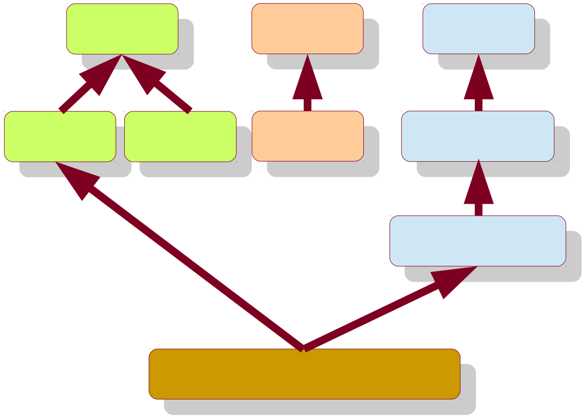
Inheritance and Virtual Functions



Rupesh Nasre.
IIT Madras

Raisoni
July 2020

Agenda

- Inheritance Basics
- Class Hierarchy
- Access Qualifiers
- Virtual Functions
- Pure Virtual Functions
- Multiple Inheritance

Background

- Classes, Objects
  - A class is a type.
  - An object is its instance.
- Constructors, Destructors
  - Constructors are automatically called on object creation.
  - Destructors are automatically called on object destruction.
- Access Qualifiers
  - public: accessible to the world
  - protected: accessible to children, grandchildren, …
  - private: accessible to self

Reuse

- In large software systems, it is not a good idea to start from scratch every time.
  - We should reuse the existing functionality and build upon it.
- Reuse in procedural style is achieved using function libraries.
- OOP provides us with another interesting way to reuse the functionality of a class.
  - An apple is a fruit, and so is orange.

