CS-202

C++ Classes – Polymorphism (Pt.2)

C. Papachristos

Autonomous Robots Lab
University of Nevada, Reno

Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	Friday		Sunday
			Lab (8 Sections)			
	CLASS		CLASS			
PASS Session	PASS Session	**Project DEADLINE**	NEW Project	PASS Session		PASS Session

Your 6th Project will be announced today Thursday 3/7.

5th Project Deadline was this Wednesday 3/6.
➢ NO Project accepted past the 24-hrs delayed extension (@ 20% grade penalty).
➢ Send what you have in time!

Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	Friday		Sunday
			Lab (8 Sections)			
	CLASS	MIDTERM Sample	CLASS			
PASS Session	PASS Session	**Project DEADLINE**	NEW Project	PASS Session		**MIDTERM Review PASS**

!

Your Midterm will be held on Thursday 3/14.

A Midterm Sample has already been announced.
➢ Following Lectures, Labs, and PASS sessions will be dedicated to recapitulation.
➢ Extra 2-hr recap PASS session arranged for this upcoming Sunday.

Today's Topics

Static *vs* Dynamic Binding

➢ Concepts & Practice

Abstract Classes

➢ Pure `virtual` Class Methods (*vs* regular "`virtual`" ones)

Polymorphism (advanced)

➢ `override` , `final` Specifiers
➢ Aggregation – the `clone()` Method
➢ `virtual` Function `return` Types – Covariant Types
➢ `virtual` Destructor(s) – Dynamic Binding considerations

Virtual Function Tables

Polymorphism

Remember: **Static** *vs* **Dynamic Binding**

What is the difference between Method Overriding possibilities?
➢ Determining which Method in Hierarchy to call.

Static Binding (usual same-signature overriding)
Compiler –at "compile-time"– determines binding.

Dynamic Binding (overriding with keyword `virtual`)
System –at "run-time"– determines binding.

Polymorphism

Remember: **Static** *vs* **Dynamic Binding**

What is the difference between Method Overriding possibilities?
➢ Determining which Method in Hierarchy to call.

Static Binding – at "compile-time" :
➢ Will resolve to what is "known" to the compiler and "certain".

Dynamic Binding – at "run-time" :
➢ Will attempt to resolve what is "`virtual`" and "uncertain".
➢ What can be dynamic (i.e. "compile-time uncertain") in Inheritance relationships?

Polymorphism

Remember: **Static** *vs* **Dynamic Binding**

Static Binding

```cpp
class Vehicle {
 public:
  void sound(){
    cout<<"Hummm..."<<endl;
  }
};
class F1Car : public Vehicle {
 public:
  void sound(){
    cout<<"Zooom!"<<endl;
  }
};
```

```cpp
int main(){
  Vehicle vehicle;
  F1Car f1car;

  vehicle.sound();          //Hummm...
  f1car.sound();            //Zooom!

  Vehicle *vehicle_Pt = &vehicle;
  vehicle_Pt->sound();      //Hummm...

  Vehicle *vehicleF1car_Pt = &f1car;
  vehicleF1car_Pt->sound(); //Hummm...

  return 0;
}
```

Objects

Object Pointers

Polymorphism

Remember: **Static** *vs* **Dynamic Binding**

Static Binding

```cpp
class Vehicle {
 public:
  void sound(){
   cout<<"Hummm..."<<endl;
  }
};

class F1Car : public Vehicle {
 public:
  void sound(){
   cout<<"Zooom!"<<endl;
  }
};
```

Objects

Object Pointers

```cpp
int main(){
  Vehicle vehicle;
  F1Car f1car;

  vehicle.sound();          //Hummm...
  f1car.sound();            //Zooom!

  Vehicle *vehicle_Pt = &vehicle;
  vehicle_Pt->sound();      //Hummm...

  Vehicle *vehicleF1car_Pt = &f1car;
  vehicleF1car_Pt->sound();  //Hummm...

  return 0;
}
```

Polymorphism

Remember: **Static *vs* Dynamic Binding**

Static Binding

```cpp
class Vehicle {
 public:
  void sound(){
    cout<<"Hummm..."<<endl;
  }
};
class F1Car : public Vehicle {
 public:
  void sound(){
    cout<<"Zooom!"<<endl;
  }
};
```

Objects

Object Pointers

```cpp
int main(){
    Vehicle vehicle;
    F1Car f1car;

    vehicle.sound();          //Hummm...
    f1car.sound();            //Zooom!

    Vehicle *vehicle_Pt = &vehicle;
    vehicle_Pt->sound();      //Hummm...

    Vehicle *vehicleF1car_Pt = &f1car;
    vehicleF1car_Pt->sound(); //Hummm...

    return 0;
}
```

