(6) C++ Abstractions

Nico Ludwig (@ersatzteilchen)

TOC

- Souces:
  - Bjarne Stroustrup, The C++ Programming Language
  - John Lakos, Large-Scale C++ Software Design

Initial Words

Yes, my slides are heavy.

I do so, because I want people to go through the slides at their own pace w/o having to watch an accompanying video.

On each slide you'll find the crucial information. In the notes to each slide you'll find more details and related information, which would be part of the talk I gave.

Have fun!

Defining the UDT Car with Classes

- Definition of the UDT *Car*:
  - It has two fields:
    - *theEngine*
    - *spareTyre*
    - *Car* has an *Engine* and a SpareTyre.
  - And three member functions:
    - *StartEngine()*
    - *SetSpareTyre()*
    - *GetSpareTyre()*

```
class Car { // UDTs Engine and Tyre elided.
	Engine* theEngine;
	Tyre* spareTyre;
public:
	void StartEngine() {
		std::cout<<"start Car"<<std::endl;
		theEngine->Start();
	}
	void SetSpareTyre(Tyre* spareTyre) {
		this->spareTyre = spareTyre;
	}
	const Tyre* GetSpareTyre() const {
		return spareTyre;
	}
};
```

- The definition of this UDT shouldn't contain any surprises.
  - Therefor we're going to use it for the following discussion.

```
// Creation and usage of a Car instance:
Car fordFocus;
fordFocus.SetSpareTyre(new Tyre);
fordFocus.StartEngine();
```

Concepts of Object Orientation

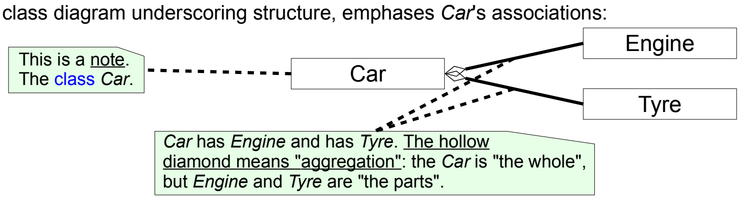
- Effectively, the combination of self contained data and behavior is our aim.

- In sum, following concepts are required to implement this aim:
  - 1. Abstraction by combining data and functions into a type.
  - 2. Encapsulation to protect data from unwanted access and modification:
    - The *day*-part of a *Date* instance should not be modifiable from "outside".
  - 3. The whole – part (aggregation or composition) association:
    - We can say "A car object has an engine object.".
  - 4. The specialization – generalization (inheritance) association:
    - We can say "three cars drive in front of me", rather than saying that there "drives a van, a bus and a sedan in front of me". The generalization is possible, because e.g. a bus is a car.

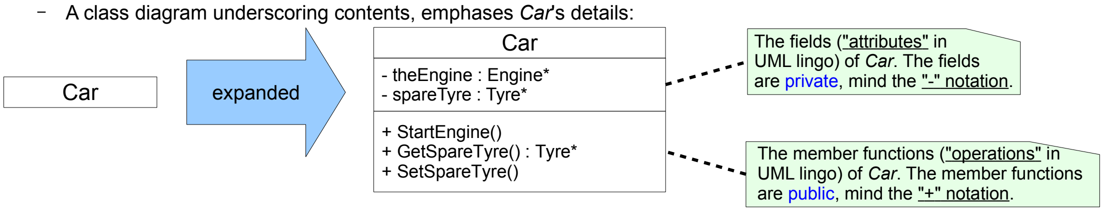
- "Object-orientation" (oo) is the umbrella term for these concepts.
  - Oo languages provide features that allow expressing these concepts.
  - Now we're going to understand specialization – generalization.

UML Notation of Car with Whole-Part Associations

- A Unified Modeling Language (UML) "class diagram" to design aggregated types.
  - A class diagram underscoring structure, emphases *Car*'s associations:



  - A class diagram underscoring contents, emphases *Car*'s details:



  - A *Car* instance aggregates an *Engine* instance (Not: the UDT *Car* aggregates the UDT Engine!)

- UML is a language to describe oo systems.
  - The class diagram in the structural notation shows the associations.
  - The conciser notation shows the details of the types (members, their accessibility etc.).
  - The static fields and methods are represented with underlined texts in UML.

Expressing Generalization – Specialization in C++

- Present UDTs can be used as base types for new types.
  - More special types inherit from more general types. This is called inheritance.
    - Instead of "inherits from" we'll occasionally use the phrases "derives from" or "extends".
  - The data (i.e. the (private) fields) of base types is inherited by the new type.
  - The behavior (i.e. the member functions) of base types is inherited by the new type.
    - The new type inherits the public interface from its base types.
    - The inheriting type can access the public interface of its base types but not their private members.
  - The new type can add more members to the inherited ones.
  - Where an object of base type was used, an object of the derived type can also be used.
    - This is called substitution principle.
- In C++ a new type can inherit from more than one type!
  - This is called multiple inheritance (MI). We are not going to discuss multiple inheritance!
- Let's learn how inheritance is notated in C++...