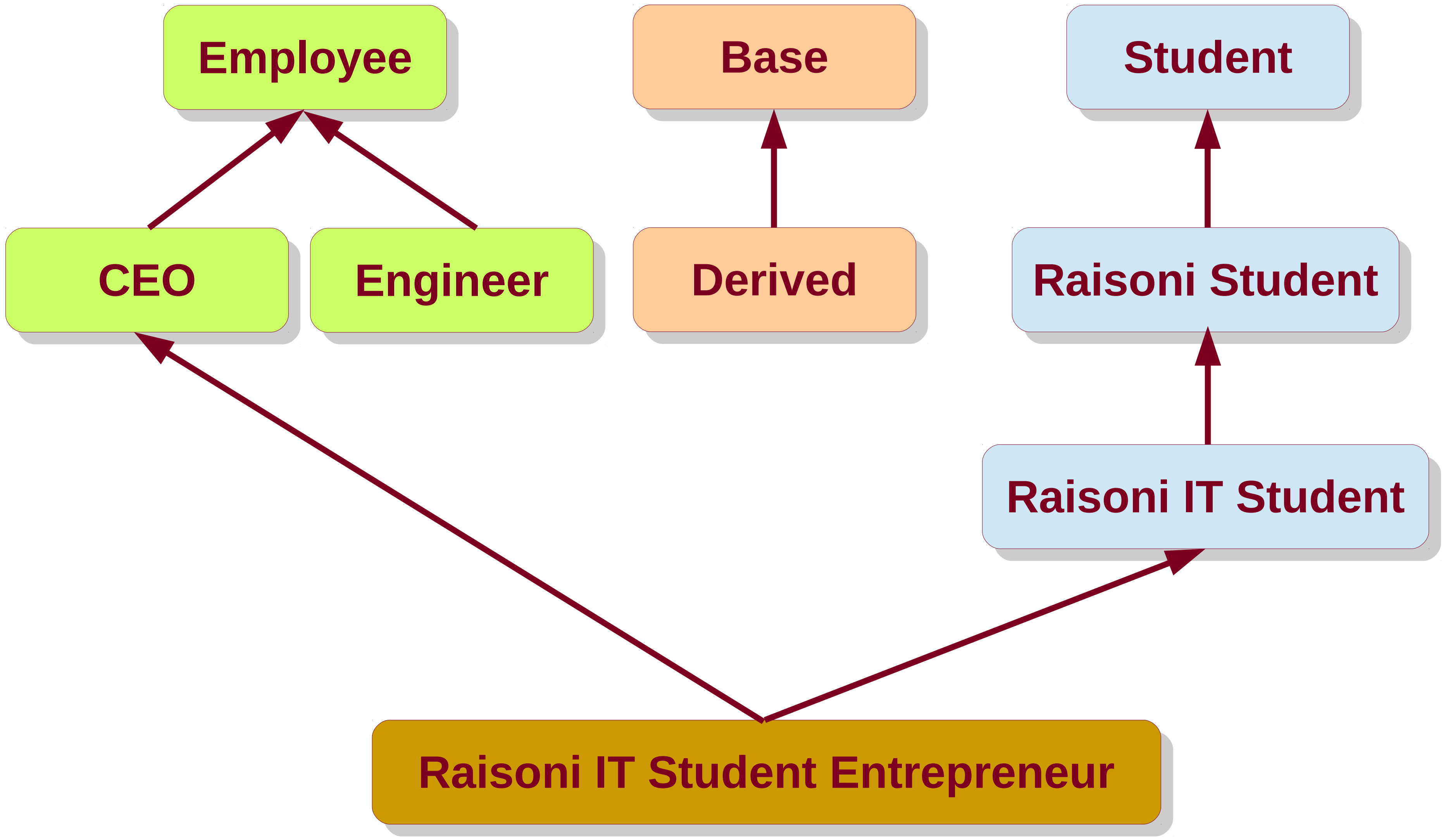
Inheritance

- **Base class**: Parent class with some functionality.
- **Derived class**: Child class which inherits properties of the parent class and defines its own.
  - It would also add other functionality.
  - Similar to how we inherit styles / behavior of our parents.

```
class Base {
public:
    void fun() {
        cout << “in base::fun.\n”;
    }
protected:
    int n;
};
class Derived:public Base {
public:
    void some() {
        n = 10;
        cout << “in Deri::some\n”;
    }
};
int main() {
    Derived d;
    d.fun();
    d.some();
}
```