Static Binding

Polymorphism

Remember: **Static *vs* Dynamic Binding**

Dynamic Binding

```cpp
class Vehicle {
 public:
  virtual void sound();
};
void Vehicle::sound(){
  cout<<"Hummm..."<<endl;
}

class F1Car : public Vehicle {
 public:
  virtual void sound();
};
void F1Car::sound(){
  cout<<"Zooom! "<<endl;
}
```

Objects

Object Pointers

```cpp
int main(){
  Vehicle vehicle;
  F1Car f1car;

  vehicle.sound();              //Hummm...
  f1car.sound();                //Zooom!

  Vehicle *vehicle_Pt = &vehicle;
  vehicle_Pt->sound();          //Hummm...

  Vehicle *vehicleF1car_Pt = &f1car;
  vehicleF1car_Pt->sound();   //Zooom!

  return 0;
}
```

Polymorphism

Remember: **Static** *vs* **Dynamic Binding**

Dynamic Binding

```cpp
class Vehicle {
 public:
  virtual void sound();
};
void Vehicle::sound(){
  cout<<"Hummm..."<<endl;
}

class F1Car : public Vehicle {
 public:
  virtual void sound();
};
void F1Car::sound(){
  cout<<"Zooom! "<<endl;
}
```

Objects

Object Pointers

```cpp
int main(){
  Vehicle vehicle;
  F1Car f1car;

  vehicle.sound();            //Hummm...
  f1car.sound();              //Zooom!

  Vehicle *vehicle_Pt = &vehicle;
  vehicle_Pt->sound();        //Hummm...

  Vehicle *vehicleF1car_Pt = &f1car;
  vehicleF1car_Pt->sound();   //Zooom!

  return 0;
}
```

Dynamic Binding

Polymorphism

Remember: **Static** *vs* **Dynamic Binding**

How to tell the difference between Method Overriding possibilities?
➢ Polymorphism: Ability to dynamically decide which Method to call.

Static Binding – at "compile-time" :
➢ Will directly resolve non-**virtual** Methods and Object-based calls.

Dynamic Binding – at "run-time" :
➢ Base Class Pointer – Derived Class Object.
➢ Keyword **virtual**.
➢ Method call on Base Class Pointer.
   at "run-time" will:
➢ Run Method of the Derived Class.

```
F1Car f1car;
Vehicle * vehicleF1car_Pt = &f1car;
```

```
virtual void sound();
```

```
vehicleF1car_Pt ->sound();
```

Remember: **Abstraction Concepts**

The Abstract Type:
(Not to be confused with general Programming Abstraction)

➢ A programming language-related implementation.

➢ Given a type system, an Abstract Type is one that cannot be *instantiated* directly (vs a Concrete Type).

`<abstract_type> Vehicle ;`   `<concrete_type> Vehicle : Car ;`

virtual Functions & Classes

Enabling Polymorphic behaviors.

```cpp
class Vehicle {
  public:
    virtual void drive();
  private:
    double m_speed;
};
void Vehicle::drive(){ /*base implementation …*/ }


class Car : public Vehicle {
 public:
  virtual void drive();
 private:
  double m_steerAngle;
  double m_throttlePos;
};
void Car::drive(){ /*derived implementation …*/ }
```

Parent class is *required* to have own Implementation
➢ Even if it's trivial or empty

virtual Functions & Classes

Enabling Polymorphic behaviors.

```cpp
class Vehicle {
  public:
    virtual void drive();
  private:
    double m_speed;
};
void Vehicle::drive(){ /*base implementation …*/ }
```

> Parent class is *required* to have own Implementation
> ➢ Even if it's trivial or empty

```cpp
class Car : public Vehicle {
  public:
    virtual void drive();
  private:
    double m_steerAngle;
    double m_throttlePos;
};
void Car::drive(){ /*derived implementation …*/ }
```

> Child Classes *may* Override if they choose to.
> ➢ If not overridden, Parent Class definition used.

Pure `virtual` Functions & Classes

Enabling purely *Abstract* Classes.

```cpp
class Vehicle {
  public:
    virtual void drive() = 0;
  private:
    double m_speed;
};
void Vehicle::drive(){ /*base implementation …*/ }


class Car : public Vehicle {
 public:
  virtual void drive();
 private:
  double m_steerAngle;
  double m_throttlePos;
};
void Car::drive(){ /*derived implementation …*/ }
```

Parent class is *not required* to have own Implementation
- ➤ Can provide their own, but don't have to.
- ➤ Considered an *Abstract* Class in either case.

