(7) C++ Abstractions

Nico Ludwig (@ersatzteilchen)

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.

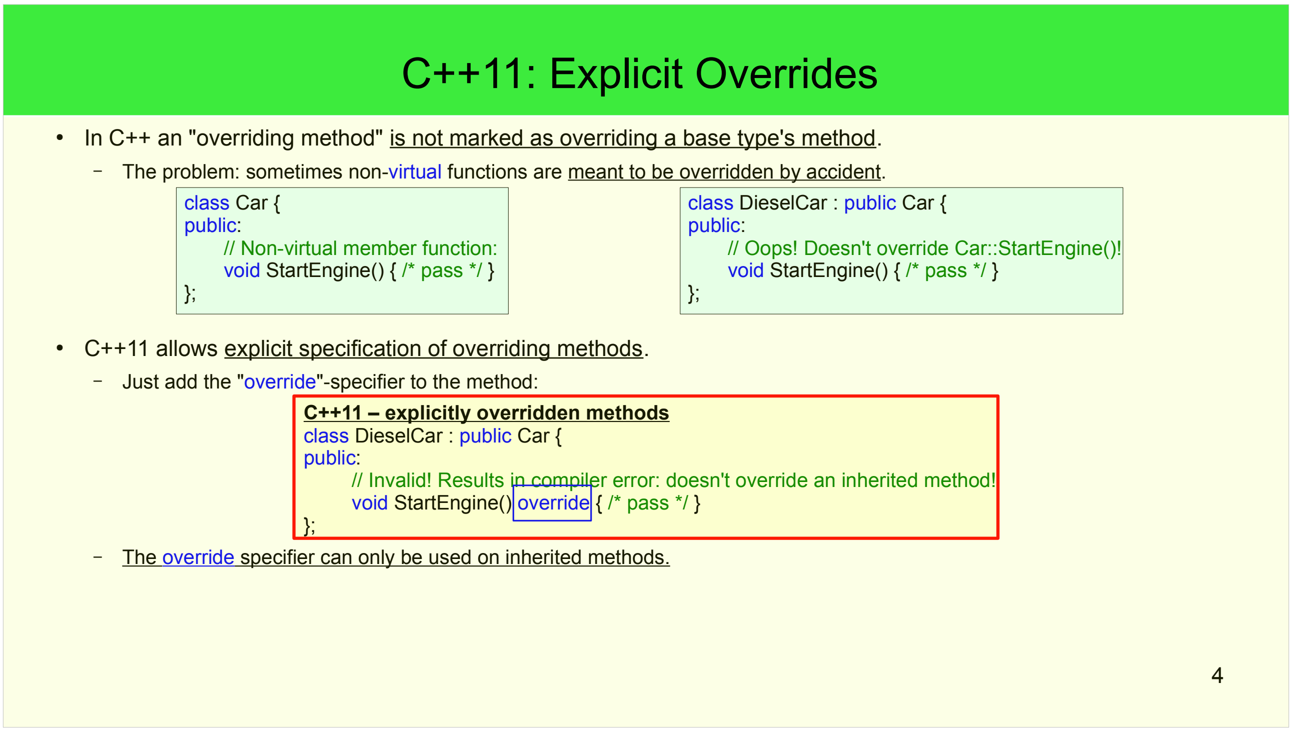
On each slide you'll find the crucial information. In the notes to each slide you'll find more details and related information, which would be part of the talk I gave.

Have fun!

TOC

- (7) C++ Abstractions
  - C++11: Explicit overrides and final Methods
  - Run Time Type Identification (RTTI) and dynamic_cast
  - protected Members
  - Abstract Types with pure virtual Member Functions
  - Virtual Destructors
  - Inheritance and Operators

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



C++11: Explicit Overrides

- In C++ an "overriding method" is not marked as overriding a base type's method.
  - The problem: sometimes non-virtual functions are meant to be overridden by accident.

```
class Car {
public:
	// Non-virtual member function:
	void StartEngine() { /* pass */ }
};
```

```
class DieselCar : public Car {
public:
	// Oops! Doesn't override Car::StartEngine()!
	void StartEngine() { /* pass */ }
};
```

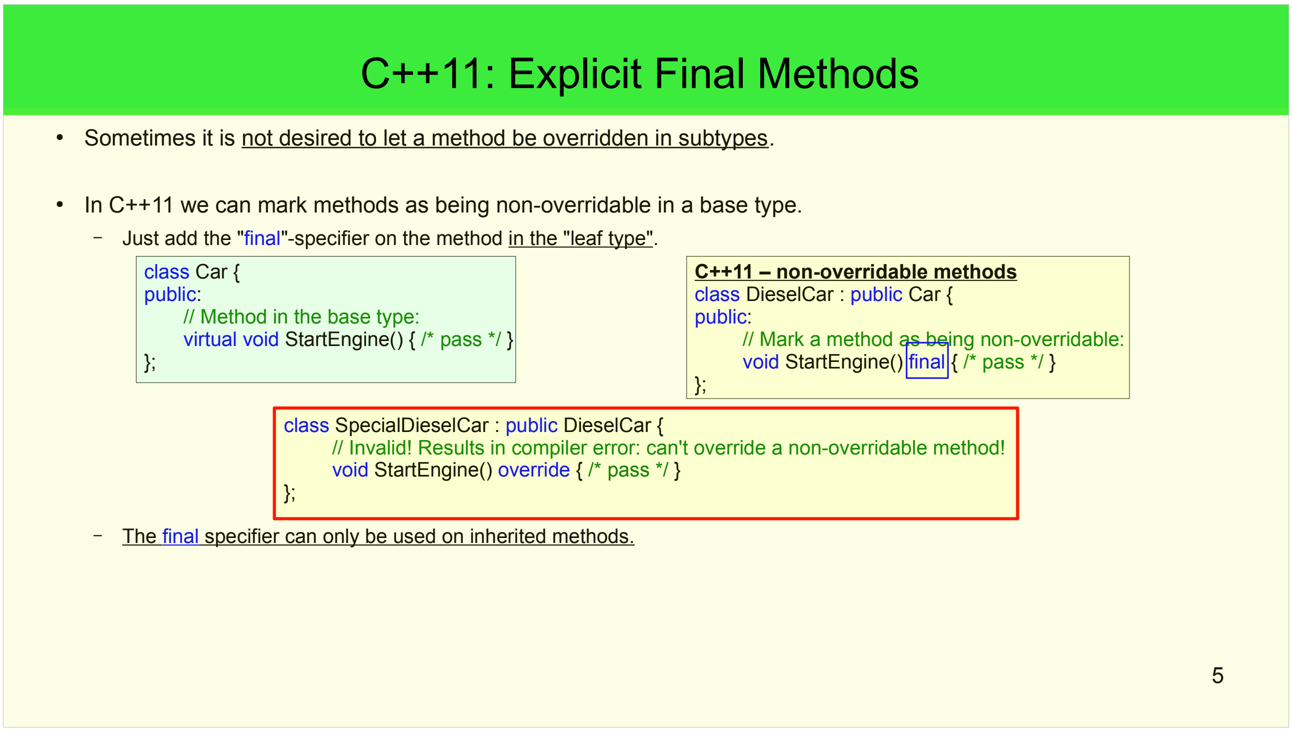
- C++11 allows explicit specification of overriding methods.
  - Just add the "override"-specifier to the method:

C++11 – explicitly overridden methods

```
class DieselCar : public Car {
public:
	// Invalid! Results in compiler error: doesn't override an inherited method!
	void StartEngine() override { /* pass */ }
};
```

  - The override specifier can only be used on inherited methods.