- C++' fundamental types and enums can not be used as base types!

Inheritance and Type Specialization in C++

- We've to adorn the class definition of the derived type by naming the base types.
  - Here we're going to introduce the UDT *Bus* being inherited from *Car*.
  - This inheritance expresses following specialization: a *Bus* is a *Car*.

```
// Bus inherits from Car:
class Bus : public Car { // (members hidden)
	static const int COUNT_SEATBENCHES = 42;
	int nOccupiedSeatBenches;
public:
	bool NewPassengerCanEnter() const {
		return COUNT_SEATBENCHES <= nOccupiedSeatBenches;
	}
};
```

- So let's instantiate and use a *Bus*:
  - Mind that we can call *Car*'s inherited and *Bus*' new member functions on a *Bus* object.

```
Bus bus;
bool hasVacantSeats = bus.NewPassengerCanEnter(); // Accessing a member function of Bus.
// Accessing a member function of Bus' base type Car:
const Tyre* spareType = bus.GetSpareTyre(); // Bus inherited (almost) all members of Car. A Bus is a Car.
```

  - Ctors of base types are not inherited to the subtype!

- The type *Bus* is derived from *Car*. So a *Bus* is a *Car*.
- A *Bus* has 42 seat benches. This means that the count of seat benches is limited.
- A certain count of seat benches (*nOccupiedSeatBenches*) is occupied by passengers.
- *Bus* introduces a new public member function *NewPassengerCanEnter()*. This member function is not present in the base type *Car*. The interface of *Bus* then consists of all public members of *Car* (save *Car*'s ctors) as well as the new public member function *NewPassengerCanEnter()*.
  - Is the type *Bus* allowed to access *theEngine* in *Car*?
- Ctors of base types are not directly inherited to the subtype! But they are called on the creation of a new instance from the most special to the most general type.

Sidebar: Non-public Inheritance

- We've used public inheritance in *Bus*. What happens without the public keyword?

```
class Car { // (members hidden)
public:
    void StartEngine() { /* pass */ }
};
```

```
// Bus inherits from Car non-publicly:
class Bus : Car { // (members hidden)
};
```

```
// => Bus inherits from Car privately:
class Bus : private Car { // (members hidden)
};
```

- In this case non-private members of *Car* are inherited as private members into *Bus*:

```
Car car;
car.StartEngine(); // Ok! Call the public function Car::StartEngine().
```

```
Bus bus;
bus.StartEngine(); // Invalid! The inherited function Car::StartEngine() can't be called! - It's private in Bus!
```

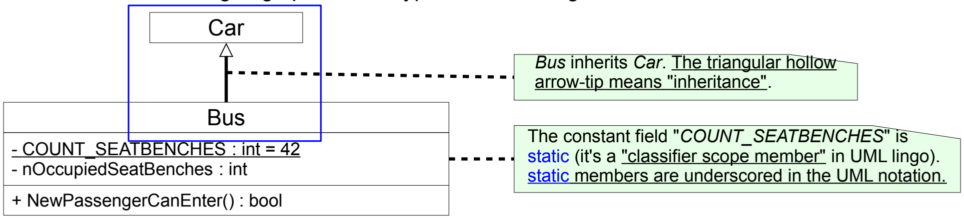
- Non-public inheritance is used rarely.
  - It can be useful with multiple inheritance and to "encapsulate substitutability".
  - Leaving away the public keyword when using inheritance is most often just a "slip of the pen".
- By default classes apply private inheritance and structs apply public inheritance.

9

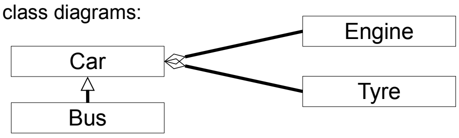
- There also exists protected inheritance. – Its effect can be derived form the effect of private inheritance.
- "Encapsulation of substitutability" means that everywhere a *Car* was used we can no longer pass a *Bus*. – Because this "is a" relationship grew private as well!

UML Notation of Car with the Specialization Bus

- The UML does also allow designing specialized types in class diagrams:



- In oo, aggregation and specialization associations create a type hierarchy!
  - This can be nicely shown in UML class diagrams:



  - We're going to extend this hierarchy during the following discussion.

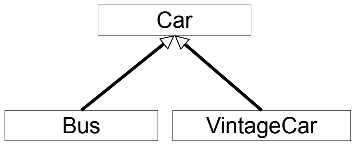
So, what is oo Programming all about?

