DIGITAL ARTS & ENTERTAINMENT

Polymorphism

virtual functions

howest
De Hogeschool West-Vlaanderen
Lid van de Associatie Universiteit Gent

Inheritance

```
class Animal
{
public:
  void MakeSound() const { std::cout << "rawr\n"; }
};

class Cat : public Animal
{
public:
  void MakeSound() const { std::cout << "Miauw\n"; }
  void EnableNightVision(){}
};
int main()
{
  Cat* pCat{ new Cat{} };
  pCat->MakeSound();
  pCat->EnableNightVision();
}
```

```
output:
Miauw
```

Example using dynamic memory allocation.

- Derived class object.
- Derived class pointer.

Result: as expected

- prints Miauw
- executes night vision

Inheritance

```cpp
class Animal
{
public:
  void MakeSound() const { std::cout << "rawr\n"; }
};

class Cat : public Animal
{
public:
  void MakeSound() const { std::cout << "Miauw\n"; }
  void EnableNightVision(){}
};
int main()
{
  Animal* pAnimal{ new Cat{} };
  pAnimal->MakeSound();
  pAnimal->EnableNightVision(); // COMPILE ERROR
}
```

```
output:
rawr
```

Example using dynamic memory allocation.

- Derived class object.
- Base class pointer.

Result:

- Animal::MakeSound function is executed.
- Cat::MakeSound is "invisible" for the compiler.
- Compile error: NightVision

Conclusion: Base class pointers can only "see" base class normal member functions.

Inheritance

```
class Animal
{
public:
  void MakeSound() const { std::cout << "rawr\n"; }
};

class Cat : public Animal
{
public:
  void MakeSound() const { std::cout << "Miauw\n"; }
  void EnableNightVision(){}
};
int main()
{
  Animal* pAnimal{ new Cat{} };
  pAnimal->MakeSound();
  ~~pAnimal->EnableNightVision();~~
}
```

```
output:
rawr
```

Remember: a pointer:

- is not only a memory address
- also has a type!

Here, the memory address is the same, but the type is different:

A Cat pointer:

- ➢ Can access all the member functions of the Cat class, and the protected/public functions of the Animal class through inheritance.

An Animal pointer:

- ➢ Can only access normal member functions of the Animal class.

Inheritance

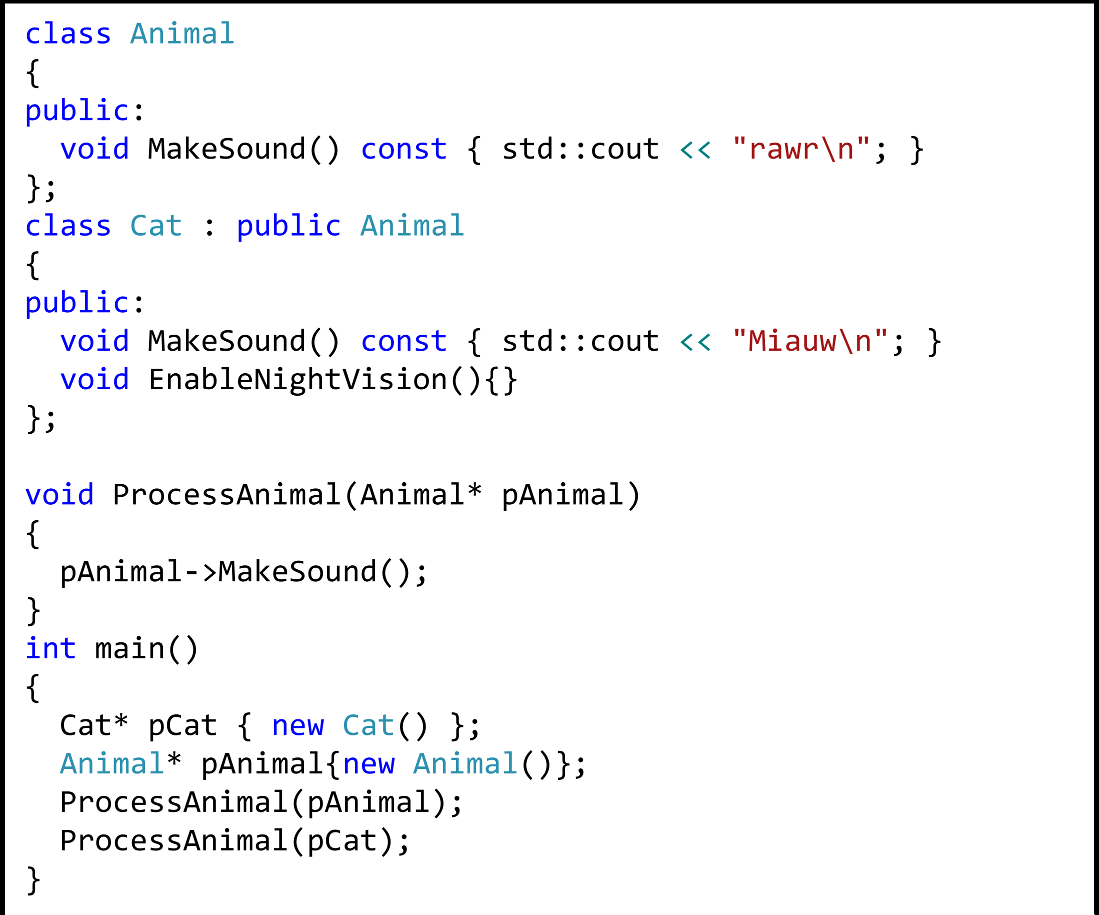
```cpp
class Animal
{
public:
  void MakeSound() const { std::cout << "rawr\n"; }
};
class Cat : public Animal
{
public:
  void MakeSound() const { std::cout << "Miauw\n"; }
  void EnableNightVision(){}
};

int main()
{
  std::vector<Animal*>animals;
  animals.push_back(new Animal());
  animals.push_back(new Cat());
  for (Animal* pAnimal:animals)
  {
    pAnimal->MakeSound();
  }
}
```

output fails:

```
rawr
rawr
```

- Question: Why would we use Base pointers or references?
  - Create a container filled with pointers to objects that are derived from Animal
  - Use a for loop to iterate over the container

Inheritance



```cpp
class Animal
{
public:
  void MakeSound() const { std::cout << "rawr\n"; }
};
class Cat : public Animal
{
public:
  void MakeSound() const { std::cout << "Miauw\n"; }
  void EnableNightVision(){}
};

void ProcessAnimal(Animal* pAnimal)
{
  pAnimal->MakeSound();
}
int main()
{
  Cat* pCat { new Cat() };
  Animal* pAnimal{new Animal()};
  ProcessAnimal(pAnimal);
  ProcessAnimal(pCat);
}
```

```
output fails:
rawr
rawr
```

- Question: Why would we use Base pointers or references?
  - A function that has a base class pointer or base class reference type as parameter.

