

CS-202

C++ Classes – Polymorphism (Pt.2)

C. Papachristos

Autonomous Robots Lab
University of Nevada, Reno



Course Week

Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	...	Sunday
			Lab (9:00-12:50)		
	CLASS		CLASS		MIDTERM Sample
PASS Session	PASS Session	Project DEADLINE	NEW Project		PASS Session

Your 6th Project will be announced today Thursday 10/12.


5th Project Deadline was this Wednesday 10/11.

- NO Project accepted past the 24-hrs delayed extension (@ 20% grade penalty).
- Send what you have in time!
- Check out **WebCampus** CS-202 Announcements for some **help** !

Course Week

Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	...	Sunday
			Lab (9:00-12:50)		
	CLASS		CLASS		MIDTERM Sample
PASS Session	PASS Session	Project DEADLINE	NEW Project		PASS Session



Your Midterm will be held on Thursday 10/19.

A Midterm Sample will be announced over the weekend.

- 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

```
class Vehicle {  
    public:  
    void Sound() {  
        cout<<"Hummm..."<<endl;  
    }  
};  
  
class F1Car : public Vehicle{  
    public:  
    void Sound() {  
        cout<<"Zoom!"<<endl;  
    }  
};
```

Objects	<code>int main() {</code>	
	<code>Vehicle vehicle;</code>	
	<code>F1Car f1car;</code>	
	<code>vehicle.Sound();</code>	<code>//Hummm...</code>
	<code>f1car.Sound();</code>	<code>//Zoom!</code>
Object Pointers	<code>Vehicle *vehicle_Pt = &vehicle;</code>	
	<code>vehicle_Pt->Sound();</code>	<code>//Hummm...</code>
	<code>Vehicle *vehicleF1car_Pt = &f1car;</code>	
	<code>vehicleF1car_Pt->Sound();</code>	<code>//Hummm...</code>
	<code>return 0;</code>	

Static Binding

Polymorphism

Remember: Static vs Dynamic Binding

Dynamic Binding

```
class Vehicle {  
    public:  
        virtual void Sound();  
};  
void Vehicle::Sound() {  
    cout<< "Hummm..."<<endl;  
}  
class F1Car : public Vehicle {  
    public:  
        virtual void Sound();  
};  
void F1Car::Sound() {  
    cout<< "Zoom!"<<endl;  
}
```

Objects	<code>int main() {</code>
	<code>Vehicle vehicle;</code>
	<code>F1Car f1car;</code>
	<code>vehicle.Sound();</code> <code>f1car.Sound();</code>
	<code>}</code>
Object Pointers	<code>Vehicle *vehicle_Pt = &vehicle;</code>
	<code>vehicle_Pt->Sound();</code>
	<code>Vehicle *vehicleF1car_Pt = &f1car;</code> <code>vehicleF1car_Pt->Sound();</code>
	<code>return 0;</code> <code>}</code>

*//Hummm...
//Zoom!*

*//Hummm...
//Zoom!*

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();
```

Abstract Classes

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 ;`



Abstract Classes

virtual Functions & Classes

Enabling Polymorphic behaviors.

```
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

Child Classes *may* Override if they choose to.

➤ If not overridden, Parent Class definition used.

Abstract Classes

Pure **virtual** Functions & Classes

Enabling purely Abstract Classes.

```
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.

Child Classes are *required* to provide own Implementations.

- If not overridden, Parent Class definition used.

Abstract Classes

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

Declaration of Pure virtual Method:

- At the same time, the Class becomes an **Abstract Class**.

```
class BaseClass {  
    public:  
    virtual void PureVirtualMethod() = 0;  
    ...  
};
```

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

```
void BaseClass::PureVirtualMethod() { /*potential base implementation ...*/ }
```

Abstract Class Objects:

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

- Not allowed to be instantiated:

```
BaseClass base;
```

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

Abstract Classes

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.

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.

```
class BaseClass {  
    public:  
    virtual void VirtualMethod();  
    void RegularMethod();  
    ...  
};  
class DerivedClass : public 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

Abstract Classes

final Specifier (C++11)

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

```
class BaseClass {  
    public:  
        virtual void VirtualMethod();  
    ...  
};  
class DerivedClass : public BaseClass{  
    public:  
        void VirtualMethod() override final;  
    ...  
};  
class ReDerivedClass : public DerivedClass{  
    public:  
        void VirtualMethod() override;  
    ...  
};
```

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

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

Abstract Classes

final Specifier (C++11)