- The reality can be simulated to a good degree with oo-types!
  - Most real associations can be expressed with aggregation and specialization.
  - Even customers can understand, how oo-types function (UML)!
- Thinking in type interfaces rather than in algorithms and procedures:
  - Interface and implementation can be separated, but types are complete.
  - First define all interfaces, then implement the type completely.
- Instead of algorithms with functions, we can build architectures with UDTs.
  - Recurring UDTs architectures are called design patterns.
  - There exist simple and complex design patterns.
- Successful (oo) programmers identify and use design patterns.

Inheritance defines Substitutability

- We discussed that, e.g., a *Bus* is a *Car* and we expressed this with inheritance.

- Another way to understand this association: <u>inheritance defines substitutability</u>!
  - Wherever the UDT *Car* is awaited, any UDT inherited from *Car* can be used, e.g. *Bus*.
  - Because a <u>*Bus* is a *Car*</u>!



- Let's understand this better with an example of substitution.
  - We'll also introduce another subtype of *Car*: *VintageCar*!

```
Bus bus;
VintageCar vintageCar;
Car anotherCar1 = bus; // Aha! We can assign a Bus to a Car object. A Bus is a substitute for a Car.
Car anotherCar2 = vintageCar; // We can assign a VintageCar to a Car object. A VintageCar is a
                              // substitute for a Car.
```

```
Bus anotherBus = vintageCar;  // Invalid! A VintageCar is no Bus. VintageCar and Bus are
                              // siblings in the inheritance hierarchy, they don't "know" each other!
```

Substitutability and Object Slicing

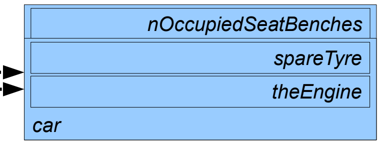
- The substitution principle as we have used it introduces a problem: slicing.

- When an object of derived type is assigned to an object of base type, the derived type's additional fields are sliced away!
  - The object *bus* contains all fields of type *Bus*:

    ```
    Bus bus;
    ```

  - Assigning *bus* to the *car* object slices the object's additional *Bus*-fields:

    ```
    Car car = bus; // Slicing (calls the cctor).
    ```

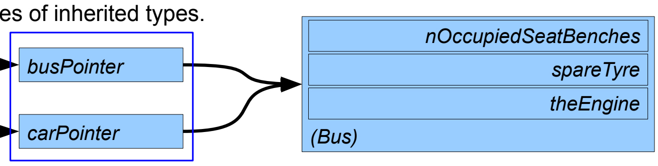


- Substitution also works for pointer (and reference) types of inherited types.

  ```
  Bus* busPointer = &bus;
  ```

  - But assigning a pointer to another pointer doesn't slice! *carPointer* still points to a "full" *Bus* object!

  ```
  Car* carPointer = &bus; // No slicing!
  ```



- Using pointers added an extra layer of indirection, which avoids object slicing!

Static and dynamic Type of an Object

- On using substitutability with pointers, an object can have a static and a dynamic type.
  - Originally we used pointers to avoid object slicing with an extra layer of indirection.
  - The pointer type of the object is the static type of the object.
    - The static type is the type of the variable.
  - The type of the object "behind the pointer" is the dynamic type of the object.
  - Hence we'll use pointers with dynamic memory allocation to make the difference clear.

- We can cast the dynamic type "out of" the statically typed pointer.
  - Then we can access the members of the dynamic type.
  - Objects of varying dynamic type can be referenced by an object of a static type.

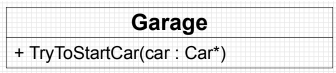
```
Car* car = new VintageCar; // Let's point car to a VintageCar. Static type: Car*, dynamic type: VintageCar.
VintageCar* vintageCarBehind = static_cast<VintageCar*>(car); // (1) Cast "the VintageCar out of car".
vintageCarBehind->CrankUntilStarted();
Bus* busBehind = static_cast<Bus*>(car);    // (2) Also ok! Cast "a Bus out of car". But it doesn't work as
                                            // intended as car doesn't point to a Bus, it is similar
                                            // to "cast contact lenses".
```

  - (1) This kind of casting is called "downcasting", as we cast "down the type hierarchy".

- The dynamic type is sometimes called the "run time type".
- The separation of static and dynamic type does also work with C++ references.
- There are languages in which variables have no static type at all, e.g. Smalltalk and JavaScript.
- (2) There is no compile time error, because *car* could really point to a *Bus*, because *Bus* is more special than *Car*, the compiler just trusts the programmer, downcasting is allowed as both types are related. So the compiler holds the cast to be ok, static_casts are type-checked at compile time. Accessing any member of *busBehind*, instead those inherited from *Car*, results in undefined behavior.