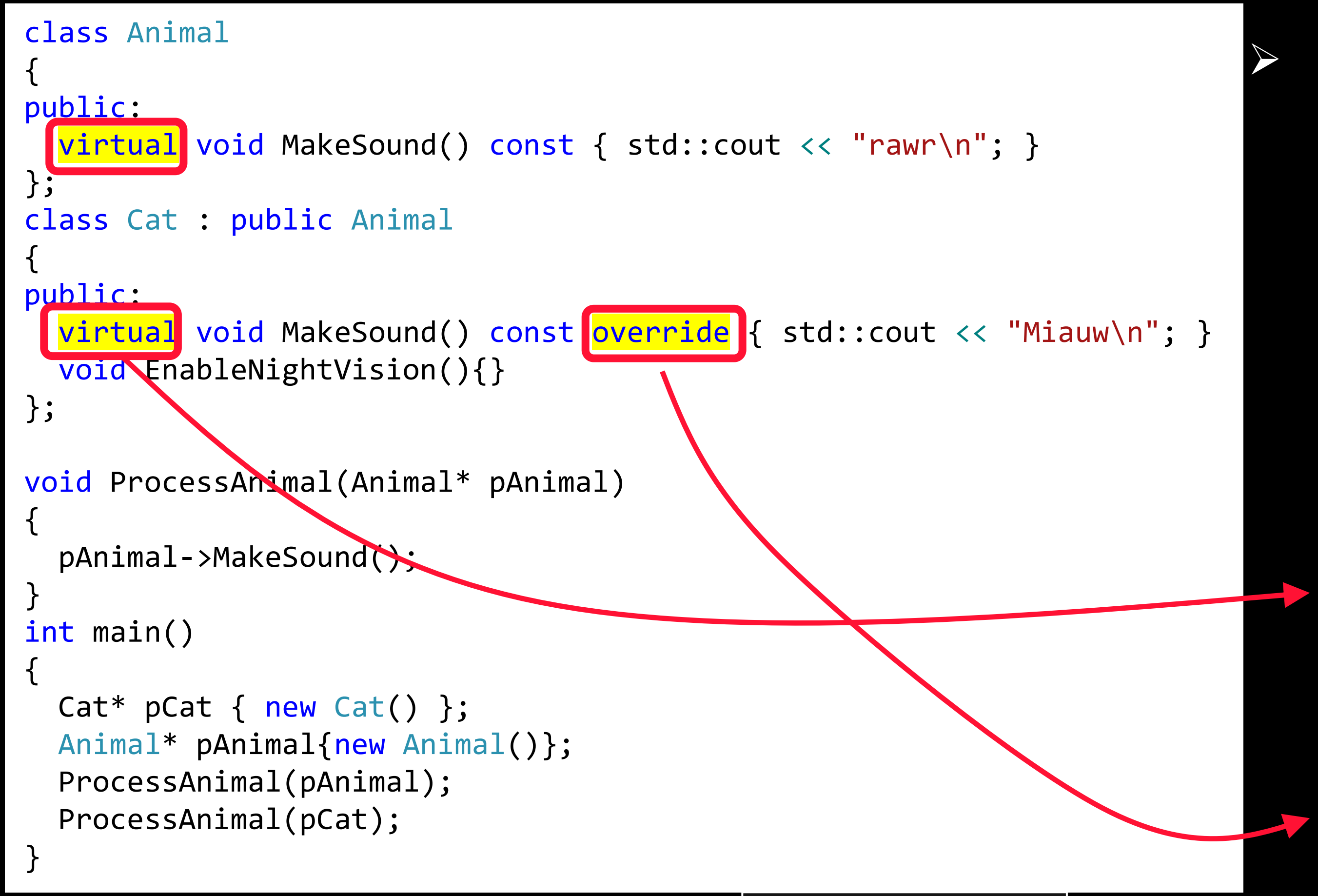
Polymorphism

```
class Animal
{
public:
  virtual void MakeSound() const { std::cout << "rawr\n"; }
};
class Cat : public Animal
{
public:
  virtual void MakeSound() const override { std::cout << "Miauw\n"; }
  void EnableNightVision(){}
};

void ProcessAnimal(Animal* pAnimal)
{
  pAnimal->MakeSound();
}
int main()
{
  Cat* pCat { new Cat() };
  Animal* pAnimal{new Animal()};
  ProcessAnimal(pAnimal);
  ProcessAnimal(pCat);
}
```

```
output success!
rawr
Miauw
```

- **Solution: virtual functions**
  - A virtual function call resolves to the most-derived version of the function that exists between the base and derived class.
  - A virtual function is "transparent".
  - This capability is known as polymorphism:
    - Depending on the type of the object the pointer refers to, a different member function is called.

Polymorphism

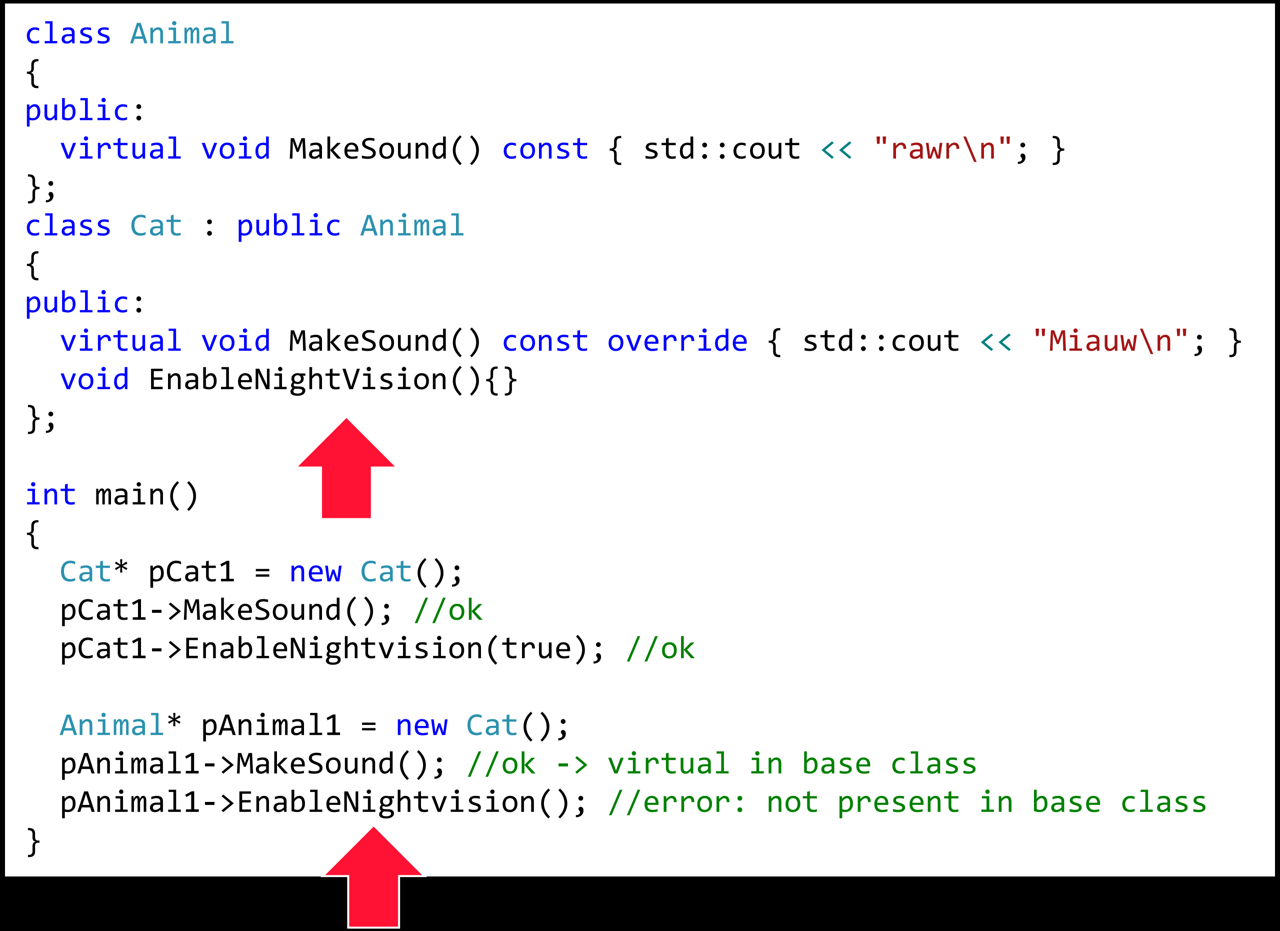


```
class Animal
{
public:
  virtual void MakeSound() const { std::cout << "rawr\n"; }
};
class Cat : public Animal
{
public:
  virtual void MakeSound() const override { std::cout << "Miauw\n"; }
  void EnableNightVision(){}
};

void ProcessAnimal(Animal* pAnimal)
{
  pAnimal->MakeSound();
}
int main()
{
  Cat* pCat { new Cat() };
  Animal* pAnimal{new Animal()};
  ProcessAnimal(pAnimal);
  ProcessAnimal(pCat);
}
```

```
output success!
rawr
Miauw
```

- **Solution: virtual functions**
  - A virtual function call resolves to the most-derived version of the function that exists between the base and derived class.
  - A virtual function is "transparent".
  - This capability is known as polymorphism:
    - Depending on the type of the object the pointer refers to, a different member function is called.
  - If a base class function is virtual, the derived function is automatically virtual too. Still mark it as virtual, so that is visible.
  - The overriding function must be marked with the “override” specifier. This checks if you are overriding a function.