Pure `virtual` Functions & Classes

Enabling purely *Abstract* Classes.

```cpp
class Vehicle {
  public:
    virtual void drive() = 0;
  private:
    double m_speed;
};
void Vehicle::drive(){ /*base implementation …*/ }
```

Parent class is *not required* to have own Implementation
- Can provide their own, but don't have to.
- Considered an *Abstract* Class in either case.

```cpp
class Car : public Vehicle {
  public:
    virtual void drive();
  private:
    double m_steerAngle;
    double m_throttlePos;
};
void Car::drive(){ /*derived implementation …*/ }
```

Child Classes are *required to* provide own Implementations.
- If not overridden, Parent Class definition used.

Pure **virtual** Specifier (**= 0 ;**)

Declaration of Pure virtual Method:
➤ At the same time, the Class becomes an Abstract Class.

```cpp
class BaseClass {
  public:
    virtual void pureVirtualMethod() = 0;
    …
};
void BaseClass::pureVirtualMethod(){ /*potential base implementation …*/ }
```

Parent class is *not required* to
have own Implementation
(i.e. can have one)

Pure `virtual` Specifier (`= 0;`)

Declaration of Pure virtual Method:

➢ At the same time, the Class becomes an Abstract Class.

```cpp
class BaseClass {
  public:
    virtual void pureVirtualMethod() = 0;

    …
};
void BaseClass::pureVirtualMethod(){ /*potential base implementation …*/ }
```

Abstract Class Objects:

➢ Cannot exist (Class is Abstract, an Object is not concrete) !

➢ Not allowed to be instantiated:

```cpp
BaseClass base;
```

```
error:cannot declare variable 'base' to be of
 abstract type 'BaseClass'
note:because the following virtual functions
are pure within 'BaseClass': …
```

Abstract Class (more specifically)

A Class that defines or inherits at least one function for which the final Overrider is Pure `virtual`.
➢ Used to represent general concepts, which act as Base Classes for concrete Classes.

No(s):
➢ No Abstract Class Objects can be created.
➢ Abstract types cannot be used as parameter types, as function return types, or as the type of an explicit conversion.

Yes(s):
➢ Pointers and References to an Abstract Class can be declared and used.

Abstract Classes

override Specifier (C++11)

Ensures (at compile time) that the Function is **virtual** and is Overriding a **virtual** Function from the Base Class.

```cpp
class BaseClass {
  public:
   virtual void virtualMethod();
   void regularMethod();
    …
};
class DerivedClass : pubic BaseClass{
  public:
   void virtualMethod() override ;
```

Signature match to **virtual** Base Class Function

```cpp
    …
};
```

override Specifier (C++11)

Ensures (at compile time) that the Function is **virtual** and is Overriding a **virtual** Function from the Base Class.

```cpp
class BaseClass {
  public:
   virtual void virtualMethod();
   void regularMethod();
    …
};
class DerivedClass : pubic BaseClass{
  public:
   void virtualMethod() override ;
   void virtualMethod() const override ;
   void regularMethod() override ;
    …
};
```

Signature match to **virtual** Base Class Function

Signature mismatch to **virtual** Base Class Function

Not overriding a **virtual** Base Class Function

Result: Compilation errors

final Specifier (C++11)

A) Ensures that the **virtual** Function cannot be Overridden in a Derived Class.

```cpp
class BaseClass {
  public:
   virtual void virtualMethod();

   …
};
class DerivedClass : public BaseClass{
  public:
   void virtualMethod() override final ;          End-of-the-line for virtual Function Overriding

   …
};
class ReDerivedClass : public DerivedClass{
  public:
   void virtualMethod() override ;

   …
};
```

End-of-the-line for **virtual** Function Overriding

error: overriding final function
'virtual void DerivedClass::virtualMethod()'

`final` Specifier (C++11)

B) Ensures that a Class cannot be Inherited from (cannot be a Base Class).

```cpp
class BaseClass {
  public:
   virtual void virtualMethod();
    …
};
class DerivedClass final : public BaseClass{
  public:
   void virtualMethod() override;
    …
};
class ReDerivedClass : public DerivedClass{
  public:
    …
};
```

End-of-the-line for Class Inheritance

`error:` error: cannot derive from 'final' base 'DerivedClass' in derived type 'ReDerivedClass'

Polymorphism

Overview (by-Example) :

```
Car myCar;
Vehicle * vehicle_Pt = &myCar;
vehicle_Pt -> drive();
```

Prototype	*Vehicle* Class	*Car* Class
void drive();	**Can** implement Function **Can** create *Vehicle* Object	**Can** implement Function **Can** create *Car* Object **Calls** Method *Vehicle::drive();*

Polymorphism

Overview (by-Example) :

```
Car myCar;
Vehicle * vehicle_Pt = &myCar;
vehicle_Pt -> drive();
```

Prototype	*Vehicle* Class	*Car* Class
`void drive();`	Can implement Function Can create *Vehicle* Object	Can implement Function Can create *Car* Object Calls Method *Vehicle::drive();*
`virtual void drive();`	Can implement Function Can create *Vehicle* Object	Can implement Function Can create *Car* Object Calls Method *Car::drive();*

Note: If no `Car::drive()` implemented, calls `Vehicle::drive()`.