- 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. – The new C++11-way to mark overrides is the override-specifier, this mark is also checked by the compiler.
- The override keyword must not be repeated on a non-inline definition.



C++11: Explicit Final Methods

- Sometimes it is not desired to let a method be overridden in subtypes.

- In C++11 we can mark methods as being non-overridable in a base type.
  - Just add the "final"-specifier on the method in the "leaf type".

```
class Car {
public:
	// Method in the base type:
	virtual void StartEngine() { /* pass */ }
};
```

C++11 – non-overridable methods

```
class DieselCar : public Car {
public:
	// Mark a method as being non-overridable:
	void StartEngine() final { /* pass */ }
};
```

```
class SpecialDieselCar : public DieselCar {
	// Invalid! Results in compiler error: can't override a non-overridable method!
	void StartEngine() override { /* pass */ }
};
```

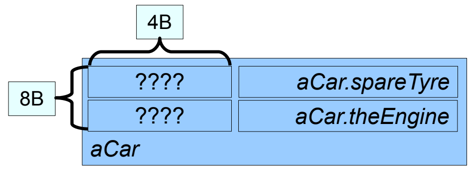
  - The final specifier can only be used on inherited methods.

- The final keyword must not be repeated on a non-inline definition.

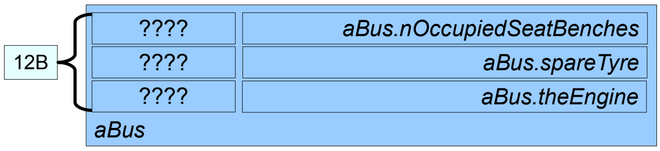
The Size of inherited Types

- When a UDT inherits from another UDT, their sizes sum up.
  - I.e. the sizes of all fields (esp. also private fields) contribute to the final sum.

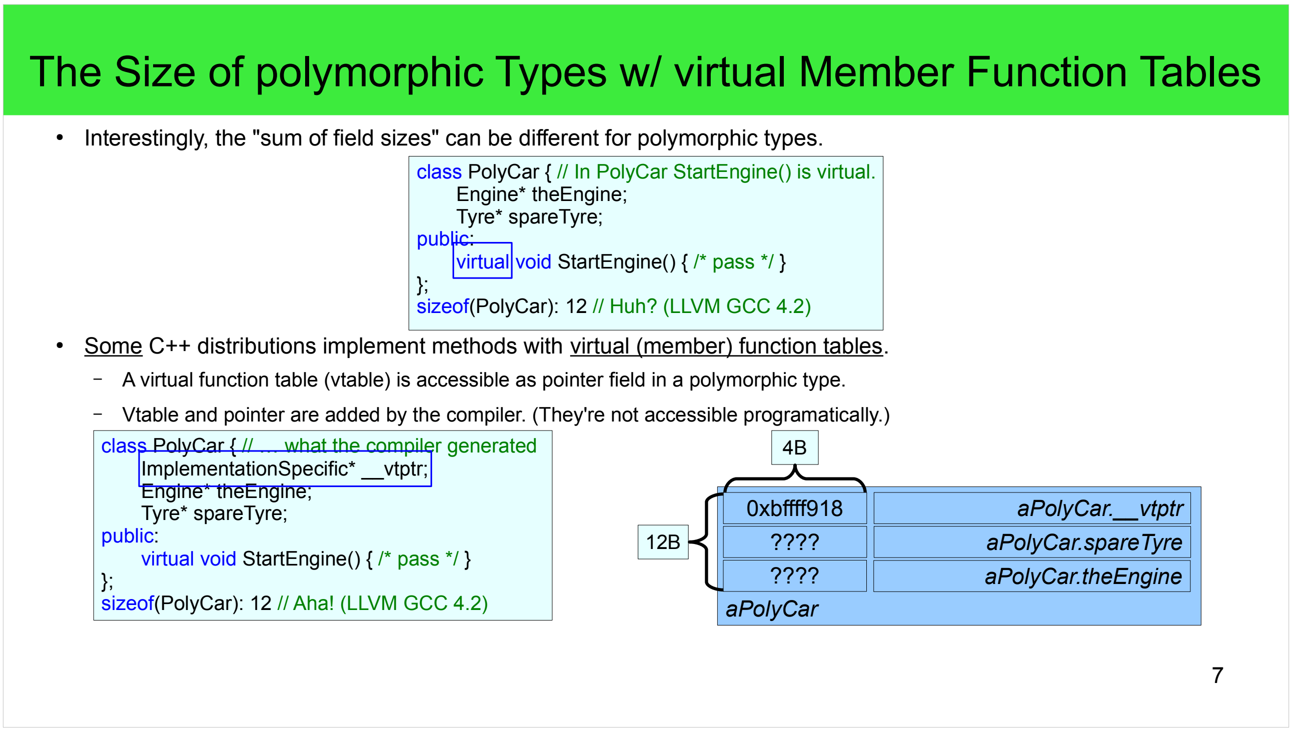
```
class Car { // UDTs Engine and Tyre elided.
	Engine* theEngine;
	Tyre* spareTyre;
public:
	void StartEngine() { /* pass */ }
};
sizeof(Car): 8
```



```
// Bus inherits from Car:
class Bus : public Car { // (members hidden)
	int nOccupiedSeatBenches;
public:
	/* pass */
};
sizeof(Bus): 12 // sizeof(int) + sizeof(Car)
```



- The sizes of polymorphic types could be different! Let's inspect this effect...



The Size of polymorphic Types w/ virtual Member Function Tables

- Interestingly, the "sum of field sizes" can be different for polymorphic types.

```
class PolyCar { // In PolyCar StartEngine() is virtual.
      Engine* theEngine;
      Tyre* spareTyre;
public:
      virtual void StartEngine() { /* pass */ }
};
sizeof(PolyCar): 12 // Huh? (LLVM GCC 4.2)
```

- Some C++ distributions implement methods with virtual (member) function tables.
  - A virtual function table (vtable) is accessible as pointer field in a polymorphic type.
  - Vtable and pointer are added by the compiler. (They're not accessible programatically.)

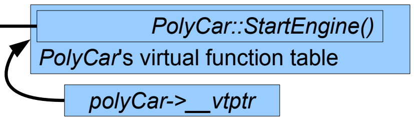
```
class PolyCar { // ... what the compiler generated
      ImplementationSpecific* __vtptr;
      Engine* theEngine;
      Tyre* spareTyre;
public:
      virtual void StartEngine() { /* pass */ }
};
sizeof(PolyCar): 12 // Aha! (LLVM GCC 4.2)
```

- The implementation of methods is compiler specific. The vtable implementation is used very often, it is the basis for the object system in the Microsoft technologies COM and .NET.
- Polymorphic types are no longer PODs, as their sizes and layouts are no longer predictable. The C++11 type trait *std::is_pod()* will return false for polymorphic types.

