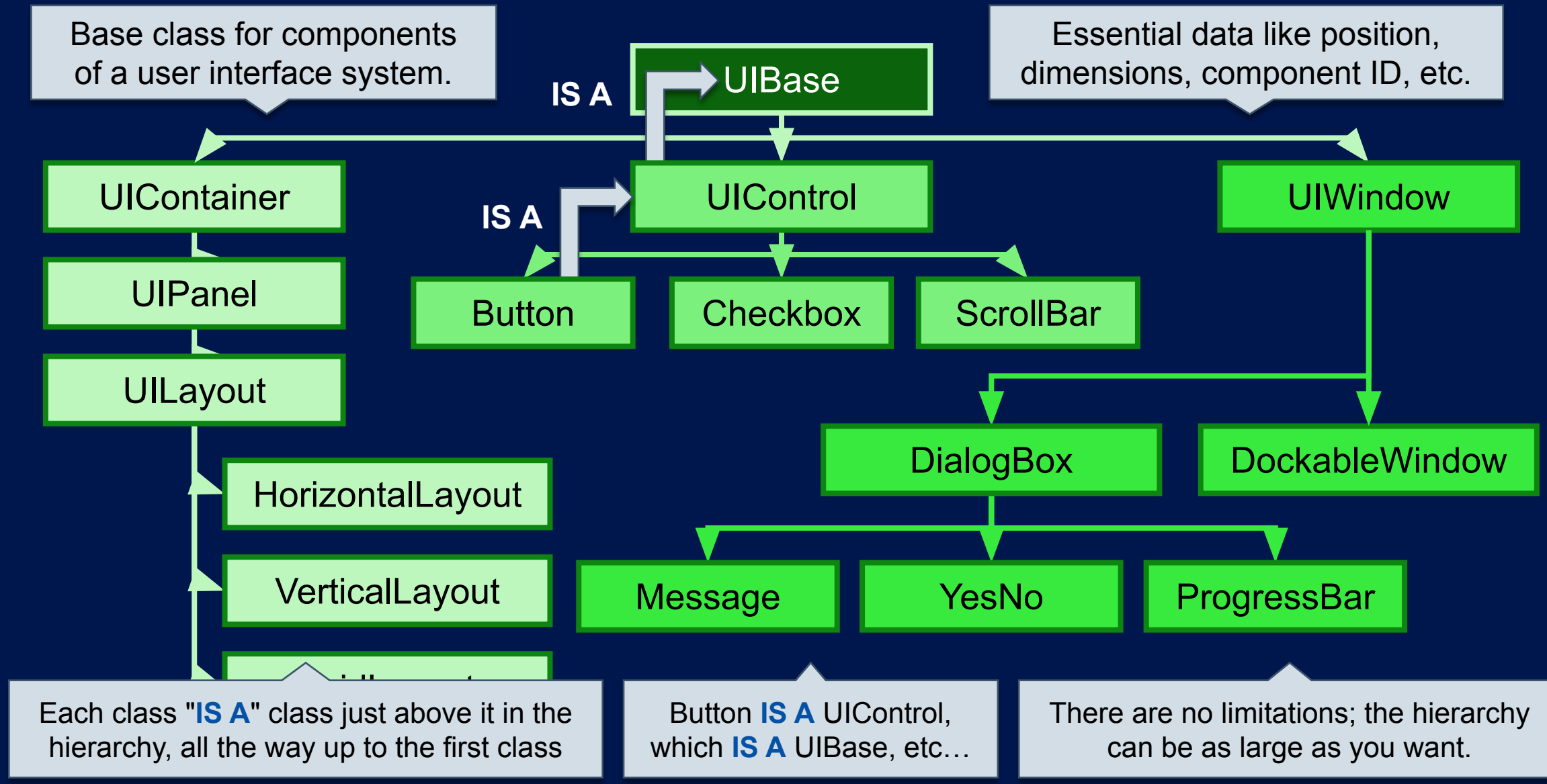

```
// Create a dangerous boss monster...
class Dragon : public Enemy
{
    int _armor;
    int _fireDamage;
public:
    void BreatheFire
    void Fly();
    void DoCoolDragonStuff();
};
```

Because of the “Is-A” relationship, all of these objects are **Characters**, some are **Characters-Plus-More**.