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

```
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	<i>Vehicle</i> Class	<i>Car</i> Class
<code>void Drive();</code>	Can implement Function Can create <i>Vehicle</i> Object	Can implement Function Can create <i>Car</i> Object Calls Method <i>Vehicle::Drive()</i> ;
<code>virtual void Drive();</code>	Can implement Function Can create <i>Vehicle</i> Object	Can implement Function Can create <i>Car</i> Object Calls Method <i>Car::Drive()</i> ;
<code>virtual void Drive()=0;</code>	Cannot implement Function Cannot create <i>Vehicle</i> Object	Must implement Function Can create <i>Car</i> Object Calls Method <i>Car::Drive()</i> ;

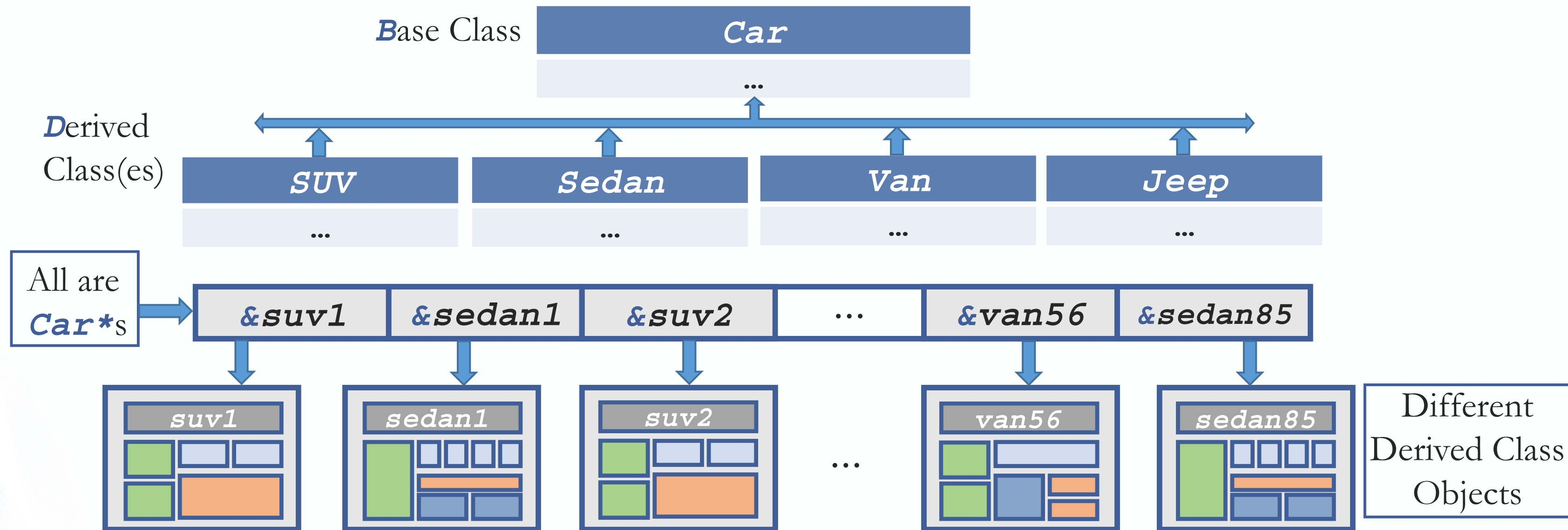
Note: This is a Pure **virtual** Function, and *Vehicle* is now an Abstract Class.

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

Polymorphism

Remember: Polymorphism & Inheritance

Supported through Pointers of Base Class-type:



Polymorphism

Polymorphism & Aggregation

Constructor(s) with Dynamic Inference:

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

Custom *Copy*-Constructor

Array of Base Class Pointers

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

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

How to know what to create?

- new SUV(...);
- new Sedan(...);
- new Van(...);
- new Jeep(...);

What underlies here might be:

- SUV*
- Sedan*
- Van*
- Jeep*

new: Instantiate Object through
Constructor call and **return** Pointer.
(But more on that later...)

Polymorphism

Polymorphism & Aggregation

The `Clone()` `virtual` Method:

```
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{  
    public:  
    virtual int IntFxn();  
    ...  
};  
  
class DerivedClass : public BaseClass{  
    public:  
    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)

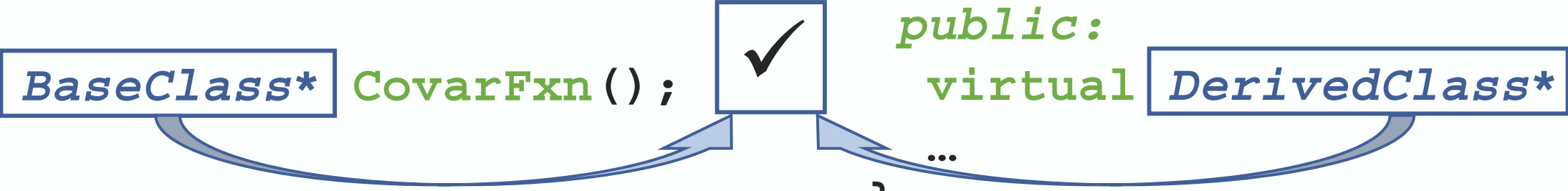
Polymorphism

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{  
    public:  
    virtual BaseClass* CovarFxn ();  
    ...  
};  
  
class DerivedClass : public BaseClass{  
    public:  
    virtual DerivedClass* CovarFxn ();  
    ...  
};
```



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

Polymorphism

Polymorphism & Aggregation

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

```
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 return appropriate (own) Pointer-Type.