Virtual Member Function Tables – Oversimplification!

```
class PolyCar { // UDTs Engine and Tyre elided.
    Engine* theEngine;
    Tyre* spareTyre;
public:
    virtual void StartEngine() { /* pass */ }
};
```

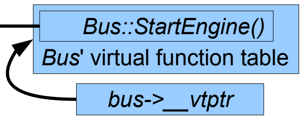
```
PolyCar* polyCar = new PolyCar;
polyCar->StartEngine();
```



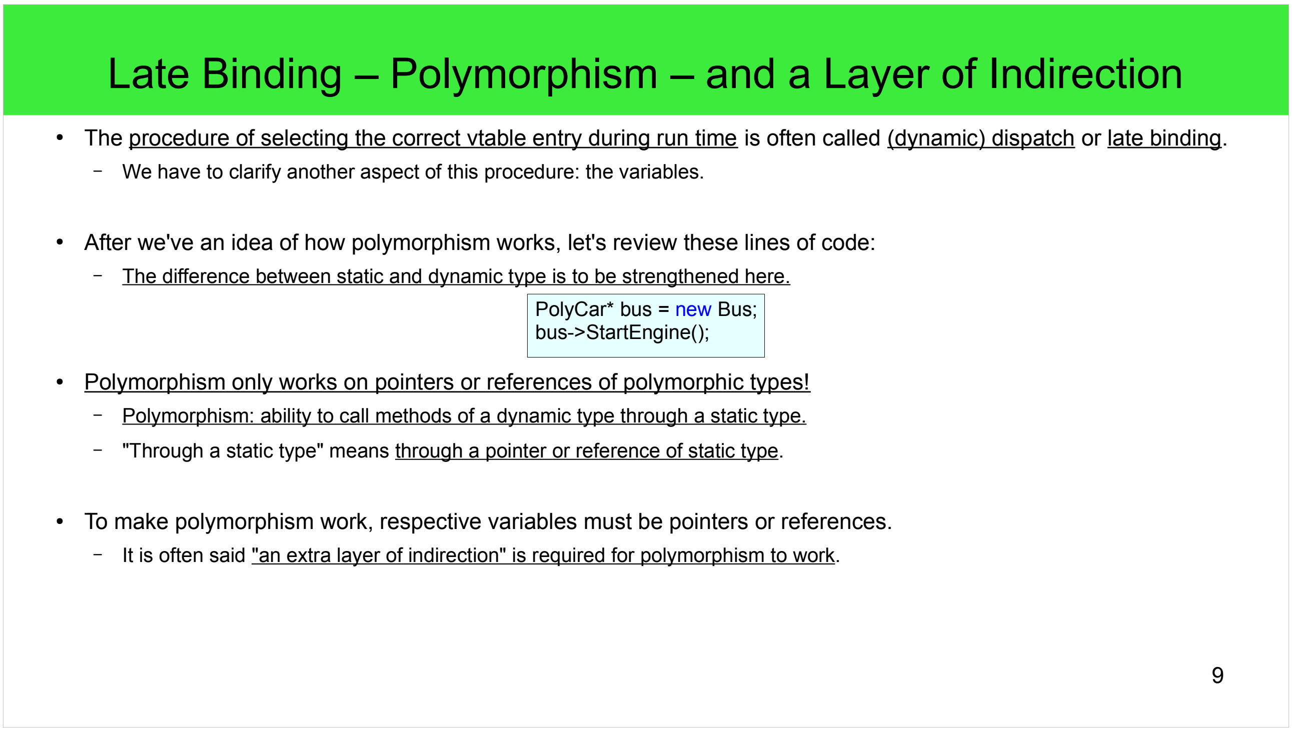
- For each polymorphic type a vtable is created.
  - It lists all methods of that type as function pointers.
  - Each instance of the polymorphic type has a pointer (*__vtptr*) to the vtable.

```
// Bus overrides StartEngine():
class Bus : public PolyCar { // (members hidden)
    int nOccupiedSeatBenches;
public:
    void StartEngine() { /* pass */ }
};
```

```
PolyCar* bus = new Bus;
bus->StartEngine();
```



- A derived type inherits the vtables well.
  - The compiler replaces inherited function pointers by pointers to overridden methods.
  - Non-overridden methods are left as inherited function pointers in the vtable.



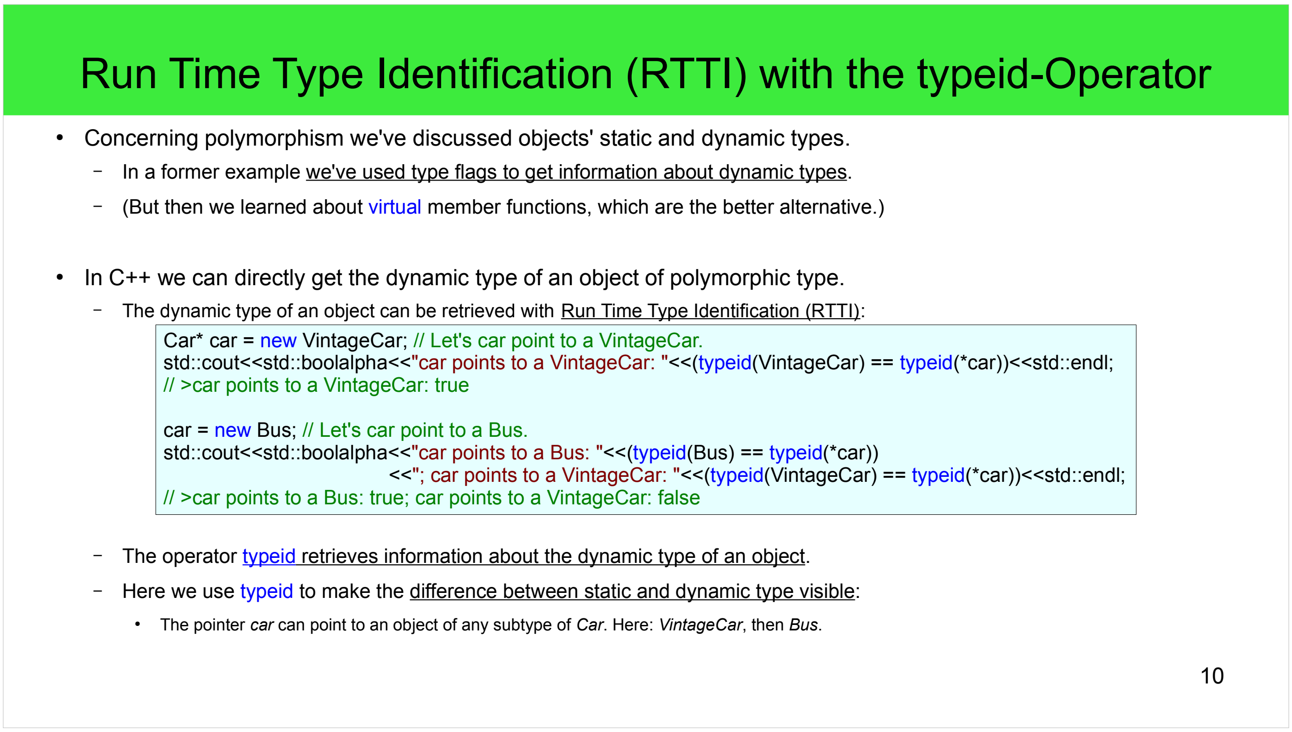
Late Binding – Polymorphism – and a Layer of Indirection

- The procedure of selecting the correct vtable entry during run time is often called (dynamic) dispatch or late binding.
  - We have to clarify another aspect of this procedure: the variables.
- After we've an idea of how polymorphism works, let's review these lines of code:
  - The difference between static and dynamic type is to be strengthened here.

```
PolyCar* bus = new Bus;
bus->StartEngine();
```

- Polymorphism only works on pointers or references of polymorphic types!
  - Polymorphism: ability to call methods of a dynamic type through a static type.
  - "Through a static type" means through a pointer or reference of static type.
- To make polymorphism work, respective variables must be pointers or references.
  - It is often said "an extra layer of indirection" is required for polymorphism to work.