Polymorphism

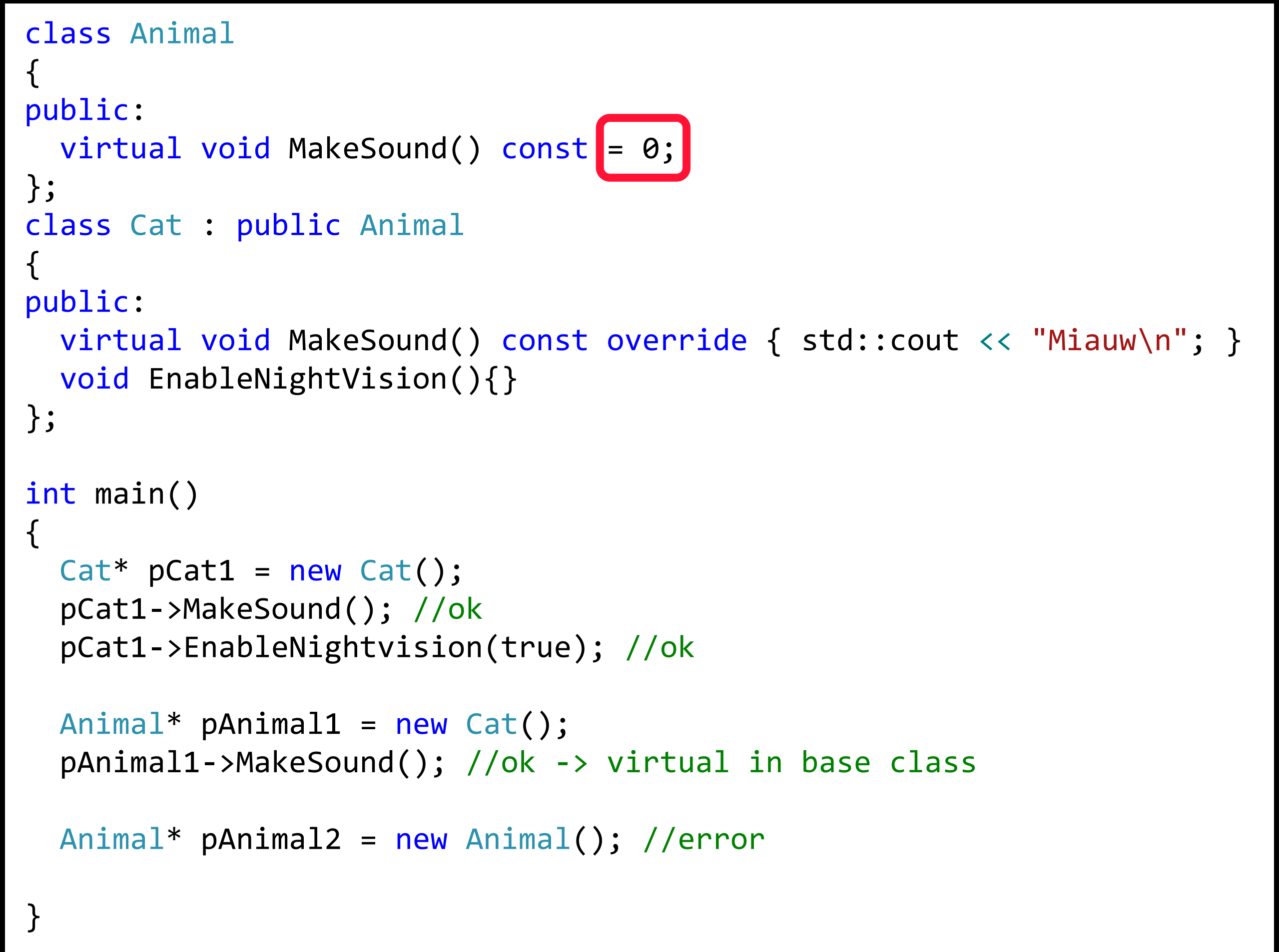


```
class Animal
{
public:
  virtual void MakeSound() const { std::cout << "rawr\n"; }
};
class Cat : public Animal
{
public:
  virtual void MakeSound() const override { std::cout << "Miauw\n"; }
  void EnableNightVision(){}
};

int main()
{
  Cat* pCat1 = new Cat();
  pCat1->MakeSound(); //ok
  pCat1->EnableNightvision(true); //ok

  Animal* pAnimal1 = new Cat();
  pAnimal1->MakeSound(); //ok -> virtual in base class
  pAnimal1->EnableNightvision(); //error: not present in base class
}
```

Using a base class pointer:

- Limitation:
  - It is not possible to call a function of a derived object through a base class reference or pointer if the function is not present in the base class.

Polymorphism: pure virtual



```
class Animal
{
public:
  virtual void MakeSound() const = 0;
};
class Cat : public Animal
{
public:
  virtual void MakeSound() const override { std::cout << "Miauw\n"; }
  void EnableNightVision(){}
};

int main()
{
  Cat* pCat1 = new Cat();
  pCat1->MakeSound(); //ok
  pCat1->EnableNightvision(true); //ok

  Animal* pAnimal1 = new Cat();
  pAnimal1->MakeSound(); //ok -> virtual in base class

  Animal* pAnimal2 = new Animal(); //error

}
```

What if a virtual base class function makes no sense or is not needed?

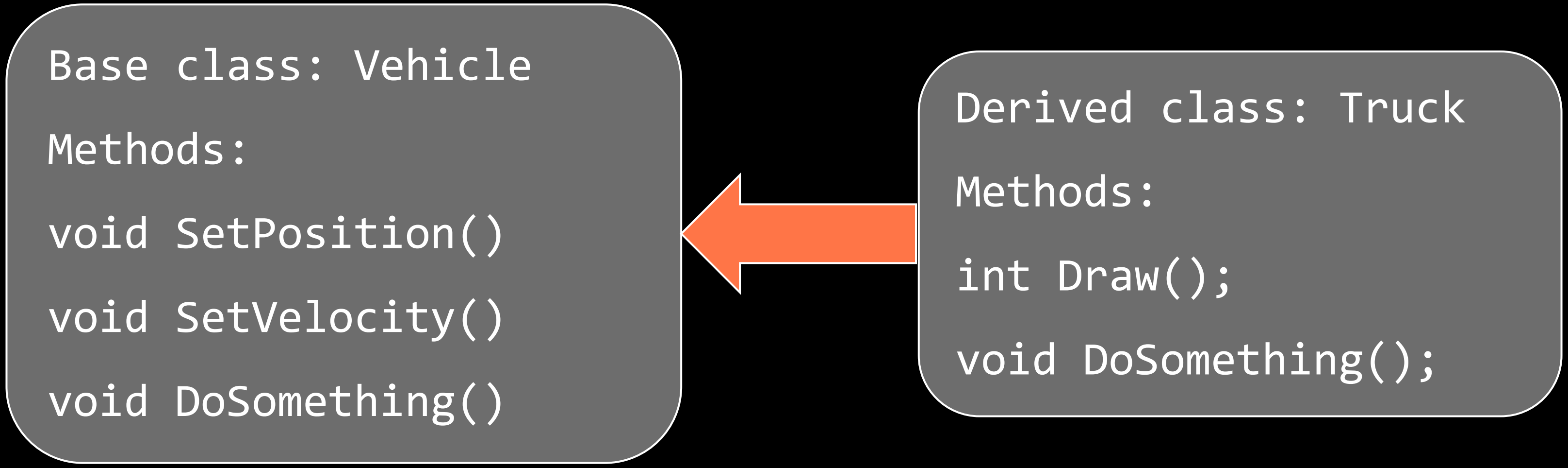
- In this case an animal makes no sound!
- Add the pure specifier to the function.
  - A pure virtual function has no definition.
  - The class is called an abstract class.
  - Creating objects of an abstract class is not possible.
  - Derived classes MUST override and define pure virtual functions (when creating instances of it) .