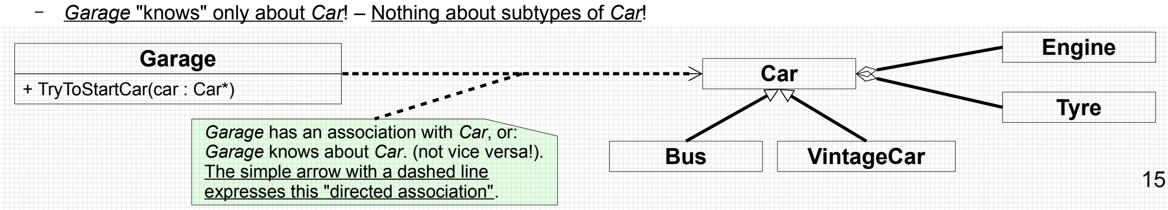
The Garage – Car Dependency

- UDTs itself can act as base types to exploit substitutability.



  - Consider the recently presented class *Car* and the new class *Garage*.
  - In *Garages*, *Cars* are tried to be started with *Car*'s member function *StartEngine()*.
    - The UDT *Garage* has an association with the UDT *Car*.
  - In *Garages* it should be allowed to start all sorts of *Cars*.
    - Because the behavior of all *Cars* is the same, however, we can start the engine.
  - All *Cars* have the publicized member function *StartEngine()* in their public interface.

- In UML we can express that "*Garage* has an association with the UDT *Car*".
  - *Garage* "knows" only about *Car*! – Nothing about subtypes of *Car*!



15

- The term "association": Aggregation and inheritance are nothing but special (and also tighter) associations.
- OO seems to be perfectly good to build GUIs, because the visual representation of GUIs can be directly simulated with oo UDTs and oo paradigms (e.g. "has"- and "is a"-associations of forms and controls). – In opposite to procedural or functional programming.

Using Cars in Garages

Garage

+ TryToStartCar(car : Car*)

```
class Garage { // (members hidden)
public:
	void TryToStartCar(Car* car) const {
		car->StartEngine();
	}
};
```

- Now let's inspect the class *Garage* and its member function *TryToStartCar()*:
  - We've just learned that the more special UDT *Bus* can substitute *Car*, so this is possible:

```
Garage joesStation;
Car* seatLeon = new Car;
Bus* mercedesIntegro = new Bus;
joesStation.TryToStartCar(seatLeon); // Should call Car's StartEngine().
joesStation.TryToStartCar(mercedesIntegro); // Should call Bus' StartEngine() inherited from Car.
```

- The substitutability allows us to pass more special types to *TryToStartCar()*.
  - But calling *StartEngine()* on the passed *car* always calls *Car::StartEngine()*!
  - We awaited *Car::StartEngine()* and *Bus::StartEngine()* to be called!
  - The problem: *StartEngine()* is called on the static type, not on the dynamic type!
  - We should analyze this effect in a more detailed manner...

- This example shows how the "substitution principle" allows that the types of a function's arguments (dynamic types) need not to be exactly the types of the declared parameters (static types).

The Problem of special Behavior

- To make the mentioned problem visible we'll introduce the UDT *VintageCar*:
  - Some *VintageCar*s can only be started with a crank (if they have no starter)!
  - Let's respect these facts in *VintageCar*'s interface:

```
class VintageCar : public Car { // (members hidden)
public:
	bool CrankUntilStarted() { /* pass */ }
	bool HasStarter() const { /* pass */ }
};
```

- Ok, then we're going to give our *fordTinLizzie* to *joesStation*:

```
VintageCar* fordTinLizzie = new VintageCar(1909);
joesStation.TryToStartCar(fordTinLizzie); // Oops! Calls Car's StartEngine() and this is not enough!
// This is no compiler error, but a logical error: the fordTinLizzie can't be started in TryToStartCar()!
```

  - Nobody in *joesStation* knows how to start our *fordTinLizzie*!
  - They can't just turn the key and keep going!
  - The passed *Car* is a *VintageCar* (dynamic type) – this is not respected in *TryToStartCar()*.
  - Maybe *VintageCar* is so special that we have to enhance *TryToStartCar()*?