Polymorphism

Overview (by-Example) :

```
Car myCar;
Vehicle * vehicle_Pt = &myCar;
vehicle_Pt -> drive();
```

Prototype	Vehicle Class	Car Class
`void drive();`	Can implement Function Can create Vehicle Object	Can implement Function Can create Car Object Calls Method Vehicle::drive();
`virtual void drive();`	Can implement Function Can create Vehicle Object	Can implement Function Can create Car Object Calls Method Car::drive();
`virtual void drive()=0;`	Could implement Function Cannot create Vehicle Object	Must implement Function Can create Car Object Calls Method Car::drive();

Note: This is a Pure `virtual` Function, and `Vehicle` is now an Abstract Class.

Overview (by-Example) :

```
Car myCar;
Vehicle * vehicle_Pt = &myCar;
vehicle_Pt -> drive();
```

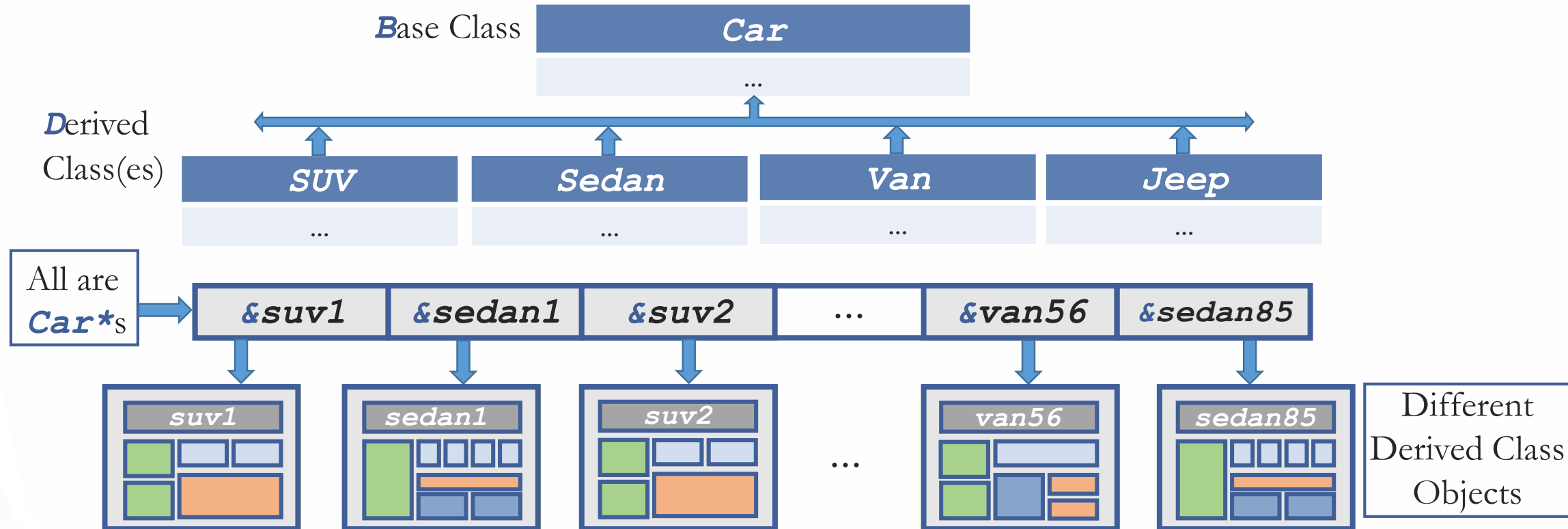| Prototype	Vehicle Class	Car Class
void drive();	Can implement Function Can create Vehicle Object	Can implement Function Can create Car Object Calls Method Vehicle::drive();
virtual void drive();	Can implement Function Can create Vehicle Object	Can implement Function Can create Car Object Calls Method Car::drive();
virtual void drive()=0;	Could implement Function Cannot create Vehicle Object	Must implement Function Can create Car Object Calls Method Car::drive();

Note: This is a Pure virtual Function, and Vehicle is now an Abstract Class.

Note: If no Car::drive() implemented, calls Vehicle::drive().

Polymorphism *(advanced)*

Remember: **Polymorphism & Inheritance**

Supported through Pointers of Base Class-type:

*B*ase Class — **Car** ...

*D*erived Class(es) — **SUV** ... **Sedan** ... **Van** ... **Jeep** ...