- 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.



Run Time Type Identification (RTTI) with the typeid-Operator

- Concerning polymorphism we've discussed objects' static and dynamic types.
  - In a former example we've used type flags to get information about dynamic types.
  - (But then we learned about virtual member functions, which are the better alternative.)
- In C++ we can directly get the dynamic type of an object of polymorphic type.
  - The dynamic type of an object can be retrieved with Run Time Type Identification (RTTI):

```
Car* car = new VintageCar; // Let's car point to a VintageCar.
std::cout<<std::boolalpha<<"car points to a VintageCar: "<<(typeid(VintageCar) == typeid(*car))<<std::endl;
// >car points to a VintageCar: true

car = new Bus; // Let's car point to a Bus.
std::cout<<std::boolalpha<<"car points to a Bus: "<<(typeid(Bus) == typeid(*car))
                        <<"; car points to a VintageCar: "<<(typeid(VintageCar) == typeid(*car))<<std::endl;
// >car points to a Bus: true; car points to a VintageCar: false
```

  - The operator typeid retrieves information about the dynamic type of an object.
  - Here we use typeid to make the difference between static and dynamic type visible:
    - The pointer *car* can point to an object of any subtype of *Car*. Here: *VintageCar*, then *Bus*.

- The operator typeid returns an object of type *std::type_info* (declared in <typeinfo>).
  - *std::type_info* objects can not be created or assigned to from using typeid (*std::type_info*'s copy assignment and the cctor are both non-public).
  - The only interesting thing we can do with *std::type_info* is directly ==/!=-comparing the results of two typeid calls.
  - It is typically an error to use typeid to compare pointer types!
  - There is also a member function *std::type_info::name()* that returns a (arbitrary and non-portable) string representation of a type. This function must also be directly called on the result of the typeid operator.

Garage::TryToStartCar() with the typeid-Operator

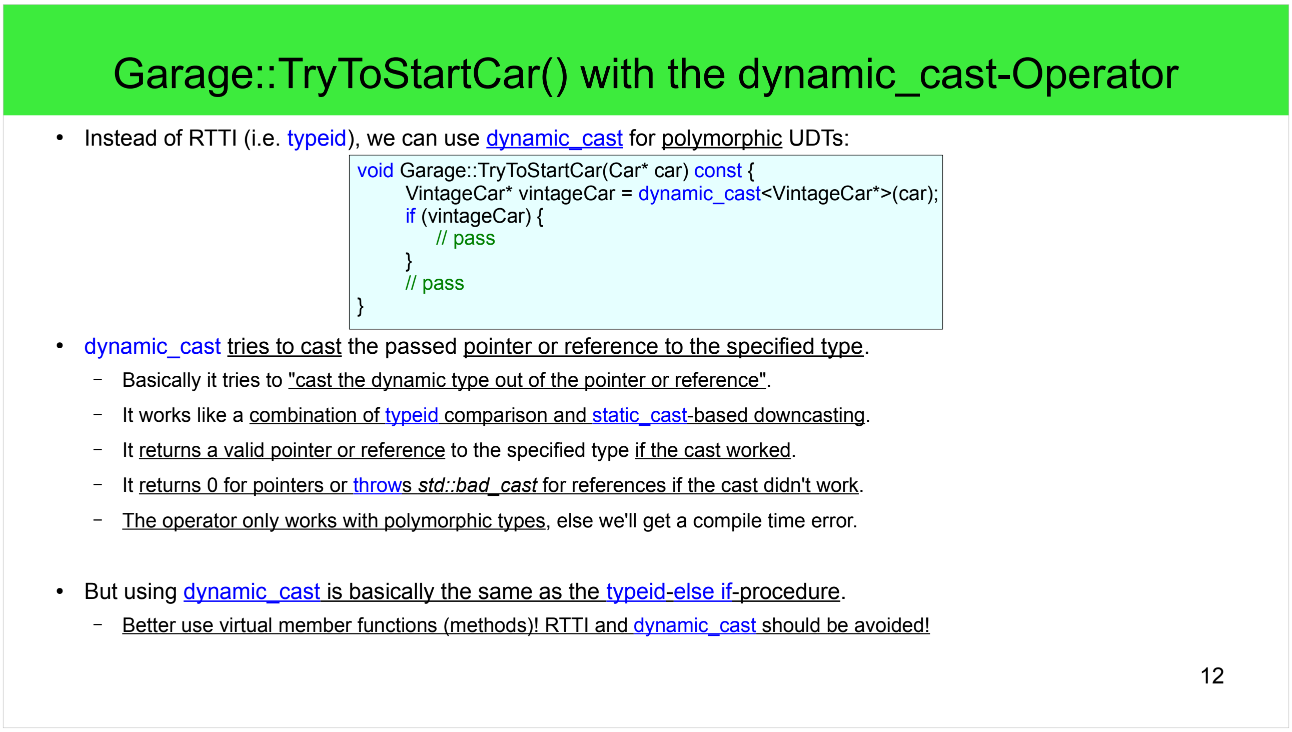
- It's possible to reimplement *Garage::TryToStartCar()* with RTTI:

```
void Garage::TryToStartCar(Car* car) const {
        if (typeid(VintageCar) == typeid(*car)) { // If car's dynamic type is VintageCar start it specially.
            VintageCar* vintageCar = static_cast<VintageCar*>(car); // Downcasting!
            if (!vintageCar->HasStarter()) {
                vintageCar->CrankUntilStarted();
            } else {
                vintageCar->StartEngine();
            }
        } else {
            car->StartEngine(); // Start other cars just by calling StartEngine().
        }
}
```

  - This implementation neglects the presence of methods, but uses RTTI and downcasts!
    - (We already learned that this is really bad: "pasta-object-orientation"!)

- With this implementation of *Garage::TryToStartCar()* we can start *fordTinLizzie*:

```
VintageCar* fordTinLizzie = new VintageCar(1909);
joesStation.TryToStartCar(fordTinLizzie);      // Ok! TryToStartCar() will pick the correct start-algorithm
                                               // depending on the run time type!
```