Type Flags to the Rescue

- We can solve the problem by the introduction of a *CarType* flag field:

```
// An enum for CarType flags:
enum CarType {
	CAR_TYPE,
	BUS_TYPE,
	VINTAGE_CAR_TYPE
};
```

```
// Add a CarType field to the UDT Car:
class Car { // (members hidden)
	CarType carType; // The type flag.
public:
	CarType GetCarType() const {
		return carType;
	}
	void SetCarType(CarType carType) {
		this->carType = carType;
	}
};
```

- Let's apply the *CarType* flag on our *Car* types:

```
// Create a Car object and flag it as being of "CAR_TYPE".
Car* seatLeon = new Car;
seatLeon->SetCarType(CAR_TYPE);
// Create a VintageCar object and flag it as being of "VINTAGE_CAR_TYPE".
Car* fordTinLizzie = new VintageCar;
fordTinLizzie->SetCarType(VINTAGE_CAR_TYPE);
```

  - How will this help us? => With this flag we can identify the dynamic type of a *Car*!

Type Flags in Action

- Now we have to modify *TryToStartCar()* to interpret the *CarType* flag:

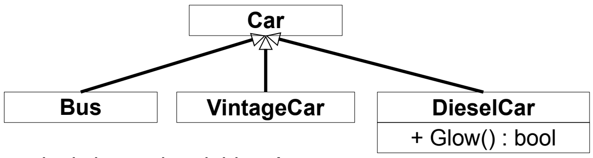
```
void Garage::TryToStartCar(Car* car) const {
	// If car's CarType flag is VINTAGE_CAR_TYPE start it in a special way:
	if (VINTAGE_CAR_TYPE == car->GetCarType()) {
		// Cast the dynamic type "out of" car:
		VintageCar* vintageCar = static_cast<VintageCar*>(car);
		// ...to access VintageCar's interface:
		if (!vintageCar->HasStarter()) {
			vintageCar->CrankUntilStarted();
		} else {
			vintageCar->StartEngine();
		}
	// Else use the default start procedure for other cars:
	} else {
		car->StartEngine(); // Start other cars just by calling StartEngine().
	}
}
```

- Yes, with this implementation of *TryToStartCar()* we can start *fordTinLizzie*:

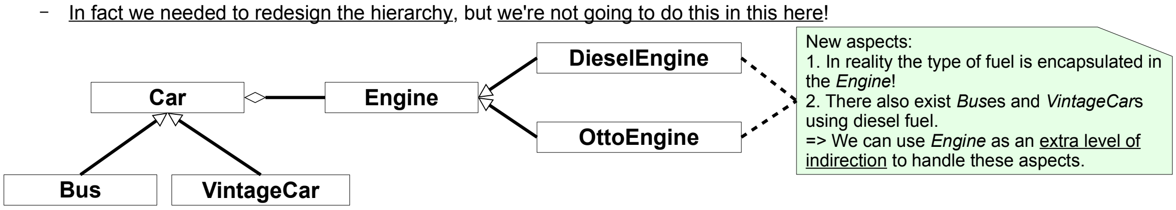
```
VintageCar* fordTinLizzie = new VintageCar(1909);
fordTinLizzie->SetCarType(VINTAGE_CAR_TYPE);
joesStation.TryToStartCar(fordTinLizzie);    // Ok! TryToStartCar() will pick the correct start-algorithm
                                             // depending on the CarType flag!
```

Other special Types need Handling

- After a while the clientele at *joesStation* grows:
  - More and more clients bring their diesel cars to *joesStation*.
  - But old diesel cars can't be started in *Garages*! A glowing procedure is required!
  - As an oo programmer we start by encapsulating the diesel concept into a UDT:



- Btw.: Virtually our type hierarchy is becoming dubious!
  - In fact we needed to redesign the hierarchy, but we're not going to do this in this here!



Type Flags are becoming nasty...

- After extending the enum *CarType* we can modify *TryToStartCar()* accordingly:

```
// The CarType flag
// DIESEL_CAR_TYPE needs
// to be added:
enum CarType {
        // (members hidden)
        DIESEL_CAR_TYPE
};
```

```
void Garage::TryToStartCar(Car* car) const {
        if (VINTAGE_CAR_TYPE == car->GetCarType()) { /* pass */
        } else if(DIESEL_CAR_TYPE == car->GetCarType()) {
            DieselCar* dieselCar = static_cast<DieselCar*>(car);
            dieselCar->Glow();
            dieselCar->StartEngine();
        } else { /* pass */ }
}
```

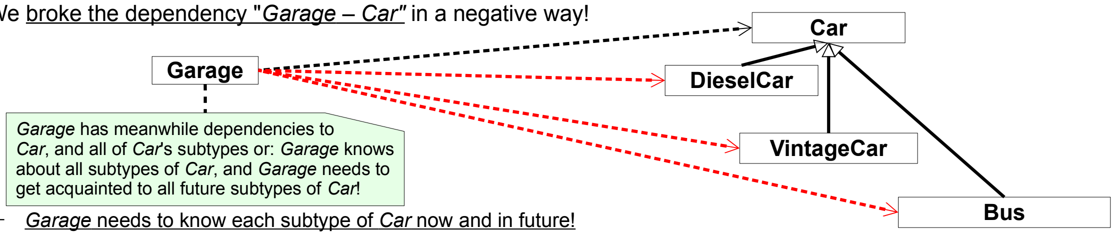
```
DieselCar* dieselCar = new DieselCar();
dieselCar->SetCarType(DIESEL_CAR_TYPE);
joesStation.TryToStartCar(dieselCar); // Ok! TryToStartCar() will pick the correct start-algorithm!
```