All are **Car***s — **&suv1** | **&sedan1** | **&suv2** | ... | **&van56** | **&sedan85**

suv1 | sedan1 | suv2 | ... | van56 | sedan85

Different Derived Class Objects

Polymorphism & Aggregation

Constructor(s) with Dynamic Inference:

```cpp
class Hideout{
 public:
  Hideout(const Hideout & h_other);
 private:
  Car * m_fleet[MAX_FLEET];
};
```

Custom *Copy*-Constructor

Array of Base Class Pointers

Polymorphism & Aggregation

Constructor(s) with Dynamic Inference:

```cpp
class Hideout{
 public:
  Hideout(const Hideout & h_other);
 private:
  Car * m_fleet[MAX_FLEET];
};

Hideout::Hideout(const Hideout & h_other){
  for (int i=0; i<MAX_FLEET; ++i){
    m_fleet[i] = new ???( *(h_other.m_fleet[i]) );
  }
}
```

Custom *Copy*-Constructor

Remember:
Implement *Deep*-Copy,
otherwise *Shallow*-Copy
behavior by default !

Array of Base Class Pointers

new: Instantiate Object through
Constructor call and **return** Pointer.
(More on the Whiteboard, and
more formally later in CS-202 …)

Polymorphism & Aggregation

Constructor(s) with Dynamic Inference:

```cpp
class Hideout{
 public:
  Hideout(const Hideout & h_other);
 private:
  Car * m_fleet[MAX_FLEET];
};
```

Array of Base Class Pointers

```cpp
Hideout::Hideout(const Hideout & h_other){
  for (int i=0; i<MAX_FLEET; ++i){
    m_fleet[i] = new ???( *(h_other.m_fleet[i]) );
  }
}
```

new: Instantiate Object through Constructor call and **return** Pointer. (More on the Whiteboard, and more formally later in CS-202 …)

How to know what to create?
➤ new **SUV**(…);
➤ new **Sedan**(…);
➤ new **Van**(…);
➤ new **Jeep**(…);

?

What underlies here might be:
➤ **SUV** *
➤ **Sedan** *
➤ **Van** *
➤ **Jeep** *

Polymorphism & Aggregation

The `clone() virtual` Method:

```cpp
class BaseClass {
  public:
    virtual BaseClass * clone() const = 0;
    …
};

class DerivedClass : public BaseClass{
  public:
    virtual DerivedClass * clone() const;
    …
};
DerivedClass * DerivedClass::clone() const{
    return new Derived( *this );
}
```

Pure **virtual**:
Is required to be implemented by Derived Class(es).

Note:
How is this allowed?
(do signatures differ?)

Instantiate a **new** Object through Derived Class *Copy*-Constructor call and **return** Derived Class Pointer.

Derived Class *Copy*-Constructor invoked with ***this** as parameter.

Polymorphism

`virtual` Function(s) & Covariant `return` Type(s)

Overriding `virtual` Class Functions does not allow to change `return` Type.
➢ For the case of Dynamic Binding, it is required that:

"whenever the Base Class Method can be called, it should also be directly replaceable by call to Derived Class Method with no change to calling code (i.e. implicit casting is not allowed)."
➢ C++ Standard enforces this by restricting `return` types:

```
class BaseClass{                    class DerivedClass : public BaseClass{
 public:                               public:
  virtual int intFxn();                   virtual double intFxn();
   …                                       …
};                                  };
```

!

Note: Not a case of different signatures due to conflicting **return** types,
         it is C++ standard-enforced requirement for Dynamic Binding.
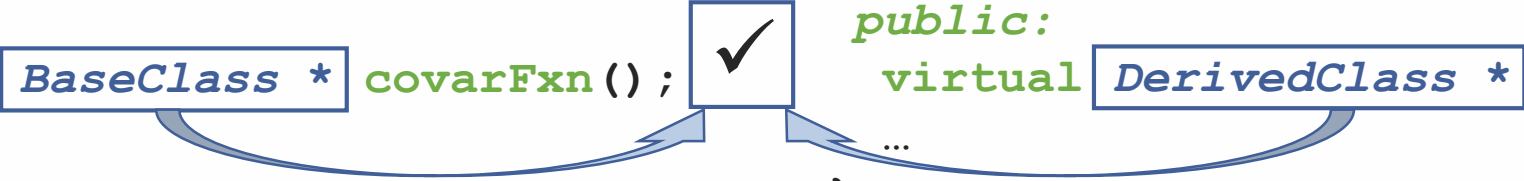   (i.e. `virtual` Functions)

Polymorphism

`virtual` Function(s) & Covariant `return` Type(s)

Overriding `virtual` Class Functions does allow to have different "Covariant" `return` Type(s).