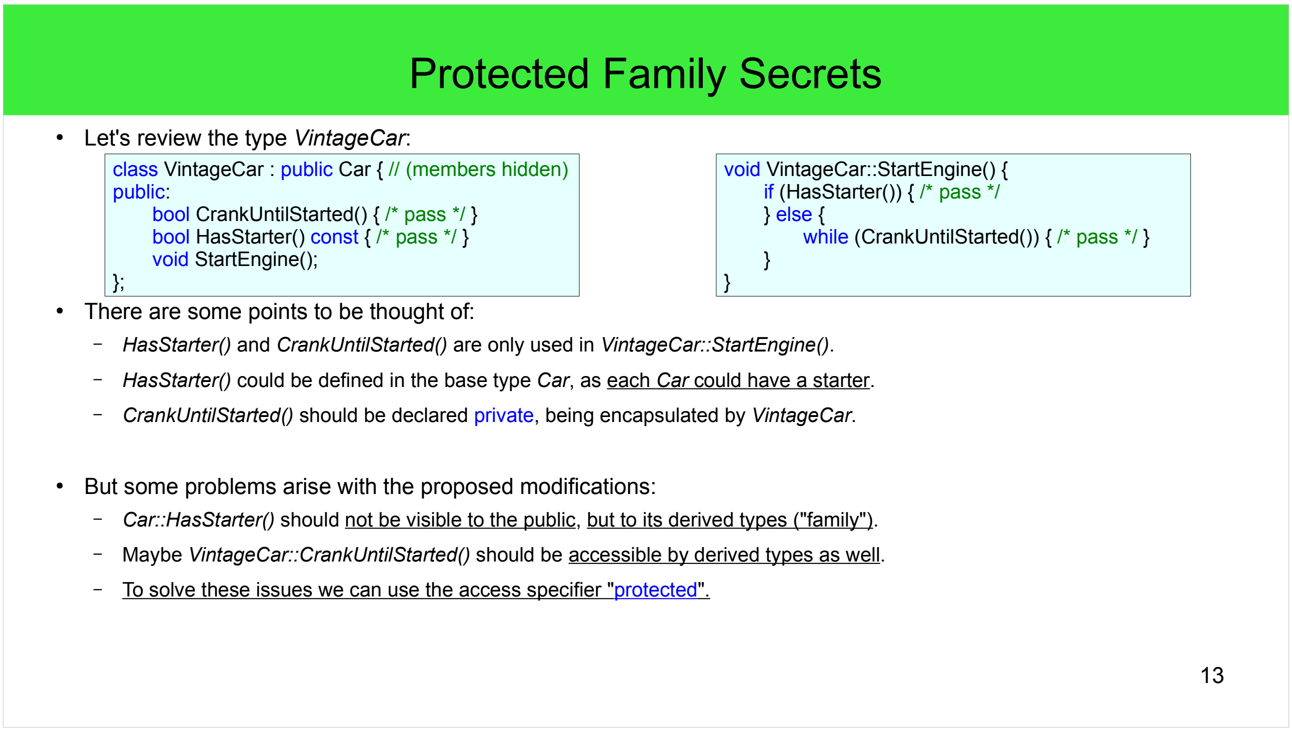
Garage::TryToStartCar() with the dynamic_cast-Operator

- Instead of RTTI (i.e. typeid), we can use dynamic_cast for polymorphic UDTs:

```
void Garage::TryToStartCar(Car* car) const {
        VintageCar* vintageCar = dynamic_cast<VintageCar*>(car);
        if (vintageCar) {
            // pass
        }
        // pass
}
```

- dynamic_cast tries to cast the passed pointer or reference to the specified type.
  - Basically it tries to "cast the dynamic type out of the pointer or reference".
  - It works like a combination of typeid comparison and static_cast-based downcasting.
  - It returns a valid pointer or reference to the specified type if the cast worked.
  - It returns 0 for pointers or throws *std::bad_cast* for references if the cast didn't work.
  - The operator only works with polymorphic types, else we'll get a compile time error.
- But using dynamic_cast is basically the same as the typeid-else if-procedure.
  - Better use virtual member functions (methods)! RTTI and dynamic_cast should be avoided!

- In opposite to static_cast, dynamic_cast is type-checked at run time! It's type safe to use dynamic_cast for downcasting!
  - dynamic_cast is the only cast that can't be expressed with the older cast syntaxes.
- The explanation why we didn't use const& as parameter types here (kind of breaking the rule we've defined lately) is, that we'd have to deal with *std::bad_cast* exceptions in this case (instead of 0-pointers). – And we didn't discuss exceptions yet.
- There is the saying "polymorphism is always better than branching". In this example the usage of dynamic type analysis (with RTTI or dynamic_cast) and if-else cascades would be the "branching". – This approach requires to add new dynamic types to be potentially handled in *Garage::TryToStartCar()* and this is a disapproving approach; always prefer polymorphism! – Design patterns, which are often based on polymorphism, help to implement even complex problems without dynamic type analysis.



Protected Family Secrets

- Let's review the type *VintageCar*:

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

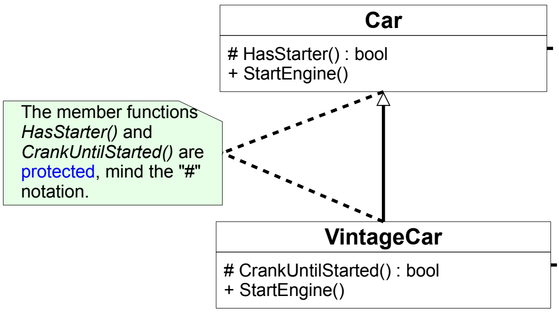
```
void VintageCar::StartEngine() {
      if (HasStarter()) { /* pass */
      } else {
            while (CrankUntilStarted()) { /* pass */ }
      }
}
```

- There are some points to be thought of:
  - *HasStarter()* and *CrankUntilStarted()* are only used in *VintageCar::StartEngine()*.
  - *HasStarter()* could be defined in the base type *Car*, as each *Car* could have a starter.
  - *CrankUntilStarted()* should be declared private, being encapsulated by *VintageCar*.
- But some problems arise with the proposed modifications:
  - *Car::HasStarter()* should not be visible to the public, but to its derived types ("family").
  - Maybe *VintageCar::CrankUntilStarted()* should be accessible by derived types as well.
  - To solve these issues we can use the access specifier "protected".

- We can also apply protected inheritance in C++.

Protected Member Functions

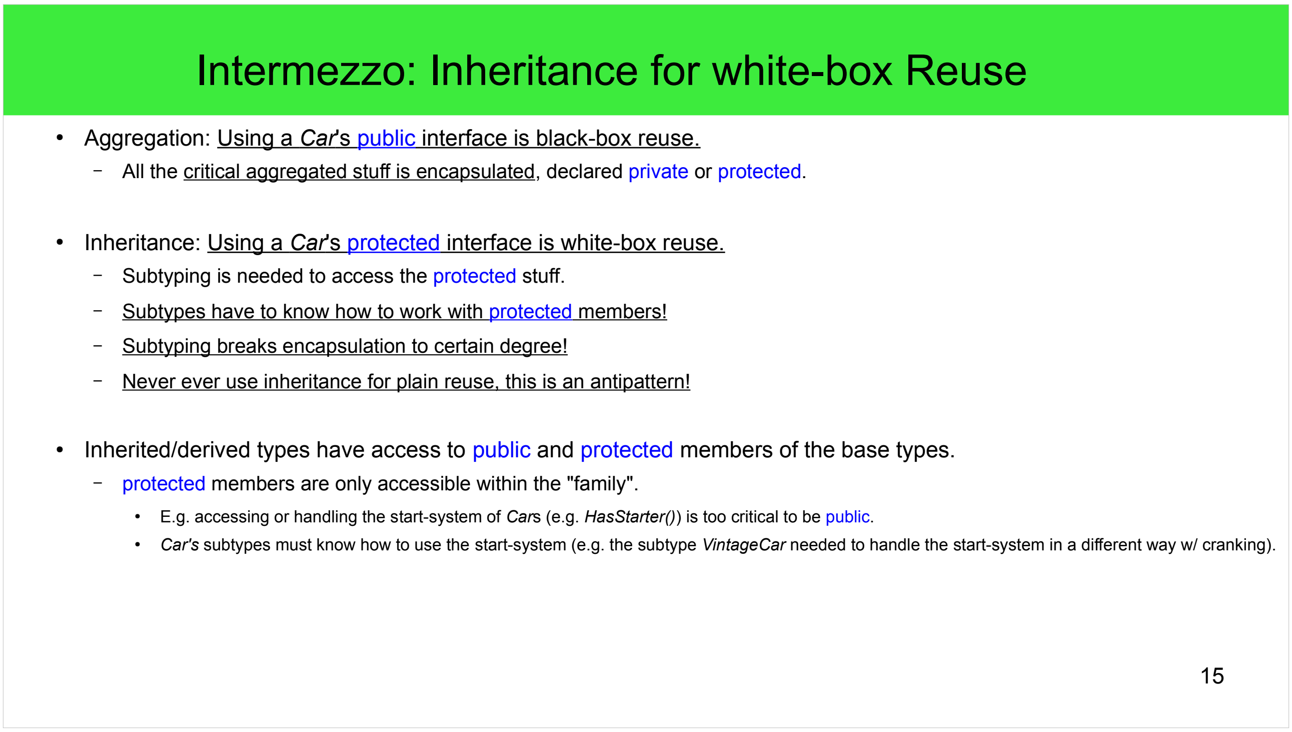
- Let's redesign *Car*/*VintageCar*:
  - Move *HasStarter()* to the type *Car* and mark it as being protected.
  - Make *VintageCar::CrankUntilStarted()* protected as well.
  - Then both member functions are only visible in the "family", not to the public.