- This works, but should we really add more *CarType* flags for upcoming *Car* types?
  - Let's remember that we also have to add more else ifs for upcoming *Car* types!
  - It's uncomfortable and unprofessional doing the same all over and knowing about that!
  - This approach is also very error prone!

- Hm... what is the basic problem we've encountered just now?

A "Pasta-object-oriented" Hierarchy

- We <u>broke the dependency "*Garage – Car*"</u> in a negative way!



  - <u>*Garage* needs to know each subtype of *Car* now and in future!</u>
    - We can see this, because *Garage::TryToStartCar()* inspects the dynamic type of the *Car* parameter.
    - We needed to <u>interpret flags</u>, because <u>we had to deal w/ different interfaces of the dyn. type</u>.
    - We have to make <u>downcasts to the dynamic type</u> respectively!
    - The bad consequence: <u>*Garage::TryToStartCar()* must be modified when new *Car* types emerge!</u>
    - <u>And *Garage* as well as the new *Car* type (maybe also the enum's cpp-file) must be recompiled!</u>
  - We can also <u>spot a "bad smell" in the class diagram</u>!
    - <u>There are dependencies to the base type *Car* and to all of its (currently known) subtypes.</u>

- It lead us to <u>spaghetti programming the oo-way</u>! How can we improve that?

- Often too many arrows pointing from a single type to multiple other types is a <u>bad smell</u>. – It can be nicely spotted in a UML class diagram.
- What we've just seen on using enum type flags is called the "typedef-enum-antipattern".

Hiding and Overriding

- The idea is to not let *Garage::TryToStartCar()* select the start algorithm!
  - Instead, the start algorithm should be put into each individual *Car*-type!

```
// Starts the Car, special implementation for VintageCars.
void VintageCar::StartEngine() {
	std::cout<<"start VintageCar"<<std::endl;
	if (HasStarter()) { // Call the hidden StartEngine() in
	                    // the base class Car!
		Car::StartEngine();
	} else {
		while (CrankUntilStarted()) { /* pass */ }
	}
}
```

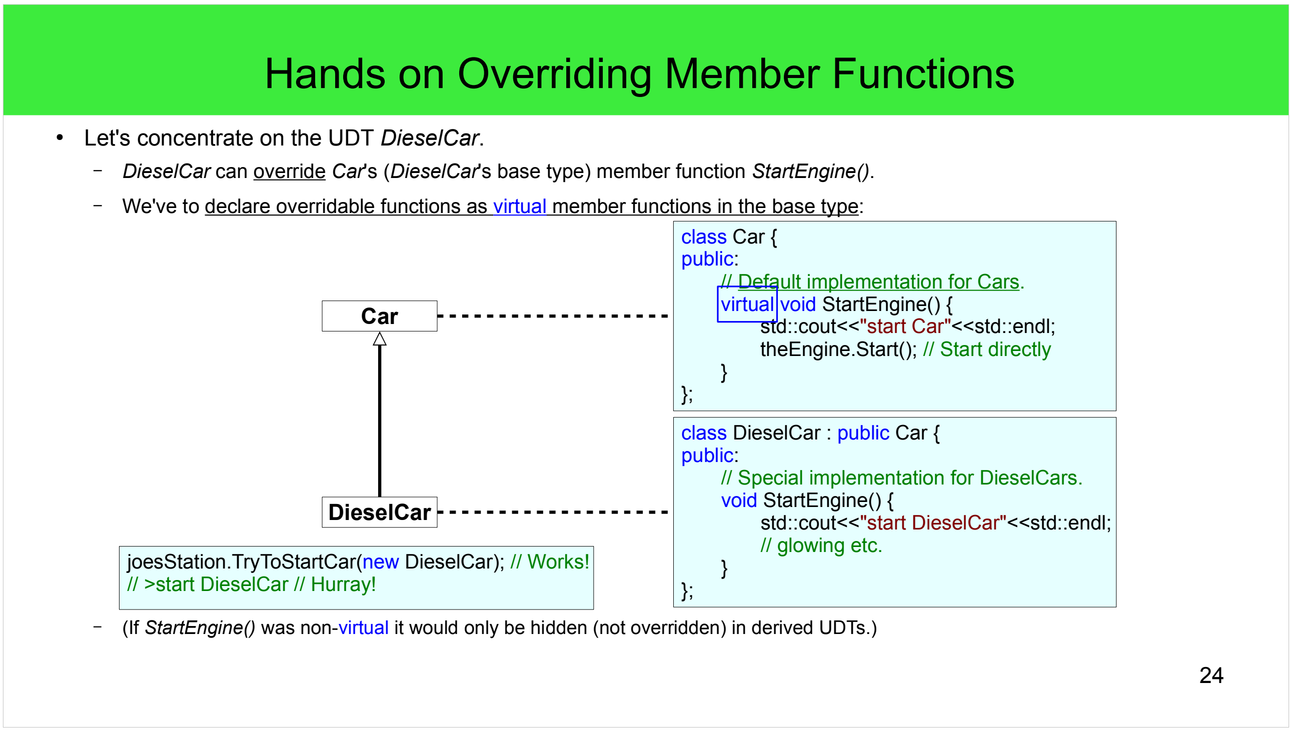
```
// Starts the Car, special implementation for
// DieselCars.
void DieselCar::StartEngine() {
	std::cout<<"start DieselCar"<<std::endl;
	Glow(); // Glowing.
	// Call the hidden StartEngine() in the
	// base class Car!
	Car::StartEngine();
}
```

```
void Garage::TryToStartCar(Car* car) const {
	car->StartEngine(); // Well, that's a simple implementation! The knowledge and responsibility about
	                    // and for the start algorithm is delegated to the dynamic type "behind" car.
}
```