➤ Typical case: Base Class & Derived Class Pointers:

```
class BaseClass{                          class DerivedClass : public BaseClass{
 public:                                   public:
  virtual BaseClass * covarFxn();   ✔       virtual DerivedClass * covarFxn();
  …                                         …
};                                        };
```

Note: Covariant `return` types (Pointers!) allowed
      in the context of Dynamic Binding.
      (i.e. `virtual` Functions)

Polymorphism & Aggregation

The `clone()` Method (put to use):

```cpp
class Vehicle {
 public:
   virtual Vehicle * clone() const = 0;

   …
};

class SUV : public Vehicle{
 public:
   virtual SUV * clone() const;

   …
};

class Sedan : public Vehicle{
 public:
   virtual Sedan * clone() const;

   …
};
```

Pure `virtual`:
Each Derived Class implements own Function and makes sure to returns appropriate (own) Pointer-Type.

```cpp
SUV * SUV::clone() const {
    return new SUV( *this );
}
```

```cpp
Sedan * Sedan::clone() const {
    return new Sedan( *this );
}
```

Polymorphism & Aggregation

Constructor(s) with Dynamic Inference:

```cpp
class Hideout{
 public:
  Hideout(const Hideout & h_other);
 private:
  Car * m_fleet[MAX_FLEET];
};

Hideout::Hideout(const Hideout & h_other){
  for (int i=0; i<MAX_FLEET; ++i){
    m_fleet[i] = h_other.m_fleet[i]->clone();
  }
}
```

Copy-Constructor

Base Class Pointer Array Member

Dynamic Binding will resolve and yield as appropriate:
- ➤ **SUV** *
- ➤ **Sedan** *
- ➤ **Van** *
- ➤ **Jeep** *

`virtual` Destructor(s)

Remember A)*:* Static *vs* Dynamic Binding, when does it occur?

➤ Base Class Pointer – to – Derived Class Object.

➤ `virtual` Class Method, call on Base Class Pointer.

Remember B)*:* Inheritance & Destructor call rules, what is the desired behavior?

➤ Standard specifies: First call to Derived Class Destructor, then Base Class Destructor.

Caveat:

Two ways to invoke a Method (the Class Destructor too !):

➤ Object-based Call (*will* resolve to Object's type Method definition).

➤ Pointer-based Call (*should* resolve to Object's type Method definition).

Polymorphism

virtual Destructor(s)

However: *Static Binding* Case

```cpp
class Vehicle {
 public:
  void ~Vehicle()
  { cout<<"~Vehicle() "; }
};

class Car : public Vehicle{
 public:
  void ~Car(){
    cout<<"~Car() ";
    delete [] engineParts;
  }
 private:
  EngineParts * engineParts;
};
```

> **delete**: Call Destructor Method to remove Object through Object Pointer.
> (More on the Whiteboard, and more formally later in CS-202 …)

```cpp
int main(){
  Car * car_Pt = new Car();
  delete car_Pt;          //dtor call
  car_Pt = NULL;

  Vehicle * vehicleCar_Pt = new Car();
  delete vehicleCar_Pt;   //dtor call
  vehicleCar_Pt = NULL;

  return 0;
}
```

Polymorphism

virtual Destructor(s)

However: *Static Binding* Case

```cpp
class Vehicle {
 public:
  void ~Vehicle()
  { cout<<"~Vehicle() "; }
};

class Car : public Vehicle{
 public:
  void ~Car(){
    cout<<"~Car() ";
    delete [] engineParts;
  }
 private:
  EngineParts * engineParts;
};
```

delete: Call Destructor Method to remove Object through Object Pointer. (More on the Whiteboard, and more formally later in CS-202 …)

```cpp
int main(){
  Car * car_Pt = new Car();
  delete car_Pt;            //~Car()
  car_Pt = NULL;

  Vehicle * vehicleCar_Pt = new Car();
  delete vehicleCar_Pt;  //~Vehicle()
  vehicleCar_Pt = NULL;

  return 0;
}
```

Static Binding makes Base Class Pointer-based call resolve to Base Class Destructor.

engineParts will never be destroyed !

Polymorphism

virtual Destructor(s)

Easy Fix: *Dynamic Binding*

```cpp
class Vehicle {
 public:
  virtual void ~Vehicle()
   { cout<<"~Vehicle()"; }
};

class Car : public Vehicle{
 public:
  virtual void ~Car(){
    cout<<"~Car()";
    delete [] engineParts;
  }
 private:
  EngineParts * engineParts;
};
```

```cpp
int main(){
  Car * car_Pt = new Car();
  delete car_Pt;           //~Car()
  car_Pt = NULL;

  Vehicle * vehicleCar_Pt = new Car();
  delete vehicleCar_Pt;   //~Car()
  vehicleCar_Pt = NULL;

  return 0;
}
```

Dynamic Binding makes Base Class Pointer-based call resolve to Derived Class Destructor.

virtual Destructor(s)

Rule of Thumb:

➢ If a Class is manipulated Polymorphically –
   give it a **virtual** Destructor.