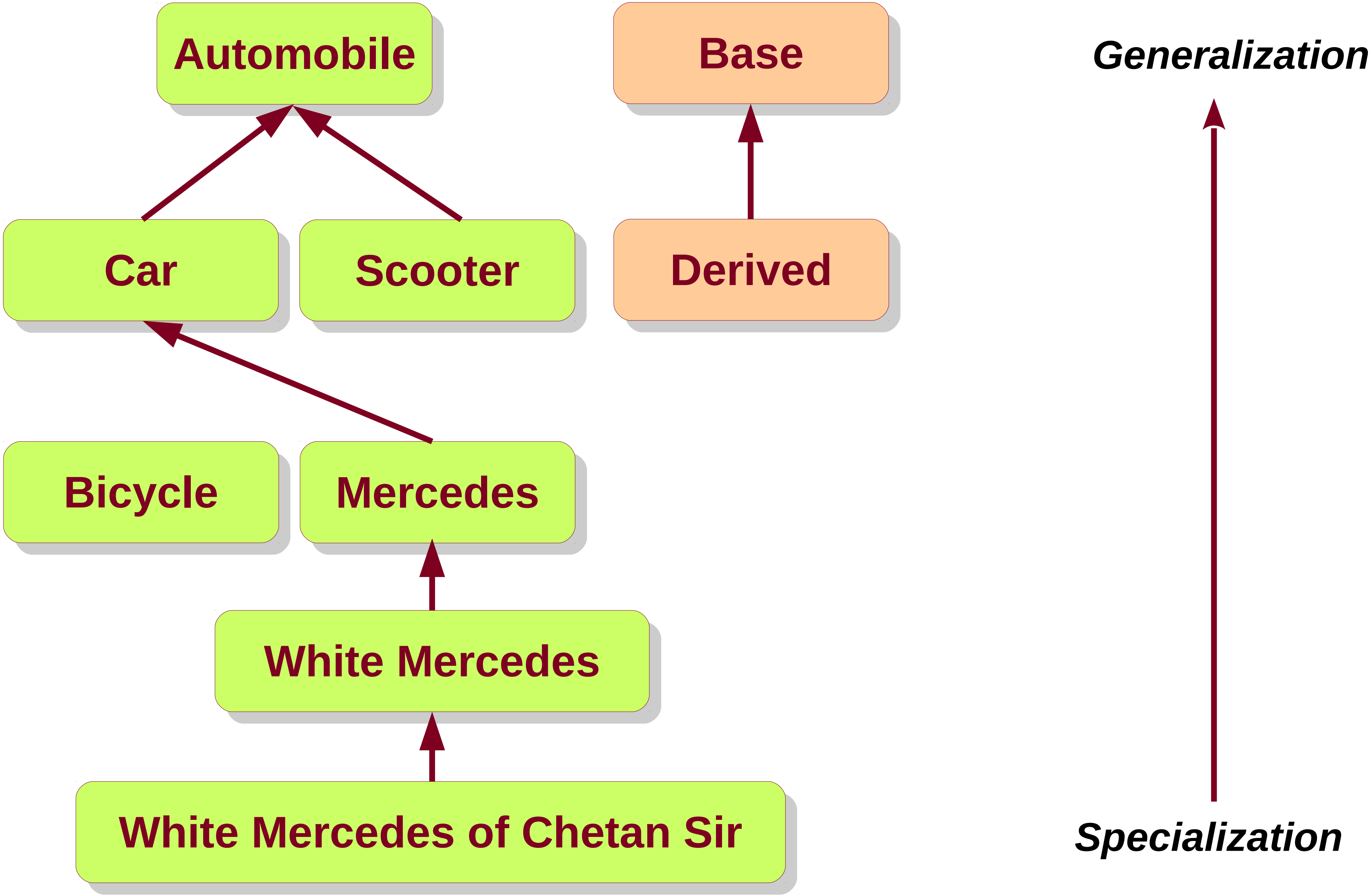
Source: 2.cpp

Derivation



Employee
Base
Student
CEO
Engineer
Derived
Raisoni Student
Raisoni IT Student
Raisoni IT Student Entrepreneur

Find Derivation



Automobile

Base

Generalization

Car

Scooter

Derived

Bicycle

Mercedes

White Mercedes

White Mercedes of Chetan Sir

Specialization

What all is inherited?

- An object of a derived class has stored in it all the fields of the base type.
- An object of the derived type can use the methods of the base type.
- But
  - Derived class needs its own constructor(s)
  - Appropriate base constructor needs to be invoked explicitly (otherwise, default is executed if exists)
  - Need to respect the access permissions

Source: 3.cpp

Access Permissions

- A derived class method can access
  - All public member functions and fields of base
  - All protected member functions and fields of base
  - All methods and fields of itself
- A derived class method cannot access
  - Any private methods or fields of base
  - Any protected or private members of any other class

	public	protected	private
class	✓	✓	✓
children	✓	✓	×
rest	✓	×	×

Constructors

Decides default visibility of members from Base in Derived.

- A derived class constructor needs to call a specific base class constructor explicitly.
- This cannot be done using an executable instruction in the body of the constructor.
- Base class object is constructed first.

```
class Base {
public:
	Base(int r) { ... }
};
class Derived: public Base {
public:
	Derived(int x, int y)
	: Base(x) {
		...
	}
};
```

Destructors

- Destructors get called in the reverse order than the constructors.
- First derived class, then base class destructor
- A special consideration is required when a Base class pointer / reference points to a derived class object, and is deleted.

```
class Base {
public:
	~Base() {cout << “~Base\n”; }
};
class Derived: public Base {
public:
	~Derived() {cout<< “~Derived\n”;}
};
int main() {
	Derived d;
	return 0;
}
```

```
$ g++ file.cpp; a.out
~Derived
~Base
```

Pointers and Inheritance

- C++ has quite strong rules towards types.
- Student * pointer cannot point to Orange class object.
- However, a base class pointer can point to derived class object.
- Can access public members of base.

```
class Base {
...
};
class Derived:public Base {
...
};
int main() {
        Base *b = new Derived();
        delete b;
        return 0;
}
```

Pointers and Inheritance

- Such a mechanism is helpful in keeping track of all objects derived from the same class together.
- This way, we can call appropriate methods of different derived classes with the same pointer.
- Otherwise, we would be forced to keep all objects in multiple arrays (think C).

```
std::vector<Base *> allobj;
Base *a[100];
```

```
for (it = allDrinks.begin();
      it != allDrinks.end();
      ++it) {
     it->createOneCup();
}
```

```
for (it = allShapes.begin();
      it != allShapes.end();
      ++it) {
     it->draw();
}
```

Pointers and Inheritance

- Why do we need new?
  - Unlike malloc, new calls the constructor.
  - Unlike free, delete calls the destructor.
- Deleting a derived object automatically calls derived destructor and then the base destructor.
- **However**, deleting a base pointer pointing to derived object calls only base destructor.

```
class Base {
...
};
class Derived:public Base {
...
};
int main() {
        Base *b = new Derived();
        delete b;
        return 0;
}
```

Source: 4.cpp

Pointers and Inheritance

- Deleting a base pointer pointing to derived object calls only base destructor.
- If you want to call the destructor of the derived class (and then base class) in such a case, then you need to mark the base destructor virtual.

```
class Base {
…
virtual ~Base();
};
class Derived:public Base {
...
};
int main() {
        Base *b = new Derived();
        delete b;
        return 0;
}
```

Classwork: Source 5.cpp

Function Polymorphism: Pointers

- A derived class can redefine a method from the base class.
- If their signatures are the same, derived class method hides the base class method.
- A base class pointer calls the base method, while a derived class pointer calls the derived method.
- A base pointer pointing to derived class calls the base method.

```
class Base {
        …
        void fun();
};
class Derived: public Base {
        void fun();
};
int main() {
        Base *b = new Derived();
        b->fun();
        ...
}
```