Any changes to the `Character` class would affect all of them!

```
int main()
{
    vector<Character> townsfolk;
    Hero thePlayer;
    vector<Enemy> badGuys;
    Dragon bigBadBoss;
    return 0;
}
```

Hierarchies Can Get Complex, Real Fast!



Containment

Objects Storing Objects

```
class Car
{ // Super cool Car stuff here };

// The Person class CONTAINS, or HAS, a Car
// And it contains a string
// And it contains a vector<int>
class Person
{
    Car _myCar;
    string _name;
    vector<int> _numbers;
public:
    void Foo()
    {
        cout << "Car's make: " << _myCar.GetMake();
    }
};
```

Containment is used regularly,
in all languages.

Some languages (like C++) support
other concepts that **model containment**
in slightly different ways.

Private Inheritance

Models a “Has-A” Relationship

```
// Private inheritance == Has-A relationship
class Person : private Car
{
    Car _otherCar; // Containment == "Has-A" relationship
public:
    void Foo()
    {
        cout << "Car's make: " << Car::GetMake(); // Inherited car
        cout << "Car's make: " << _otherCar.GetMake(); // Containment
    }
};

Car& Person::GetCar()
{
    return Car? return Car::Car? return Person::Car?
    return _otherCar; // Okay to return using containment
}
```

Same syntax, just a different keyword

The inherited object doesn’t have a name.

`InheritedType::Function()` or `InheritedType::Variable` is how we access the class.

Because the inherited data isn’t a variable, it’s not “available” outside the class.

It’s not hugely important that you master private inheritance.

We’re talking about it to point out how language features may offer options.

Protected Inheritance

A Rarely-Used Oddity

Protected and private inheritance have "quirks" and may not be used all that often.

Many other languages don't support anything other than public.

- Private inherited members stays private, as always.
- Protected inherited members stay protected.
- The main feature: Public inherited members **become** protected in the derived class.
  - Public members turn into "club members only" access.
- If B inherits A, only derived classes know about this inheritance.
  - Any code outside the "club" doesn't know B derives from A.
  - Why would this be useful? That's a good question…
- Your mileage may vary… but I have personally have **never** used protected inheritance (and in this course, it will never be mentioned again!).

Inheritance and Object Creation

- When you instantiate a derived class, it must be **constructed** (this isn't new!)
- The base class also must be constructed—objects are instantiated in **parts**.

```
class Person
{
    string
    int _a
public:
    Person(const char* name, int age);
};

class VIP : public Person
{
    int _secLevel;
public:
    VIP(const char* name, int age, int secLevel);
};
```

…get into **this** constructor?

…which gets sent to this constructor…

How does this data…

```
VIP ambassador("Bob", 32, 6);
```

What we need (but can't write here)

```
ambassador.PersonConstructor("Bob", 32);
```

The solution for this is the **Constructor Initializer List**.

Constructor Initializer List

A way to call the constructor of the base class (and pass data if necessary)

```
VIP::VIP(const char* name, int age, int secLevel) : Person(name, age)
{
    _secLevel = secLevel;
}
```

Constructor Initializer List

This invokes the base class constructor.

Data can be “forwarded” to the base class (to use in any way it needs).

VIP can still do whatever it needs to in its own constructor.

If needed, we can “hard code” default values to provide to base classes.

```
VIP::VIP() : Person("Default VIP", 99)
{
    _secLevel = 0;
}
```

VIP could have any number of constructors.

Each one must send the Person constructor what it needs (`string` and `int`, in this example).

If the base class has a default constructor, you can omit this part (it gets called automatically).

Initializer List Without Inheritance

- An alternative, more efficient way to initialize member variables.
  - Performance gains may be minimal for small data.
- Also allows for setting per-object `const` variables.

```
class Example
{
    int _x, _y;
    const int _constValue; // Typically this is initialized here
public:
    Example(int param1, int param2, int param3);
};

Example::Example(int param1, int param2, int constParam)
: _x(param1), _y(param2), _constValue(constParam)
{
    // x = param1; Slower alternative
    // y = param2; Slower alternative
    // constValue = param3; Compiler error: can't change a const value
}
```

In this course, this can just be treated as a style preference.

Later in your programming career? It could be a significant optimization!