Rule of Knowledge & Experience:

If a Class is Polymorphic its Destructor should:
a) Either be a **public** and **virtual** Dtor.
b) Or be a **protected** and non-**virtual** Dtor.

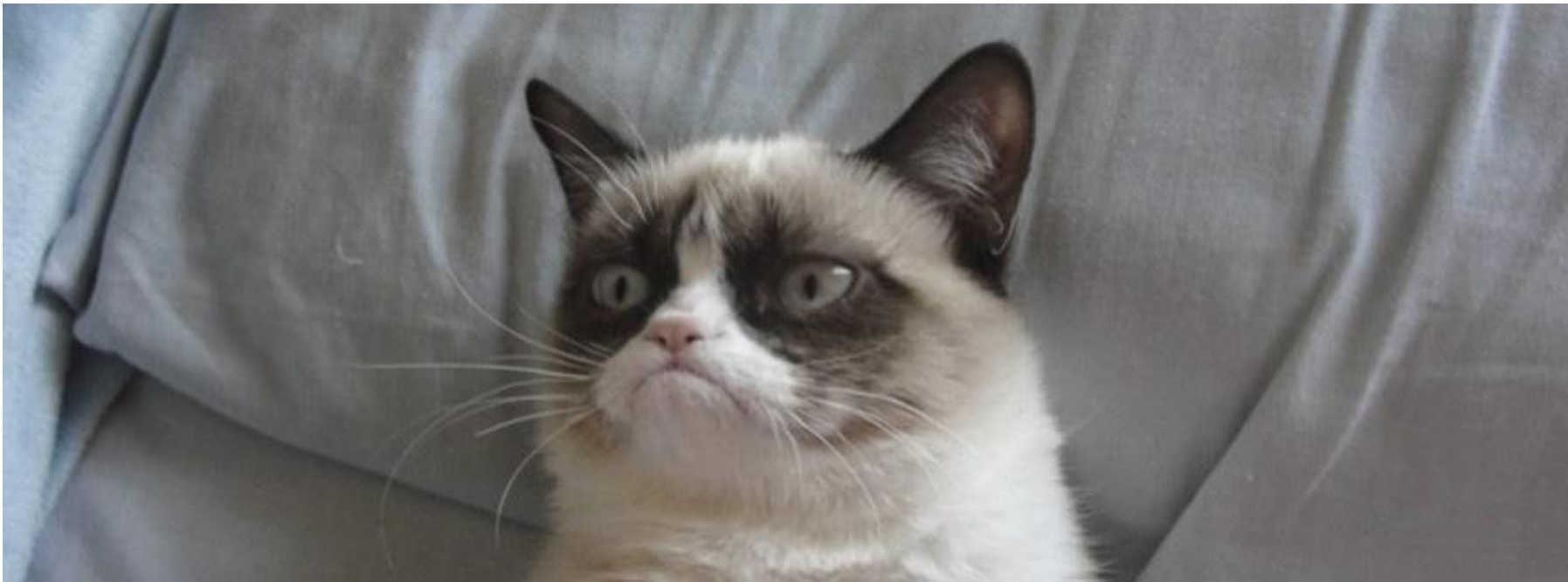
Note:
Check out the commented Inheritance sample
on WebCampus to correlate why.

```cpp
class Vehicle {
 public:
  virtual void ~Vehicle()
   { cout<<"~Vehicle()"; }
};

class Car : public Vehicle{
 public:
  virtual void ~Car(){
      cout<<"~Car() ";
      delete [] engineParts;
  }
 private:
  EngineParts * engineParts;
};
```

`virtual` Constructor(s)

NO

➢ A virtual call is a mechanism to get work done given partial information. In particular, "virtual" allows us to call a function knowing only any interfaces and not the exact type of the object. To create an object you need complete information. In particular, you need to know the exact type of what you want to create. Consequently, a "call to a constructor" cannot be virtual. *Bjarne Stroustrup*
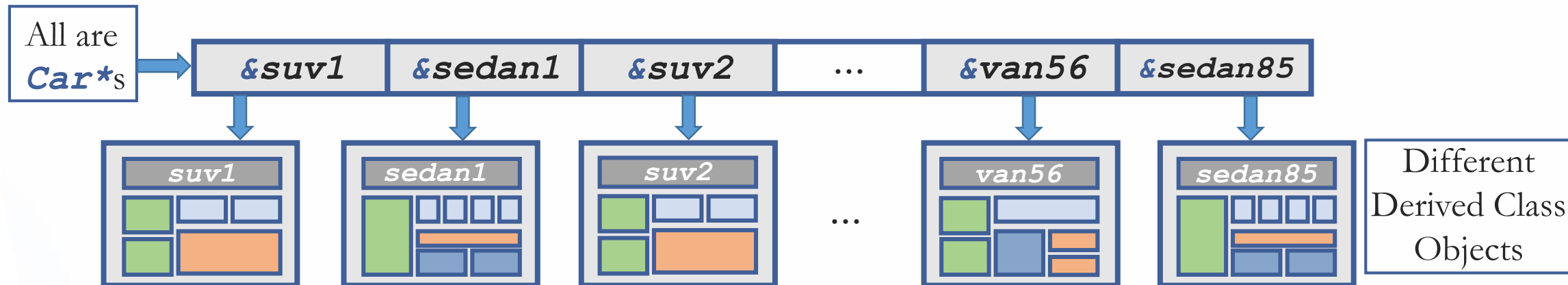
Polymorphism Behind the Scenes

Supported through Pointers of Base Class-type.