- But it can't work, as we didn't override the member function *Car::StartEngine()*!

```
joesStation.TryToStartCar(fordTinLizzie); // But... it doesn't work!
// >start Car // Car::StartEngine() is called!
joesStation.TryToStartCar(new DieselCar); // Doesn't work either!
// >start Car // Car::StartEngine() is called!
```


- Why do we need to call *StartEngine()* with a *Car*::-prefix?
  - 1. *VintageCar* could derive from multiple other base types, so one base type needs to be explicitly designated.
  - 2. If *StartEngine()* would be called in *VintageCar::StartEngine()* it lead to an infinite recursion, because *Car::StartEngine()* is hidden in *VintageCar::StartEngine()*.

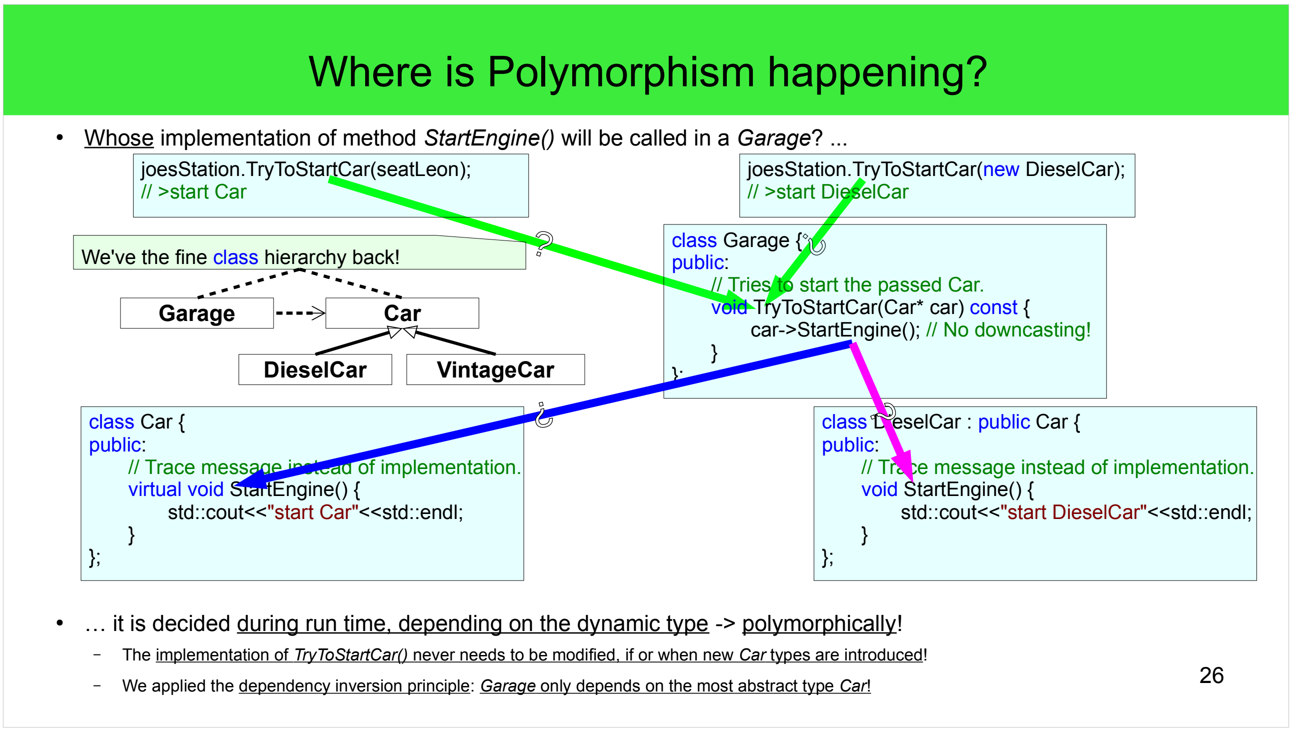


- Sometimes the keyword virtual is also written on the overriding methods (in this example it could be written on the method *DieselCar::StartEngine()*). It's done because of tradition and in order to mark that the method overrides another method. However, it is optional and it will not be checked by the compiler. – We're not going to repeat the virtual keyword on overriding methods in this course's code examples.
- The virtual keyword must not be written on a non-inline definition.

Polymorphism – chooses Implementation during Run Time

- The inherited implementation of a member function could be inadequate!
  - *Car::StartEngine()* inherited by *DieselCar* must glow, before starting the *Engine*!
- Inheriting types can override inherited implementations of member functions.
  - In C++ overridable member functions must be marked with the virtual keyword.
- virtual member functions are often called methods in C++!
- Overriding (late binding) is not overloading (early binding)!
  - Overriding takes effect during run time, overloading during compile time!
  - Overrides in a subtype mustn't change the signature of the overridden method.
- Calling implementations of dynamic types on objects is called oo polymorphism.

- The names of the parameters in an overriding method need no to be same as in the to-be-overridden method.