Multiple Inheritance

- It's possible to derive from multiple base classes.
- Not all languages support this concept—for some, only a single base class is allowed.

```
class A {/*Insert cool stuff here*/};
class B {/*Insert cool stuff here*/};
class C {/*Insert cool stuff here*/};

class OmniClass : public A, public B, public C
{
public:
    OmniClass(int x, float y, short z);
};

OmniClass::OmniClass(int x, float y, short z)
: A(x), C(y, z) // B() is default for this example
{
}
```

Base classes must still be constructed and given whatever they need (if anything).

Multiple Inheritance Example

```
class Vehicle
class Car : public Vehicle
class Airplane : public Vehicle
```

- Now, how to make a flying car…

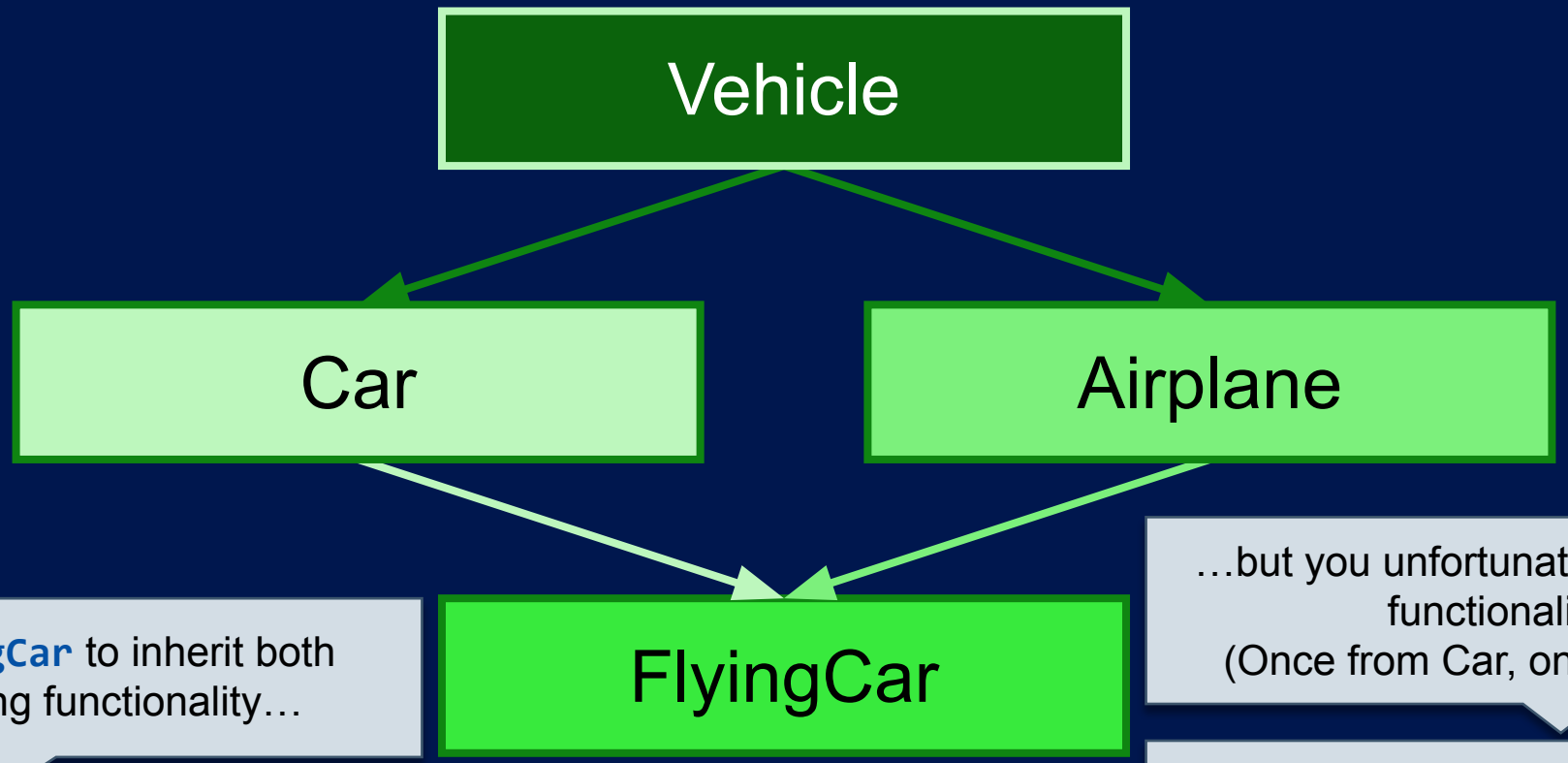
```
class FlyingCar : public Car, public Airplane
{ };

// Elsewhere…

FlyingCar delorean;
delorean.Bounce();
delorean.DeployLandingGear();
delorean.AddPassenger();
// delorean.ProbablyFlyIntoARoof()
```