Polymorphism

The keyword *virtual* :

- You only write the keyword *virtual* in the header file
  - you write the method as usual in the .cpp file, so without the virtual keyword

Example



Base class: Vehicle

Methods:

void SetPosition()

void SetVelocity()

void DoSomething()

Derived class: Truck

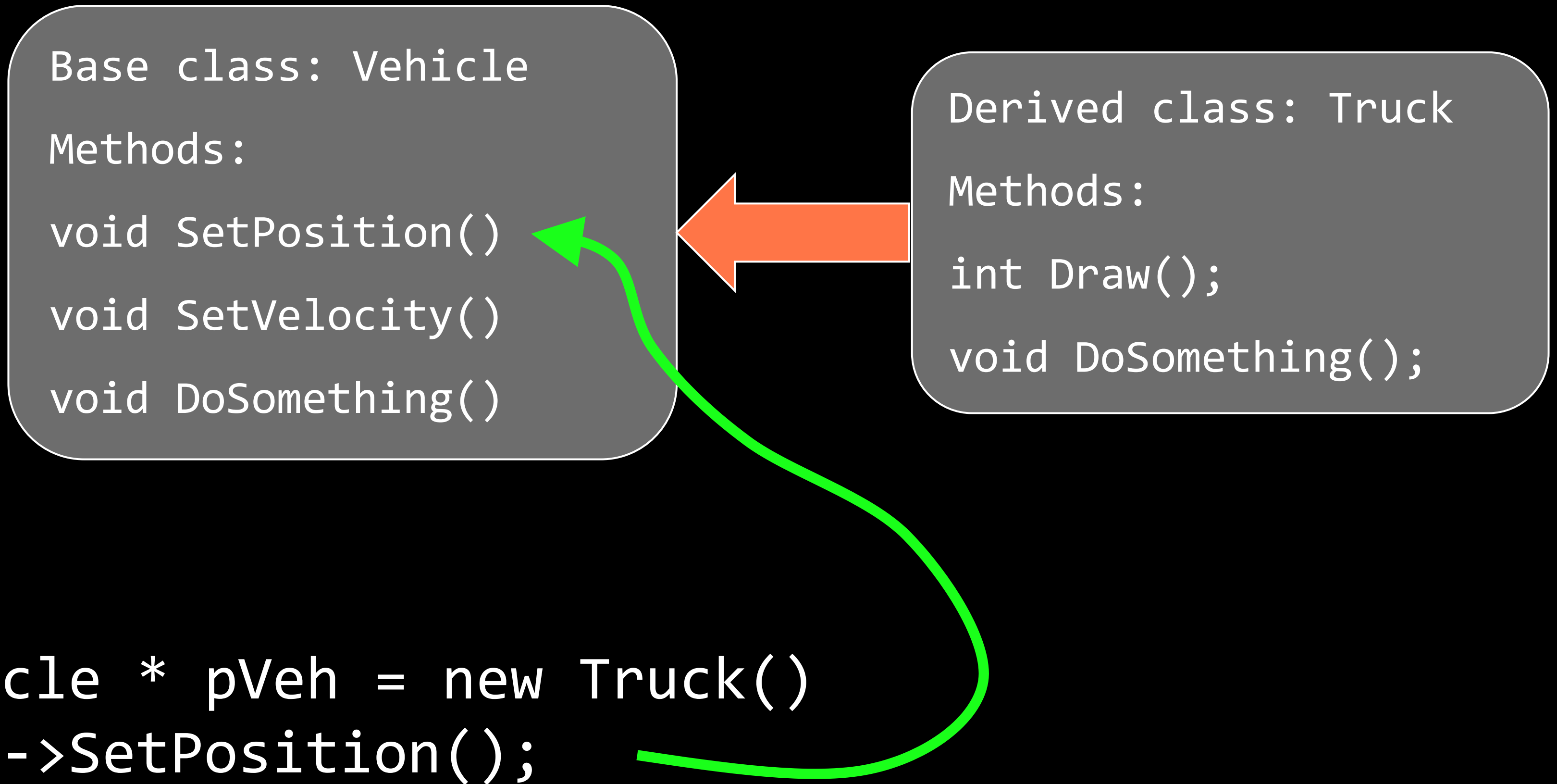
Methods:

int Draw();

void DoSomething();

```
Vehicle * pVeh = new Truck()
pVeh->SetPosition();
pVeh->DoSomething();
pVeh->Draw();
```

Example



Base class: Vehicle

Methods:

void SetPosition()

void SetVelocity()

void DoSomething()

Derived class: Truck

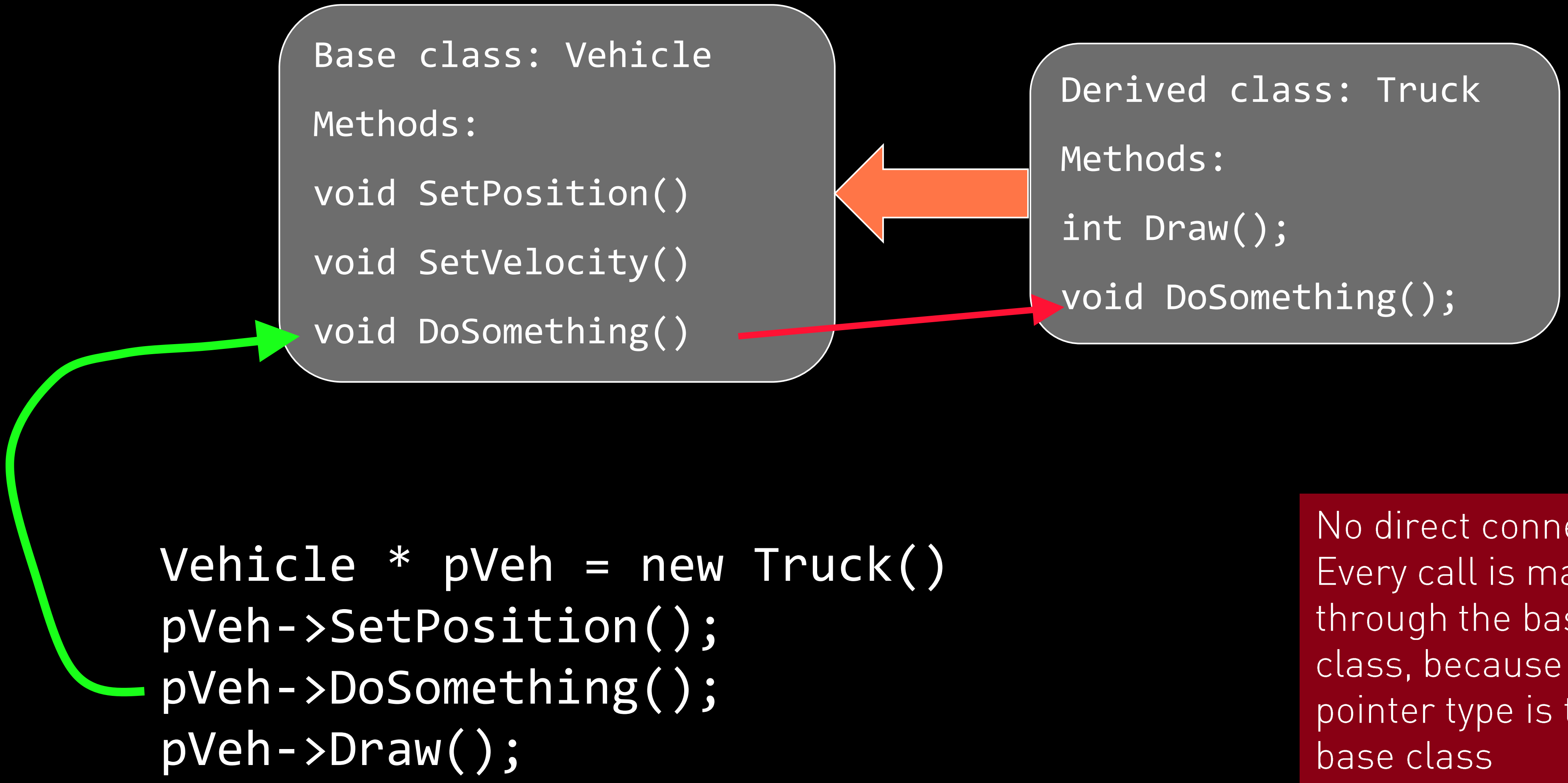
Methods:

int Draw();

void DoSomething();

```
Vehicle * pVeh = new Truck()
pVeh->SetPosition();
pVeh->DoSomething();
pVeh->Draw();
```

Example



Base class: Vehicle

Methods:

void SetPosition()

void SetVelocity()

void DoSomething()

Derived class: Truck

Methods:

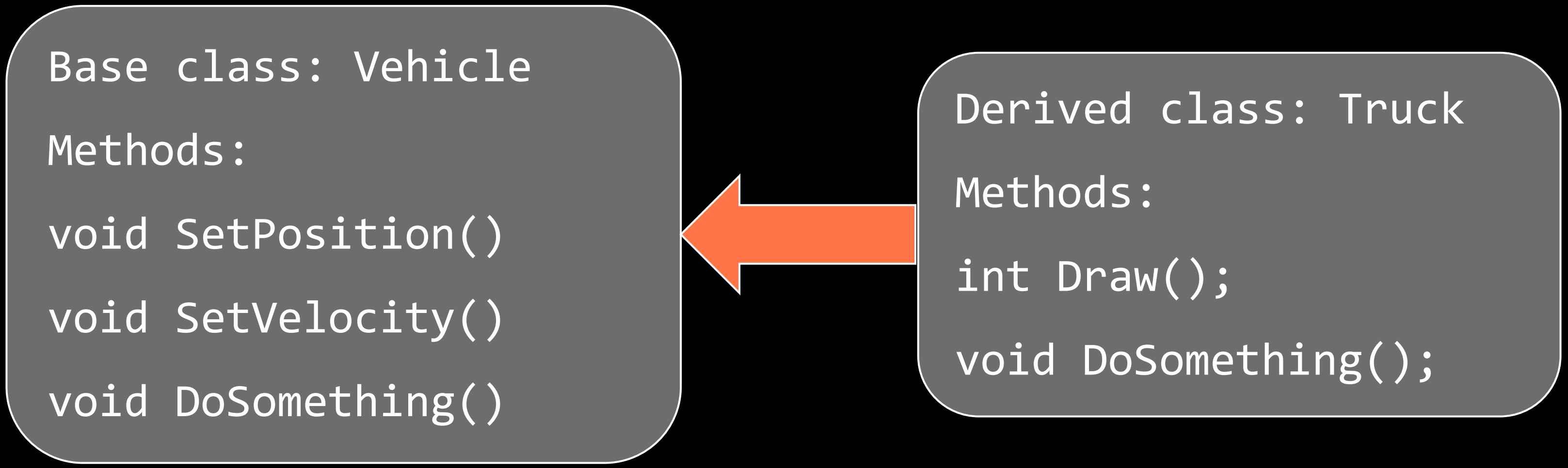
int Draw();

void DoSomething();

```
Vehicle * pVeh = new Truck()
pVeh->SetPosition();
pVeh->DoSomething();
pVeh->Draw();
```

No direct connection. Every call is made through the base class, because the pointer type is the base class

Example



Base class: Vehicle

Methods:

```
void SetPosition()
void SetVelocity()
void DoSomething()
```

Derived class: Truck

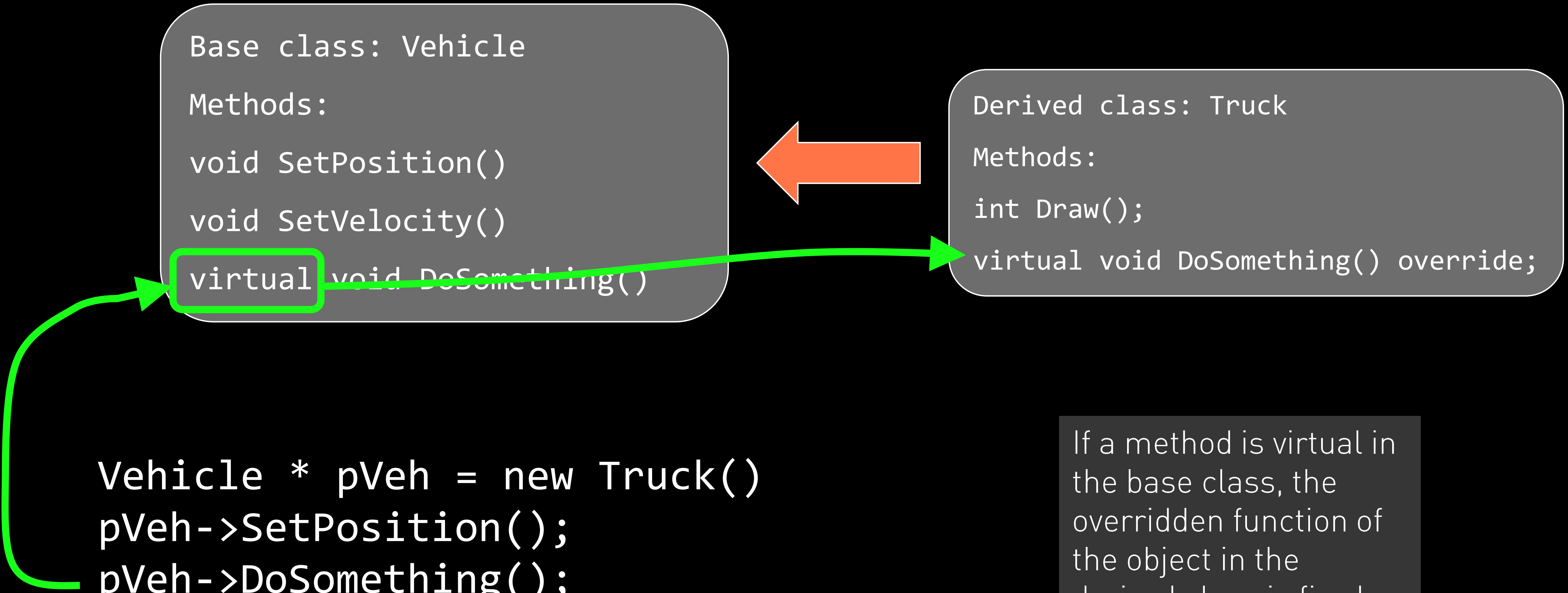
Methods:

```
int Draw();
void DoSomething();
```

```
Vehicle * pVeh = new Truck()
pVeh->SetPosition();
pVeh->DoSomething();
pVeh->Draw();
```

Error: There is no Draw method in the pointer type class (Vehicle).