The member functions *HasStarter()* and *CrankUntilStarted()* are protected, mind the "#" notation.

```
class Car { // (members hidden)
protected:
	bool HasStarter() const { /* pass */ }
public:
	virtual void StartEngine() { /* pass */ }
};
```

```
class VintageCar : public Car { // (members hidden)
protected:
	bool CrankUntilStarted() { /* pass */ }
public:
	void StartEngine() {
		if (HasStarter()) { /* pass */
		} else {
			while (CrankUntilStarted()) { /* pass */ }
		}
	}
};
```



Intermezzo: Inheritance for white-box Reuse

- Aggregation: Using a *Car*'s public interface is black-box reuse.
  - All the critical aggregated stuff is encapsulated, declared private or protected.
- Inheritance: Using a *Car*'s protected interface is white-box reuse.
  - Subtyping is needed to access the protected stuff.
  - Subtypes have to know how to work with protected members!
  - Subtyping breaks encapsulation to certain degree!
  - Never ever use inheritance for plain reuse, this is an antipattern!
- Inherited/derived types have access to public and protected members of the base types.
  - protected members are only accessible within the "family".
    - E.g. accessing or handling the start-system of *Cars* (e.g. *HasStarter()*) is too critical to be public.
    - *Car's* subtypes must know how to use the start-system (e.g. the subtype *VintageCar* needed to handle the start-system in a different way w/ cranking).

- In fact the protected members of our types should be as good documented (e.g. via comments) as the public members to make white-box reuse possible!

Abstract Types

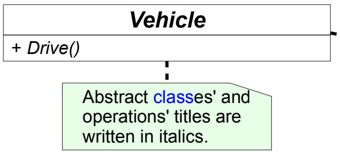
- There are types that are too abstract to have instances.
  - This means that some methods can not provide suitable (default) implementations.

- E.g. a new UDT *Vehicle* is more abstract than *Car*:
  - *Vehicle* is too abstract of its own, we should not assume that it has an *Engine*, or even a *SpareTyre*... it could be left away. (*StartEngine()* also makes no sense!)
  - More concrete subtypes (as *Car*) could fill, what we left away in *Vehicle*.
  - *Vehicle* is even too abstract to provide a default implementation for e.g. *Drive()*.

- Abstract types in C++:
  - UDTs can provide methods without implementation.
  - These methods are called abstract methods.
  - Abstract methods are defined as pure virtual member functions in C++.
  - Abstract types allow to abstract a type's usage from its implementation.
  - Now let's implement the abstract type *Vehicle*...

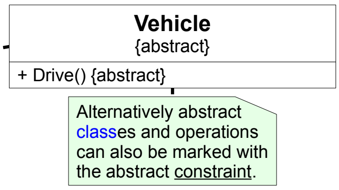
The abstract Type Vehicle

- A C++ implementation of the abstract type *Vehicle* might look like this:



Abstract classes' and operations' titles are written in italics.

```
// The abstract UDT Vehicle:
class Vehicle {
public:
        // A pure virtual member function:
        virtual void Drive() = 0;
};
```



Alternatively abstract classes and operations can also be marked with the abstract constraint.

- An inheriting type can override/implement abstract methods.

```
// The concrete UDT Car:
class Car : public Vehicle { // (members hidden)
public:
        // Override Drive():
        void Drive() {
             std::cout<<"zooming off..."<<std::endl;
        }
};
```

```
// The abstract UDT Bike:
class Bike : public Vehicle {
public:
        // Doesn't override Drive():
        void ChangeUp() { /* pass */ }
};
```

  - Inheriting UDTs that don't implement all abstract methods become itself abstract UDTs!
    - I.e. *Bike* is an abstract UDT!

Implementation and Usage of abstract Types

- Implementation side of abstract types:
  - Assign 0 to the method declaration in the top level abstract type.
    - Inherited abstract methods must be overridden in concrete types.
    - Inherited abstract methods need not to be overridden in abstract types.

- Usage side of abstract types:
  - No instances of abstract UDTs can be created; *Vehicle* is just a too abstract type.

```
Vehicle* vehicle = new Vehicle; // Invalid! Instance of abstract UDT can't be created.
```

  - The only idea behind abstract types is to support polymorphism.

```
// The concrete UDT RacingBicycle:
class RacingBicycle : public Bike {
public:
        void Drive() {
             std::cout<<"woosh..."<<std::endl;
        }
};
```

```
void Move(Vehicle* vehicle) {
        vehicle->Drive();
}
Move(new Car);
// >zooming off...
Move(new RacingBicycle);
// >woosh...
```

  - The is-a relationship holds true: A *Car* is a *Vehicle* and a *RacingBicycle* is a *Vehicle*.
    - The substitution works nicely with abstract types! – We're going to revisit this idea!

Abstract default Implementations

- Abstract methods are allowed to provide a default implementation.
  - The implementation must be provided as non-inline member function definition.

```
// The abstract UDT Vehicle:
class Vehicle {
public:
        // A pure virtual member function:
        virtual void Drive() = 0;
};
```