- In C++, member functions are not virtual by default. One of the reasons for this is, that the C++ designers wanted programmers to make an explicit choice here: virtual member functions can cost performance, in many cases they cannot be "optimized away by the compiler".
- Polymorphism happens during run time and information like the vtables need to be evaluated during run time to make method dispatching work. In other words: calling methods is more costly than calling non-virtual member functions.



- Which implementation of *StartEngine()* will be called?
  - *Car*'s implementation?
  - *VintageCar*'s implementation?
- Polymorphism:
  - In future, the implementation of the method *StartEngine()* of any other sub type of *Car* could be called!
  - In fact, the operator<< as we've overloaded it in a former example uses polymorphism in a similar way *StartEngine()* does:
    - As operator<<'s *lhs* is of type *std::ostream&*, <u>it accepts *std::cout* as well *std::ofstreams*</u> (so the <u>substitution principle as well as static and dynamic types are exploited</u>).
- When a virtual member function is called, the dynamic type of the object determines, which implementation of that member function is really called. When a non-virtual member function is called, the static type of the object determines, which concrete implementation is called.

When to use Inheritance?

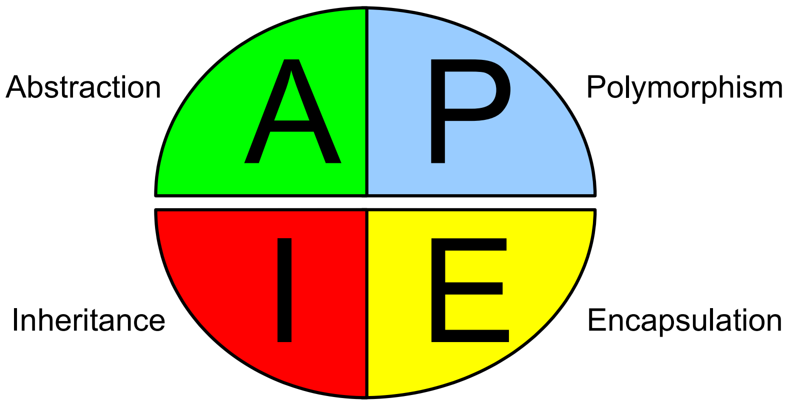
- Inheritance can be used to express:
  - 1. Generalization – specialization associations.
    - *Bus* is more special than *Car*, this statement justifies inheritance.
  - 2. The substitution principle and polymorphism.
    - Substitution: Whenever a *Car* is awaited; an object of any inherited UDT can be used.
    - Polymorphism: Methods may behave be differently in derived types.
  - 3. White-box reuse.

- The most important usages are the substitution principle and polymorphism.
  - It comes along with generalization – specialization associations.
  - The reuse of the interface (the behavior) is the primary usage aspect!
  - White-box reuse is a secondary aspect!

- Inheritance for reusing interfaces is the basis for most design patterns.

APIE

- The core principles of object orientation can be summarized as "A PIE":
  - (after Peter van der Linden)



Thank you!