There is an issue, however…
Because both Car and Airplane are **both** Vehicles…

The Diamond Problem

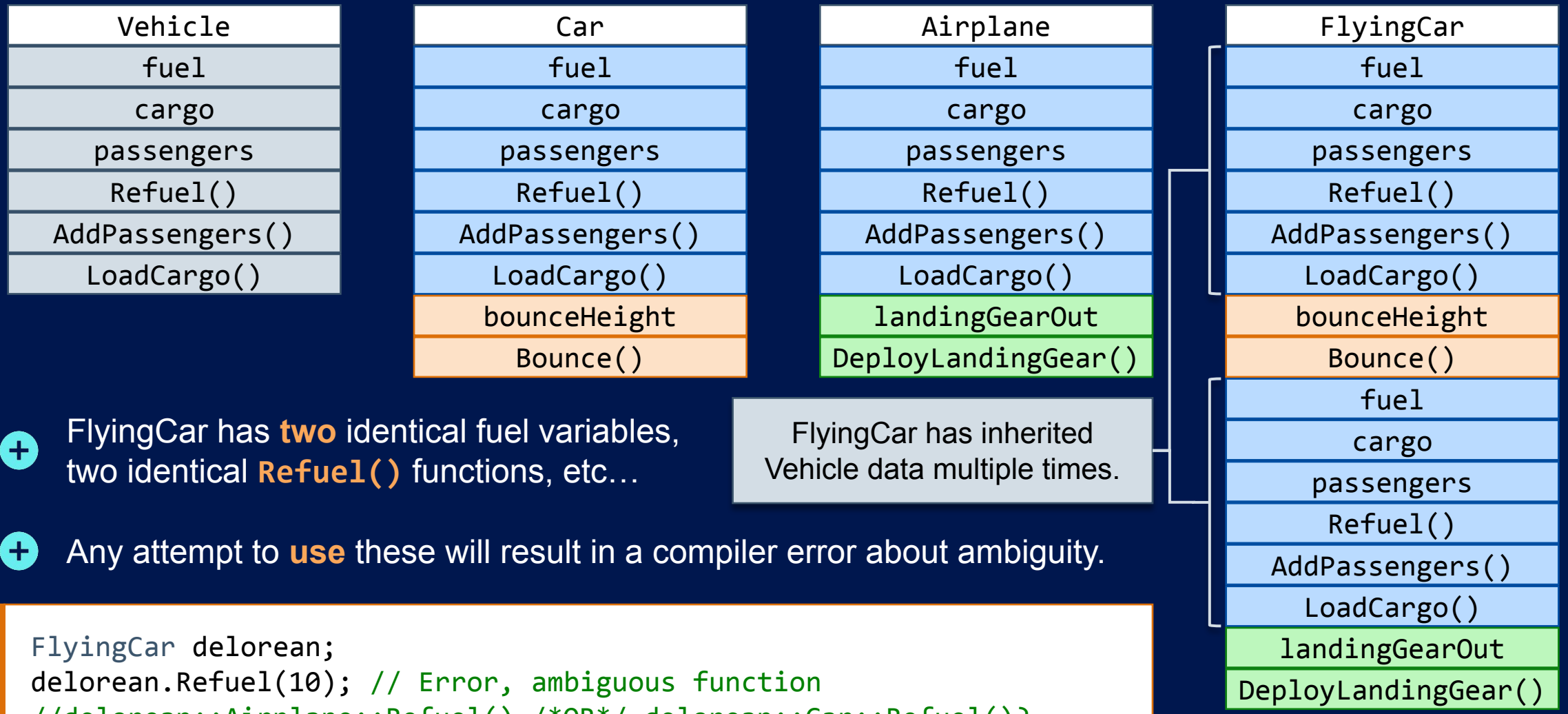


You want `FlyingCar` to inherit both driving and flying functionality…

…but you unfortunately inherit `Vehicle` functionality twice!
(Once from Car, once from Airplane)

The result is duplicate, “ambiguous” code that confuses the compiler.
(Sorry, compiler!)