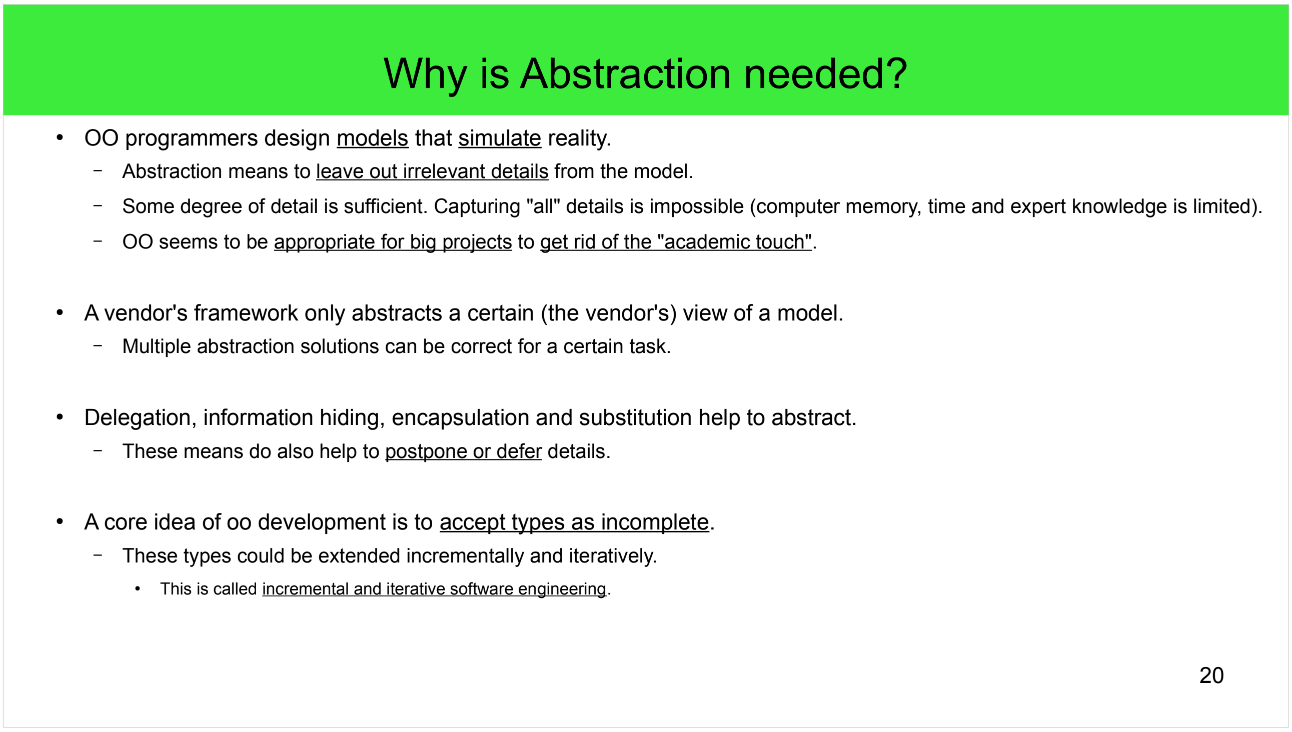
```
// A default implementation for inheriting types:
void Vehicle::Drive()  { // Mind: it must be a non-inline member function!
        std::cout<<"driving..."<<std::endl;
}
```

- Inheriting types are allowed to call this default implementation like so:

```
// Override Drive():
void Car::Drive() {
        Vehicle::Drive(); // Calls inherited default implementation:
        std::cout<<"zooming off..."<<std::endl;
}
```

  - Calling *Car::Drive()* shows that it works:

```
Car car;
car.Drive();
// >driving...
// >zooming off...
```



Why is Abstraction needed?

- OO programmers design models that simulate reality.
  - Abstraction means to leave out irrelevant details from the model.
  - Some degree of detail is sufficient. Capturing "all" details is impossible (computer memory, time and expert knowledge is limited).
  - OO seems to be appropriate for big projects to get rid of the "academic touch".
- A vendor's framework only abstracts a certain (the vendor's) view of a model.
  - Multiple abstraction solutions can be correct for a certain task.
- Delegation, information hiding, encapsulation and substitution help to abstract.
  - These means do also help to postpone or defer details.
- A core idea of oo development is to accept types as incomplete.
  - These types could be extended incrementally and iteratively.
    - This is called incremental and iterative software engineering.

- Abstraction from lat. abstrahere: "to remove something".
- What does "incremental and iterative" mean?
  - Incremental: the engineering will be performed in steps.
  - Iterative: some parts of the engineering process will be repeated to improve these aspects.

Calling the Ctors of base Classes

- If the base types have dctors they'll be called by the ctors of subtypes:

```
class Car { // (members hidden)
public:
        Car() {
            std::cout<<"Car::Car()"<<std::endl;
        }
};
```

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

```
Bus bus;
// >Car::Car() // Calls Car's dctor implicitly.
```

- If a base type doesn't provide a dctor, nothing can be implicitly called of course!

```
class Car { // Car w/o dctor!
public:
        Car(double power) { /* pass */ }
};
```

```
Bus bus; // Invalid! Base type Car provides no dctor.
```

  - Instead we have to call another ctor of the base type in the subtype's ctor explicitly.
    - Syntactically we have to call ctors of the base types in the initializer list of a subtype's ctor:

```
class Bus : public Car {
public:
        // Ctor, which calls one of Car's ctors.
        Bus(double power) : Car(power) { /* pass */ }
};
```

```
Bus bus(320); // Fine!
```

Inheritance and Destructors

- A derived UDT accesses the dtors of its base types on destruction.

```
class A {
public:
        ~A() {
            std::cout<<"~A"<<std::endl;
        }
};
```

```
class B : public A {
public:
        ~B() {
            std::cout<<"~B"<<std::endl;
        }
};
```

  - When the dtor of an object is called, the base types' dtors are also called:

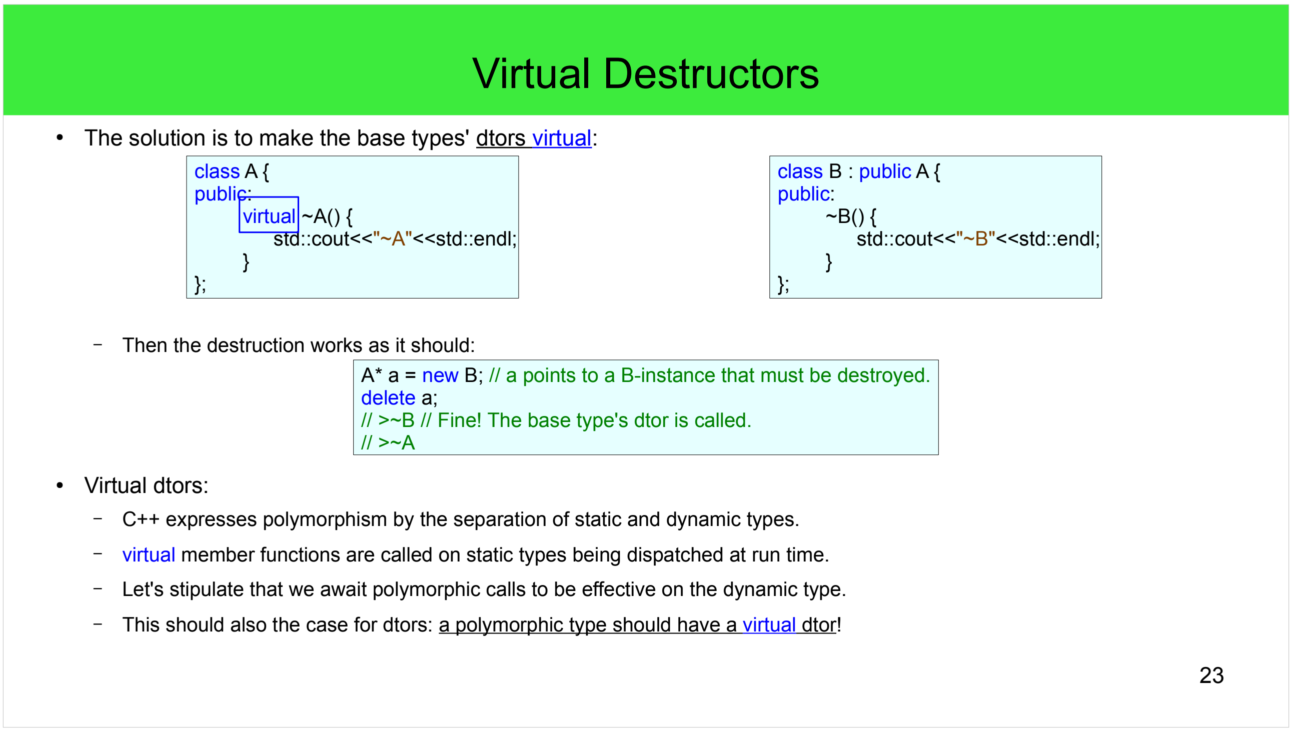
```
{ // With RAII:
        B b;
}
// >~B // The dtors are called from the most
// >~A // special to the most general type.
```

```
// With dynamic allocation:
B* b = new B;
delete b;
// >~B
// >~A
```