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

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


Polymorphism

Polymorphism & Aggregation

Constructor(s) with Dynamic Inference:

```
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*

Polymorphism

virtual Destructor(s)

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

- Base Class Pointer – 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

```
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. (But more on that later...)

```
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*

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

```
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.

Polymorphism

virtual Destructor(s)

Rule of Thumb:

- If a Class has one or more **virtual** Methods
– 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.

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

Polymorphism

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.

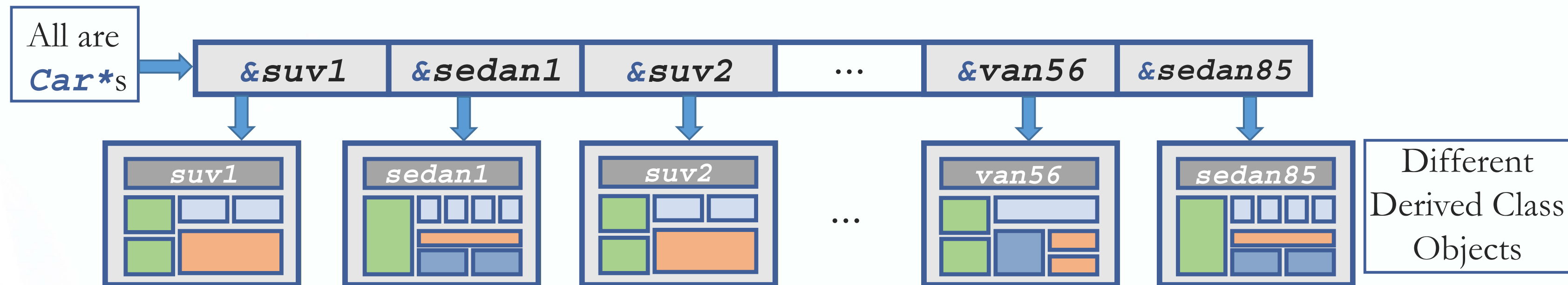
Bjarne Stroustrup

Virtual Function Tables

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 compiler know which Child Class’s version of the function to call?



Virtual Function Tables

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();

Virtual Function Tables

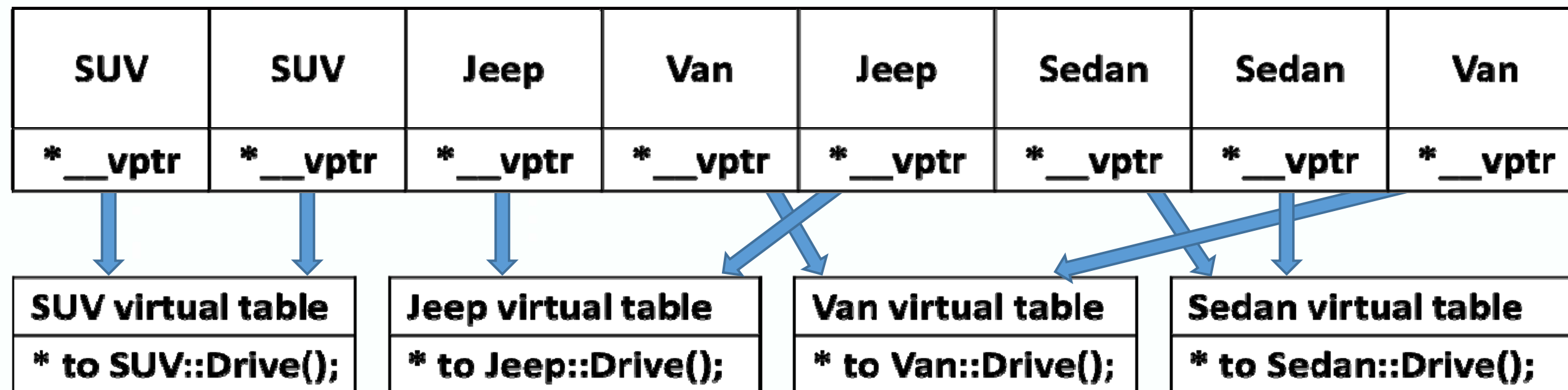
Polymorphism Behind the Scenes

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”.



CS-202

Time for Questions !