Example



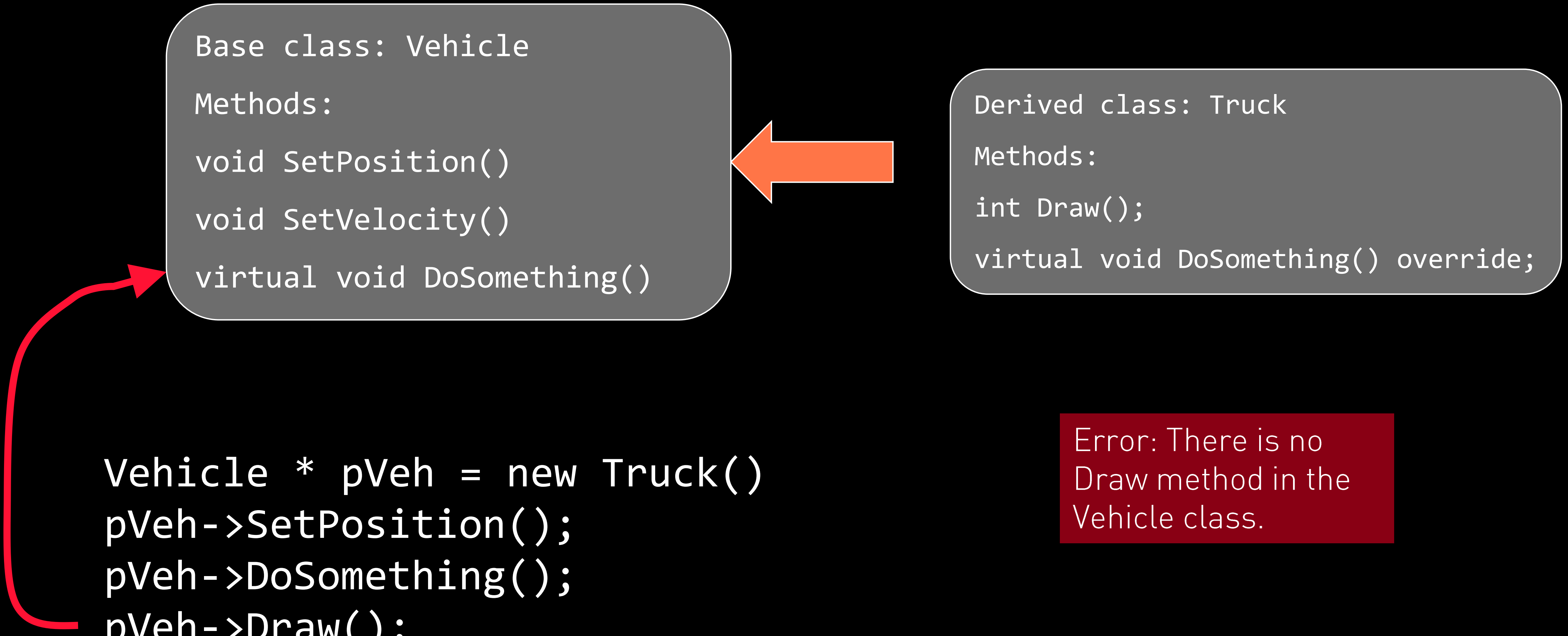
```
Base class: Vehicle

Methods:

void SetPosition()

void SetVelocity()

virtual void DoSomething()
```

```
Derived class: Truck

Methods:

int Draw();

virtual void DoSomething() override;
```

```
Vehicle * pVeh = new Truck()
pVeh->SetPosition();
pVeh->DoSomething();
pVeh->Draw();
```

If a method is virtual in the base class, the overridden function of the object in the derived class is fired.

How do we fix this problem?



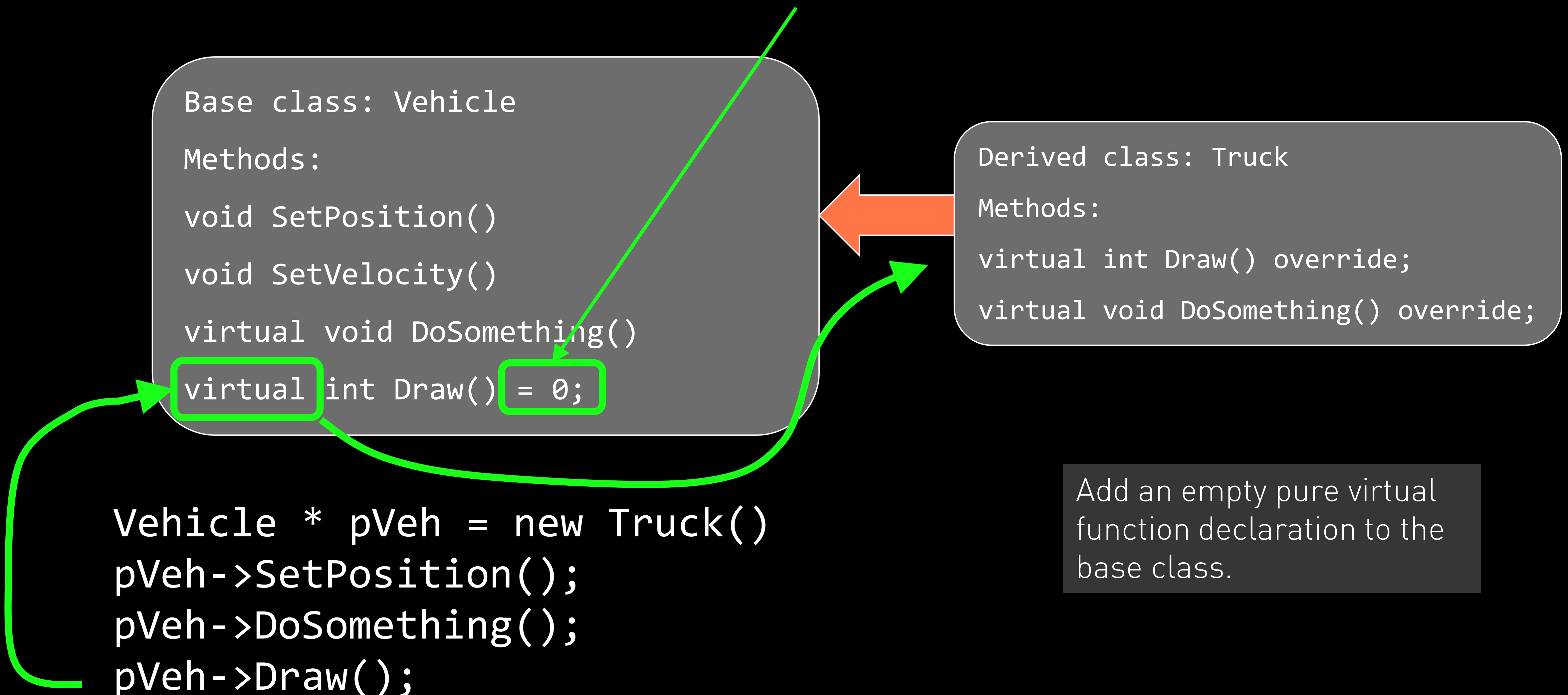
```
Base class: Vehicle
Methods:
void SetPosition()
void SetVelocity()
virtual void DoSomething()
```

```
Derived class: Truck
Methods:
int Draw();
virtual void DoSomething() override;
```

```
Vehicle * pVeh = new Truck()
pVeh->SetPosition();
pVeh->DoSomething();
pVeh->Draw();
```

Error: There is no Draw method in the Vehicle class.