➢ Happens at "run-time".

➢ But if the `drive()` function is `virtual`, how does the program know which Child Class's version of the function to call at runtime?

All are `Car*`s

| &suv1 | &sedan1 | &suv2 | ... | &van56 | &sedan85 |

| suv1 | sedan1 | suv2 | ... | van56 | sedan85 |

Different Derived Class Objects

Polymorphism Behind the Scenes

The Compiler uses Virtual Function Tables whenever Polymorphism is used.

Virtual function tables are created for:
➢   Classes with `virtual` functions.
➢   Child Classes of those Base Classes.

The Compiler adds a "hidden" variable to the Base class:
➢   This is Inherited down to every Derived class in the Class Hierarchy.

SUV	SUV	Jeep	Van	Jeep	Sedan	Sedan	Van
*__vptr	*__vptr	*__vptr	*__vptr	*__vptr	*__vptr	*__vptr	*__vptr

It also adds virtual Table of Functions, with Pointers to own `virtual` Class Methods.

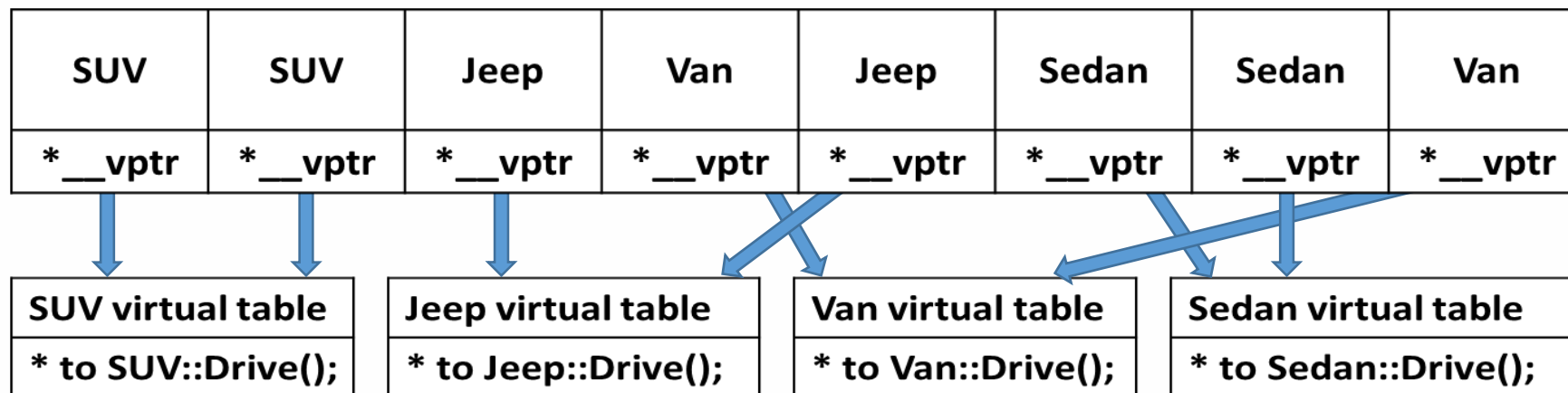| SUV virtual table	Jeep virtual table	Van virtual table	Sedan virtual table
* to SUV::Drive();	* to Jeep::Drive();	* to Van::Drive();	* to Sedan::Drive();

Polymorphism Behind the Scenes

The Compiler uses Virtual Function Tables whenever Polymorphism is used.

Virtual function tables are created for:

➢ Base member **`*__vptr`** is Inherited down to every Derived class.

➢ When an Object is created **`*__vptr`** is set to point to the virtual table for that Class.

`virtual` Function Pointer(s) manipulation happens at "run-time".

SUV	SUV	Jeep	Van	Jeep	Sedan	Sedan	Van
*__vptr	*__vptr	*__vptr	*__vptr	*__vptr	*__vptr	*__vptr	*__vptr

SUV virtual table	Jeep virtual table	Van virtual table	Sedan virtual table
* to SUV::Drive();	* to Jeep::Drive();	* to Van::Drive();	* to Sedan::Drive();

CS-202

Time for Questions !