Function Polymorphism: Iterators

- We expect the iterator to invoke methods of the appropriate types, square->draw() and circle->draw and triangle->draw, etc.
- But iterator has a pointer to the base type Shape *.
- How would it invoke the function of the derived class?

```
std::vector<Base *> allobj;
Base *a[100];
```

```
for (it = allDrinks.begin();
        it != allDrinks.end();
        ++it) {
       it->createOneCup();
}
```

```
for (it = allShapes.begin();
        it != allShapes.end();
        ++it) {
       it->draw();
}
```

Virtual Functions

- We expect the iterator to invoke methods of the appropriate types, square->draw() and circle->draw and triangle->draw, etc.
- But iterator has a pointer to the base type Shape *.
- How would it invoke the function of the derived class?

```
class Shape {
public:
      virtual void draw();
};
class Circle: public Shape {
public:
      void draw();
};
```

```
for (it = allShapes.begin();
       it != allShapes.end();
       ++it) {
      it->draw();
}
```

Virtual Functions

- If a function is virtual in the base class, it indicates that a derived class may want to override it.
- When a virtual method is invoked using a base class pointer, appropriate version of the method is invoked.

```
class Shape {
public:
    virtual void draw();
};
class Circle: public Shape {
public:
    void draw();
};
```

```
for (it = allShapes.begin();
      it != allShapes.end();
      ++it) {
    it->draw();
}
```

Binding

- Consider the following code.

```
Base *b;
if (input < 10)
  b = new Base();
else
  b = new Derived();

b->fun();
```

- How does the compiler know which **fun** method to call – Base::fun or Derived::fun?

Binding

- In general, the method invoked cannot be known at compile time.
- Thus, a compiler cannot figure out the type base pointer is pointing to.
- Therefore, we need to depend upon the run-time information.
- Compiler generates code to maintain a runtime table of pointer references, called virtual function table (*vtbl*).

```
Base *b;
if (input < 10)
  b = new Base();
else
  b = new Derived();

b->fun();
```

non-virtual functions → static binding
virtual functions → dynamic binding

Virtual Methods

- A virtual method declared in the base class makes the method virtual in base class, and in all the classes transitively derived from it.
- Constructors cannot be virtual.
- Destructors should be virtual, unless a class is not going to be used as a base class.
- Friends cannot be virtual functions.

Abstract Class

- A function can be pure virtual function.
  - virtual void fun() **= 0**;
- This makes the class abstract.
- Abstract class cannot be instantiated.
  - But its pointer / reference can be created.
- A derived class not implementing a pure virtual function is also abstract.
- A pure virtual function may have its definition in the abstract class.

Source: 6.cpp

Multiple Inheritance

- C++ allows deriving from multiple base classes.
  - Java doesn't.
- The derived class inherits properties of both the base classes.
- If there is ambiguity (same method in both bases), compiler issues an error.
- Multiple inheritance makes the type hierarchy a DAG.
  - In Java, it is a tree.

```
class Derived: public BaseOne,
                              public BaseTwo {

};
```

Exercises

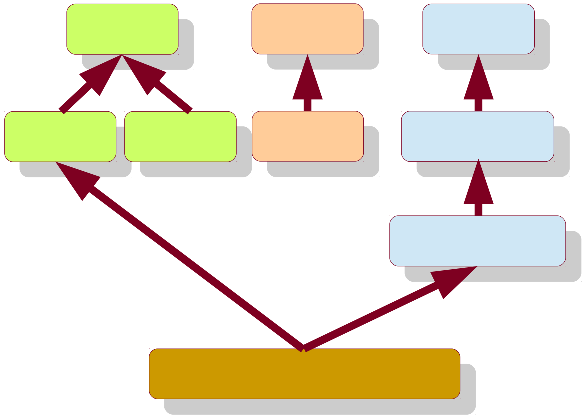
- **Quiz**: https://www.geeksforgeeks.org/c-plus-plus-gq/virtual-functions-gq/
- Create a hierarchy of Student, IT Student, Second Year IT Student, MBA Student, First Year Student. Identify one function and one field in each class which cannot be present in others.
- Create an abstract type Shape. Create classes Circle, Square, Rectangle, Triangle, Polygon. Maintain proper hierarchy. Now, enable the following functionality in main.

```
for (it = allShapes.begin(); it != allShapes.end(); ++it) {
      it->draw();
}
```

Summary

- ✓ Inheritance Basics
- ✓ Class Hierarchy
- ✓ Access Qualifiers
- ✓ Virtual Functions
- ✓ Pure Virtual Functions
- ✓ Multiple Inheritance

Inheritance and Virtual Functions



Rupesh Nasre.

IIT Madras

Raisoni

July 2020