Pure Virtual



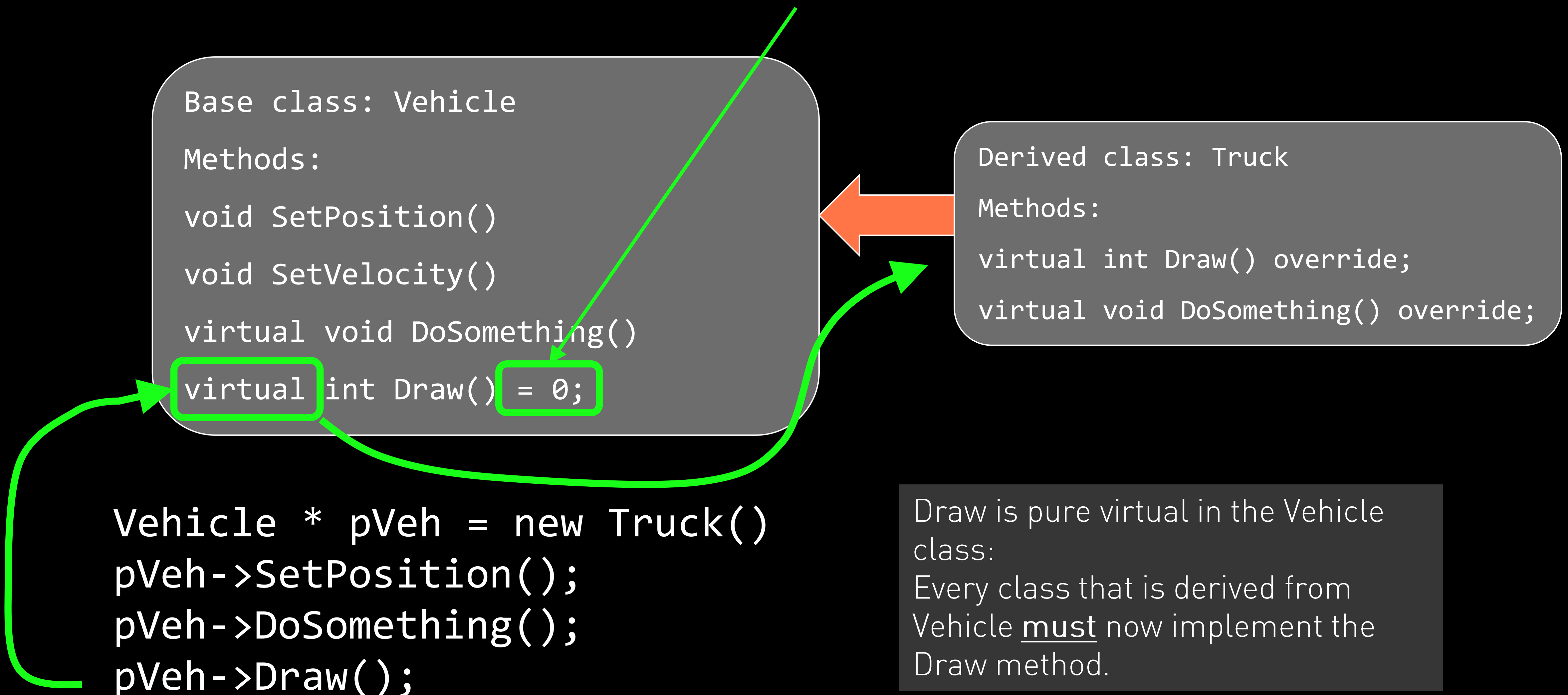
```
Base class: Vehicle
Methods:
void SetPosition()
void SetVelocity()
virtual void DoSomething()
virtual int Draw() = 0;
```

```
Derived class: Truck
Methods:
virtual int Draw() override;
virtual void DoSomething() override;
```

```
Vehicle * pVeh = new Truck()
pVeh->SetPosition();
pVeh->DoSomething();
pVeh->Draw();
```

Add an empty pure virtual function declaration to the base class.

Pure Virtual



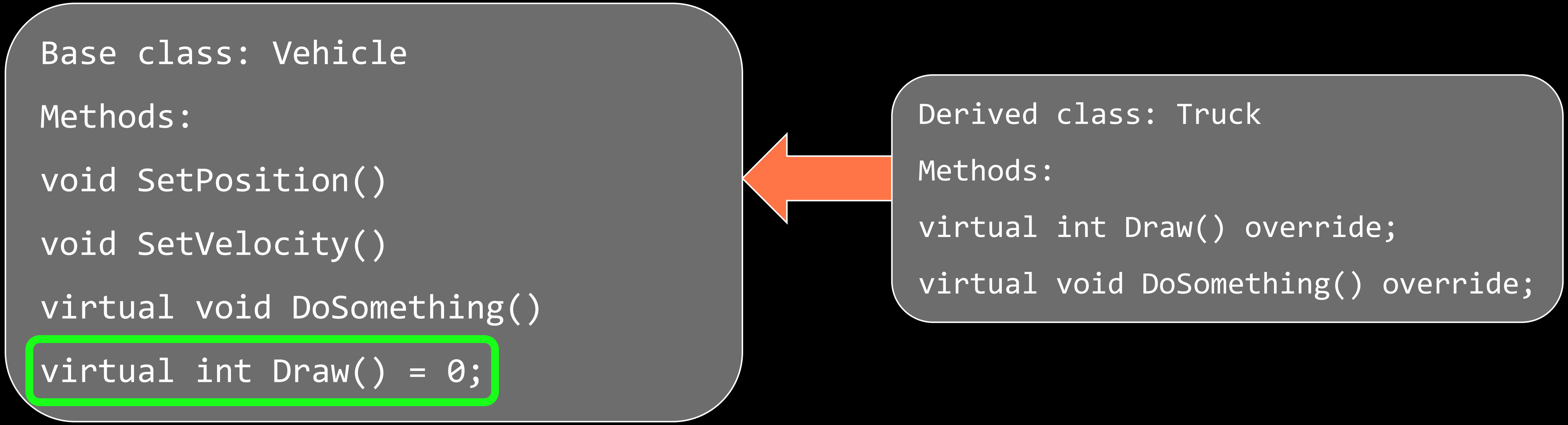
```
Base class: Vehicle
Methods:
void SetPosition()
void SetVelocity()
virtual void DoSomething()
virtual int Draw() = 0;
```

```
Derived class: Truck
Methods:
virtual int Draw() override;
virtual void DoSomething() override;
```

```
Vehicle * pVeh = new Truck()
pVeh->SetPosition();
pVeh->DoSomething();
pVeh->Draw();
```

Draw is pure virtual in the Vehicle class:
Every class that is derived from Vehicle <u>must</u> now implement the Draw method.

Pure Virtual



```
Base class: Vehicle
Methods:
void SetPosition()
void SetVelocity()
virtual void DoSomething()
virtual int Draw() = 0;
```

```
Derived class: Truck
Methods:
virtual int Draw() override;
virtual void DoSomething() override;
```

Because there is no definition for the Draw() function in Vehicle, C++ will be unable to make an object of this class. The Vehicle class is now called an abstract class.

If you try to write “new Vehicle()” anyway, you get the compile error:

```
"Vehicle" : cannot instantiate abstract class
```

Polymorphism

Pure Virtual

- By setting a method "pure virtual", we declared that the method is not given a definition. In other words, we only say that the method is there (this way it is inherited). We don't say what the method does.

- Pure virtual methods are only written in the header file. You are not allowed to write them out in the .cpp file (this would be a definition).

- Every class that is derived form a base class that has pure virtual methods, MUST implement these methods.

- It is not possible to make objects from a class that has pure virtual methods.

Abstract class

- Defines an abstract type which cannot be instantiated but can be used as a base class.
- It defines or inherits at least one function for which the final overrider is pure virtual.
  - pure-specifier: `= 0`
  - Example: `virtual void function() = 0;`
- UML: *italics*

Summary

- Inheritance
  - Declaration without using the keyword "virtual": overriding function
  - the type of the pointer determines what overridden method definition will be used.