The Diamond Problem



- FlyingCar has **two** identical fuel variables, two identical `Refuel()` functions, etc…
- Any attempt to **use** these will result in a compiler error about ambiguity.

```
FlyingCar delorean;
delorean.Refuel(10); // Error, ambiguous function
//delorean::Airplane::Refuel() /*OR*/ delorean::Car::Refuel()?
```

Solution: `virtual` Inheritance

That's all you need to do to solve the diamond problem on a basic level.

For more explanation on this, check out: https://isocpp.org/wiki/faq/multiple-inheritance

- With virtual inheritance, the "in-between" classes would add the `virtual` keyword to the inheritance:

```
class Vehicle
//class Car : public Vehicle
//class Airplane : public Vehicle

// virtual inheritance for "in between" classes
class Car : virtual public Vehicle
class Airplane : virtual public Vehicle

// Derive as normal for FlyingCar
class FlyingCar : public Car, public Airplane
```

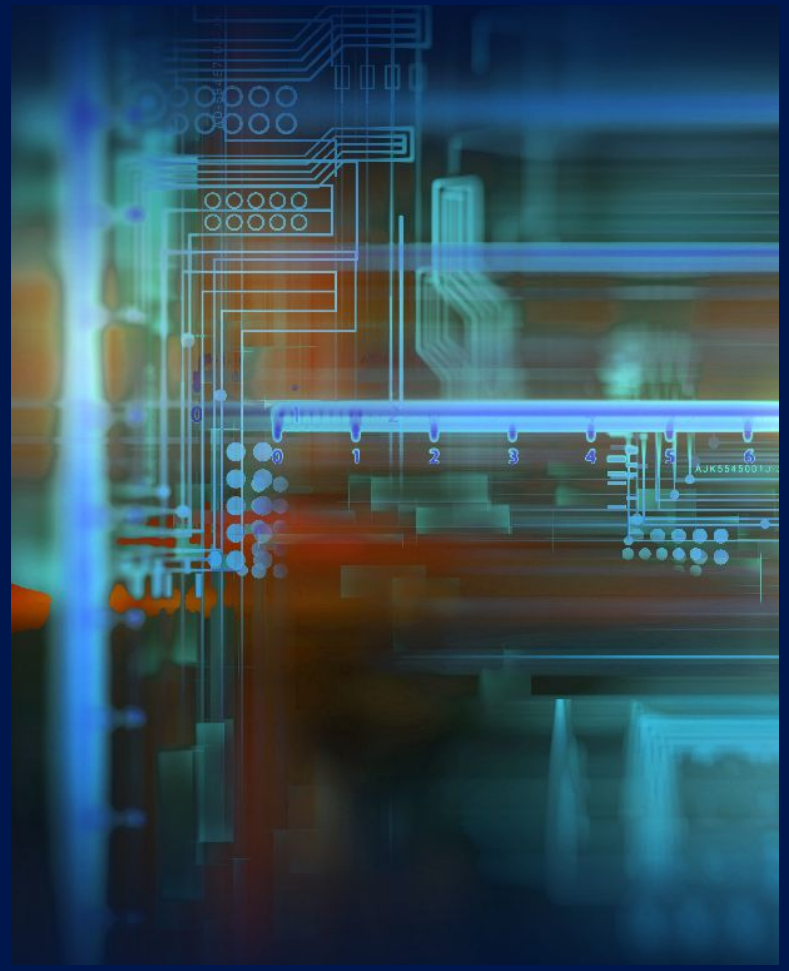
Inheriting Vehicle twice causes the problem in the first place.

Virtual tells the compiler "don't inherit stuff more than once".

Recap

- Inheritance is one of 3 pillars of object-oriented programming.
  - A way to reuse code by **deriving** new classes from existing **base** classes.
- **Base (parent) class**: The original class
- **Derived (child) class**: New class that inherits, or "gets" functionality from the base class
- Inheritance often models an "Is-A" relationship—**a derived IS A base**.
  - A Car IS A Vehicle, a Hero IS A Character, etc.
  - If you say it that way and it sounds funny, inheritance might not be the right choice! (A Hero IS A Vehicle…? A Car IS A Character…?)
- The concept is the same across most languages, with slight differences like multiple or private inheritance.

Conclusion



Placeholder for the instructor’s welcome message. Video team, please insert the instructor’s video here.

Thank you for watching.