- But there is a problem: it doesn't work on dynamic types!

```
A* a = new B; // a points to a B-instance that must be destroyed.
delete a;
// >~A // Oops! Only the dtor of the static type is called!
```

  - Obviously dynamic dispatch doesn't work on dtors!
  - But the solution is simple: we have to make the base types' dtors polymorphic!



Virtual Destructors

- The solution is to make the base types' dtors virtual:

```
class A {
public:
        virtual ~A() {
            std::cout<<"~A"<<std::endl;
        }
};
```

```
class B : public A {
public:
        ~B() {
            std::cout<<"~B"<<std::endl;
        }
};
```

  - Then the destruction works as it should:

```
A* a = new B; // a points to a B-instance that must be destroyed.
delete a;
// >~B // Fine! The base type's dtor is called.
// >~A
```

- Virtual dtors:
  - C++ expresses polymorphism by the separation of static and dynamic types.
  - virtual member functions are called on static types being dispatched at run time.
  - Let's stipulate that we await polymorphic calls to be effective on the dynamic type.
  - This should also the case for dtors: a polymorphic type should have a virtual dtor!

- We can also have pure virtual dtors.

Inheritance and Operators

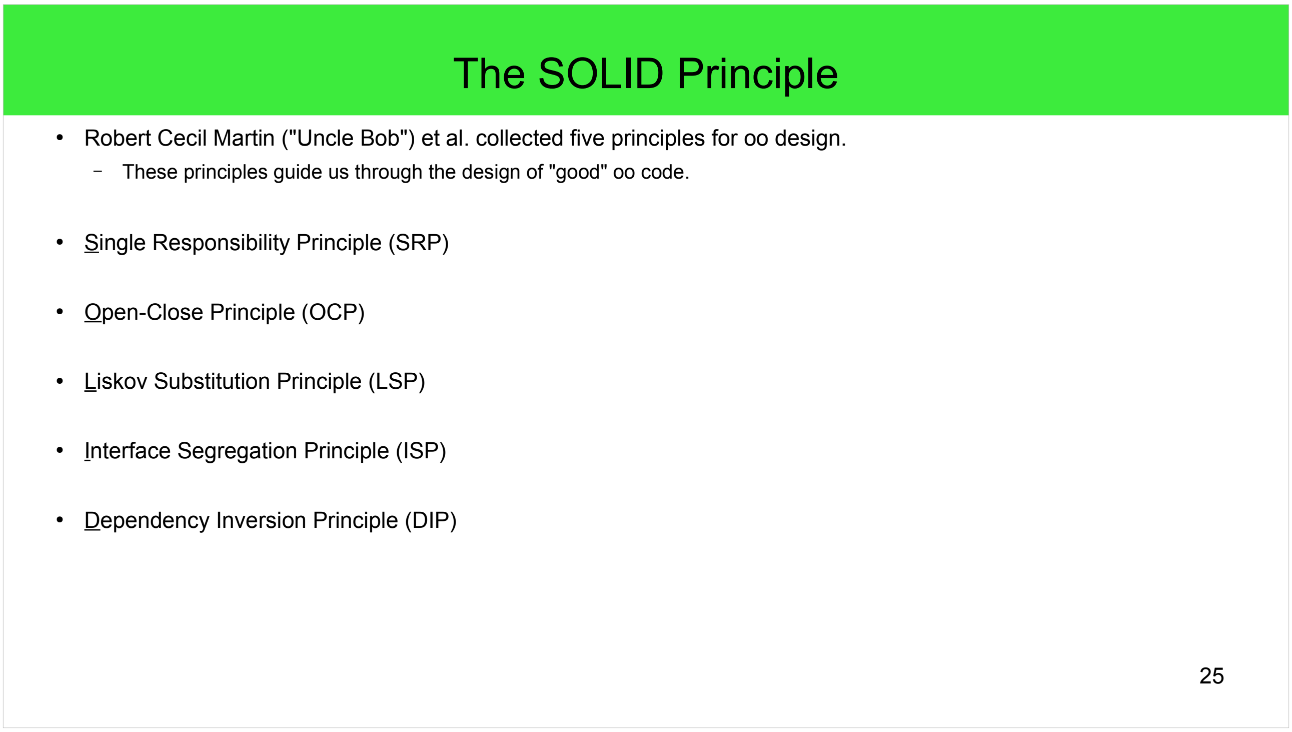
- Derived types don't inherit all members of its base types, because it would introduce inconsistencies. Not inherited are:
  - ctors and the non-virtual dtor,
  - assignment operators and
  - friends.

- Not inherited just means, that those members are no part of the public interface of the derived type!
  - But the implementations of those members are still "callable" in a derived type! – So we can reuse them!

- However, usually the operators of base types are called in the implementation of those in the derived type.
  - This example shows the idiomatic "delegation" of operator calls to reuse the functionality of base types.

```
class SpecialPerson : public Person { // (types hidden)
public:
      SpecialPerson(const SpecialPerson& original) : Person(original) { // Just forward to the cctor of the base type.
      }

      SpecialPerson& operator=(const SpecialPerson& rhs) {
          if (this != &rhs) {
               Person::operator=(rhs);  // Just call copy-operator= of the base type.
          }                              // Mind, that the result of Person::operator= can't be returned, because the
          return *this;                  // return type is too general! We call Person::operator= only for the side effect
      }                                  // on this!
};
```



The SOLID Principle

- Robert Cecil Martin ("Uncle Bob") et al. collected five principles for oo design.
  - These principles guide us through the design of "good" oo code.
- Single Responsibility Principle (SRP)
- Open-Close Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

- SRP: Types should have only one reason to change. This requirement also applies to functions.
- OCP: Open types for extension, but close them for modification, i.e. extend them w/o modification of the core code, rather code will be added (e.g. via polymorphism). If we need to modify a type's interface, the calling code needs to be modified (ripple effect), which is expensive. Separate things that change, from others that don't (i.e. find the vector of change to drive design patterns).
- LSP: Exploit subtyping to put the substitution principle into effect.
- ISP: Use small type interfaces to which callers depend on, so that changes in other type interfaces don't bother them. Smaller interfaces are more stable than bigger ones.
- DIP: Let types only depend on abstract types (the contracts), never on concrete types (the implementation) to reduce coupling. Layers of lower abstraction (more concrete type) depend on layers of higher abstraction (more abstract types), never vice versa.
- The SOLID principle enumerates a set of guidelines, not laws.
- Additionally, there exists the principle of Command Query Separation (CQS) formulated by Bertrand Meyer: Do getters only query values, but don’t change state? Do setters not return values?

Thank you!