- Polymorphism:
  - declaration with the keyword "virtual": virtual function
  - the type of the object determines what overridden method definition will be used.
  - with the keyword "virtual" and no definition (=0) : pure virtual function
    - the class becomes an *abstract class*
    - it is not possible to create an instance of this class
  - Base class pointers pointing at a derived object

Destructors must always be virtual.

Why do we need a virtual destructor?

- Non-virtual base class destructor:
  - If we delete a derived object using a base pointer:
    - only the destructor of the base class is called. Causing possible memory leaks.

- Virtual base class destructor:
  - If we delete a derived object using a base pointer:
    - first the derived destructor is called
    - after that, the base class destructor is called

Destructors must always be virtual.

- If at least one function is virtual, then the destructor MUST be virtual.
- If the destructor is not virtual, then it should not be possible to inherit from the class. (how? “final”see last slide)
- If no destructor is implemented, the generated destructor is NOT virtual (!).

Conclusion: If at least one function is virtual, you must define a virtual destructor.

Find the bug (no compiler error)

```
class Animal
{
public:
  virtual void MakeSound() { std::cout << "rawr\n"; }
};

class Cat : public Animal
{
public:
  virtual void MakeSoumd() { std::cout << "Miaauw\n"; }
};

class Dog : public Animal
{
public:
  virtual void MakeSound() { std::cout << "Bark\n"; }
};
```

```
int main()
{

  Animal* pAnimal = new Cat();
  pAnimal->MakeSound();
  delete pAnimal;


  std::cin.get();
  return 0;
}
```

output:

```
rawr
```

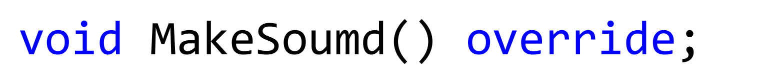
Find the error

```cpp
class Animal
{
public:
  virtual void MakeSound() { std::cout << "rawr\n"; }
};

class Cat : public Animal
{
public:
  virtual void MakeSoumd() { std::cout << "Miaauw\n"; }
};

class Dog : public Animal
{
public:
  virtual void MakeSound() { std::cout << "Bark\n"; }
};
```

```cpp
int main()
{

  Animal* pAnimal = new Cat();
  pAnimal->MakeSound();
  delete pAnimal;


  std::cin.get();
  return 0;
}
```

Cat::MakeSoumd() is not overriding Animal::MakeSound() --> typo?

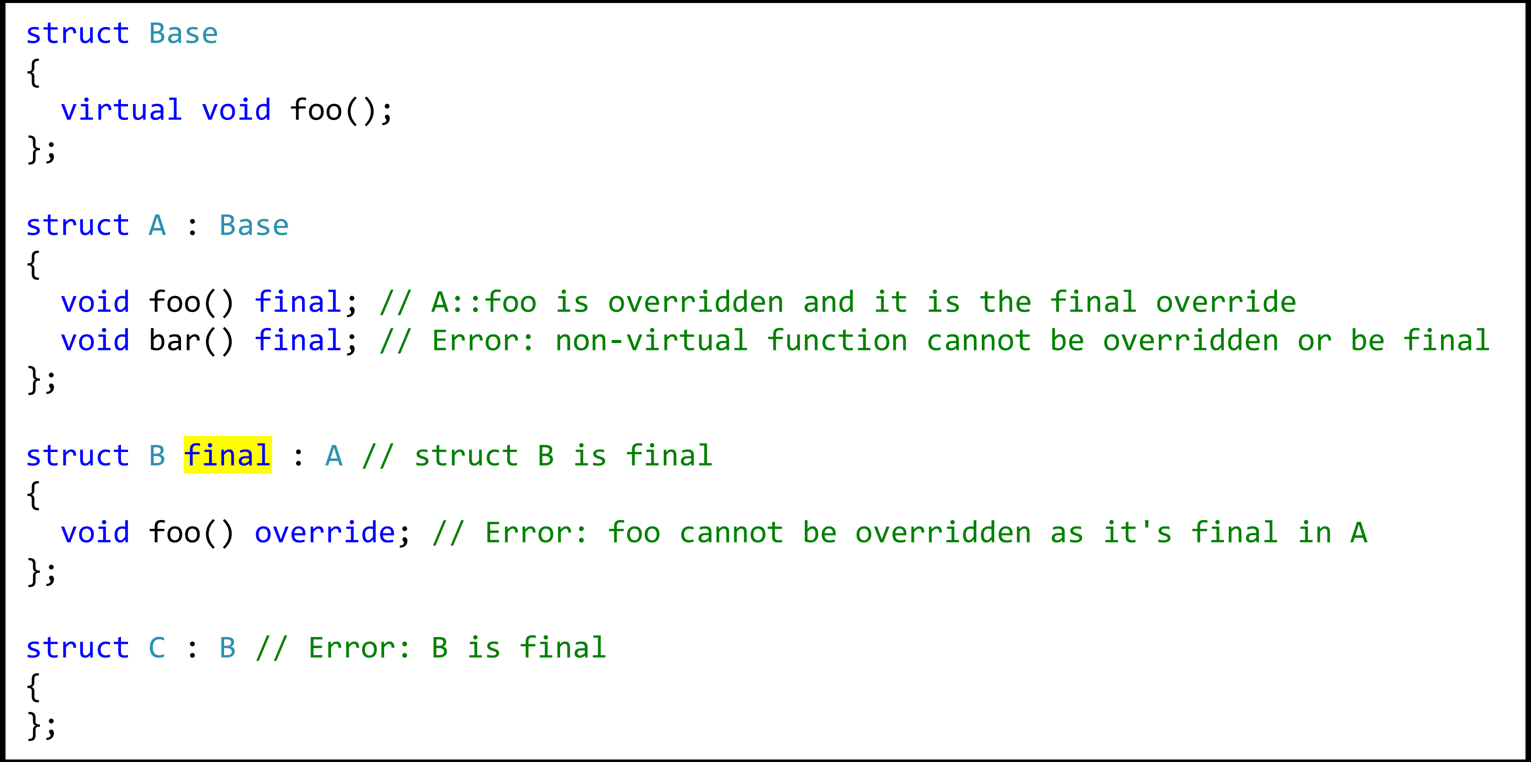
Override specifier



```
void MakeSoumd() override;
```

- Specifies that a virtual function overrides another virtual function.
- In a member function declaration or definition, override ensures that the function is virtual and is overriding a virtual function from the base class.
- The program is ill-formed (a compile-time error is generated) if this is not true.
- Always use this override specifier!

Function final specifier



```
struct Base
{
  virtual void foo();
};

struct A : Base
{
  void foo() final; // A::foo is overridden and it is the final override
  void bar() final; // Error: non-virtual function cannot be overridden or be final
};

struct B final : A // struct B is final
{
  void foo() override; // Error: foo cannot be overridden as it's final in A
};

struct C : B // Error: B is final
{
};
```

- Specifies that a virtual function cannot be overridden in a derived class.
- e.g. When a destructor is not virtual.

Class final specifier

```
class Sprite() final
{
  Sprite();
  ~Sprite();
}
```

or

```
class Sprite()
{
  Sprite();
  virtual ~Sprite();
}
```

- Specifies that the class can not be derived from.
- Example: to avoid having to use a virtual destructor, make the class final.

- Why? Having a virtual function introduces a virtual function table, calling functions wit that table takes a bit longer.

Class final specifier

```
class Sprite() final
{
  Sprite();
  ~Sprite();
}
```

or

```
class Sprite()
{
  Sprite();
  virtual ~Sprite();
}
```

- If a destructor is not virtual, the class must be marked as final!

Reference

https://www.learncpp.com/cpp-tutorial/virtual-functions/