CS-202

C++ Classes – Polymorphism (Pt.1)

C. Papachristos
Autonomous Robots Lab
University of Nevada, Reno

Course Week

Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	Friday
			Lab (4 Sections)	
	CLASS	RL – Session	CLASS	
PASS Session	PASS Session	**Project DEADLINE**	NEW Project	

Your 5th Project Deadline is this Wednesday 2/28.

- ➢ PASS Sessions held Monday-Tuesday
- ➢ RL Session held Wednesday

get all the help you may need!

- ➢ 24-hrs delay after Project Deadline incurs 20% grade penalty.
- ➢ Past that, NO Project accepted. Better send what you have in time!

Today's Topics

Polymorphism Concepts & Practice

- Abstraction via Polymorphism

Polymorphism in Inheritance

- Base Class Pointer(s)
- Arrays (or generally Data Structures) of Base Class Pointers

`virtual` Class Methods

Late Binding

- Static Binding *vs* Dynamic Binding

Polymorphism (prelude)

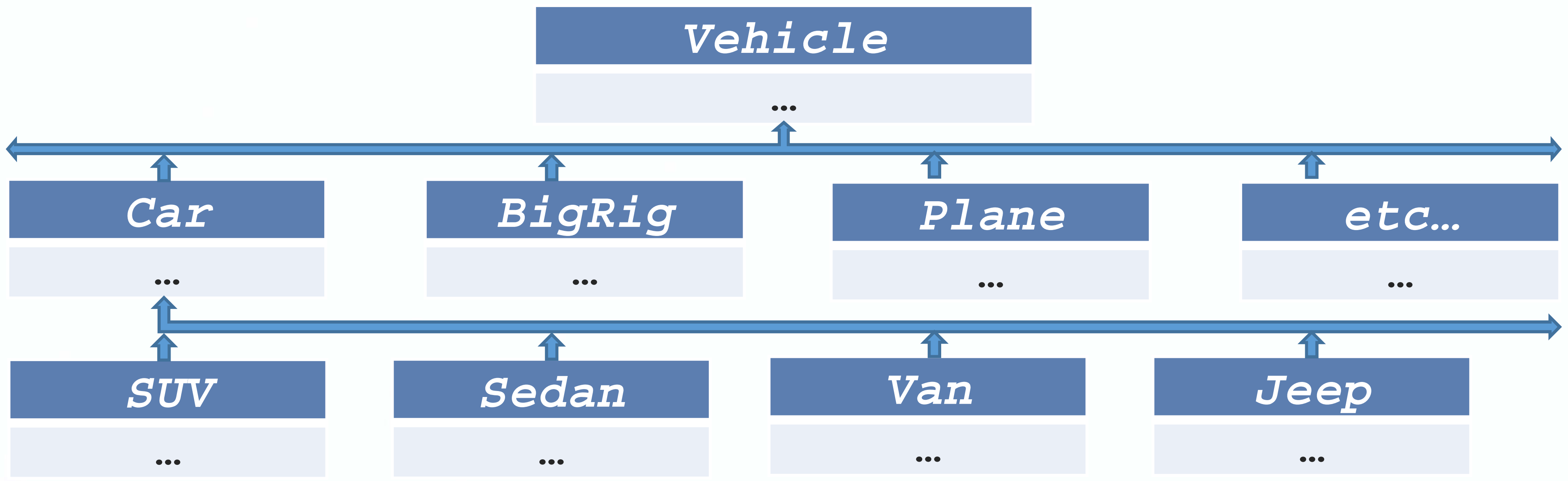
Polymorphism & Inheritance

Polymorphism means “the ability to take many forms”.

- Allowing a single (*Remember*: Overriding *vs* Overloading) behavior to take on many type-dependent forms.
- Hence, grants the ability to manipulate Objects in a type-independent way.

Pointers of Base Class-type:

- They are used to address “*Common Ancestor*”-type Objects from a Class Hierarchy.



Polymorphism & Inheritance

Base Class-type Pointers :

- A Pointer of a Base Class type can point to an Object of a Derived Class type.

```
SUV    suv1;
Sedan  sedan1, sedan2;
Jeep   jeep1;

Car *  suv1_Pt    = &suv1;
Car *  sedan1_Pt = &sedan1, * sedan2_Pt = &sedan2;
Car *  jeep1_Pt   = &jeep1;
```

This is valid: A Derived Class (`SUV`, `Sedan`, `Van`, `Jeep`) "is a type of" Base Class (`Car`).

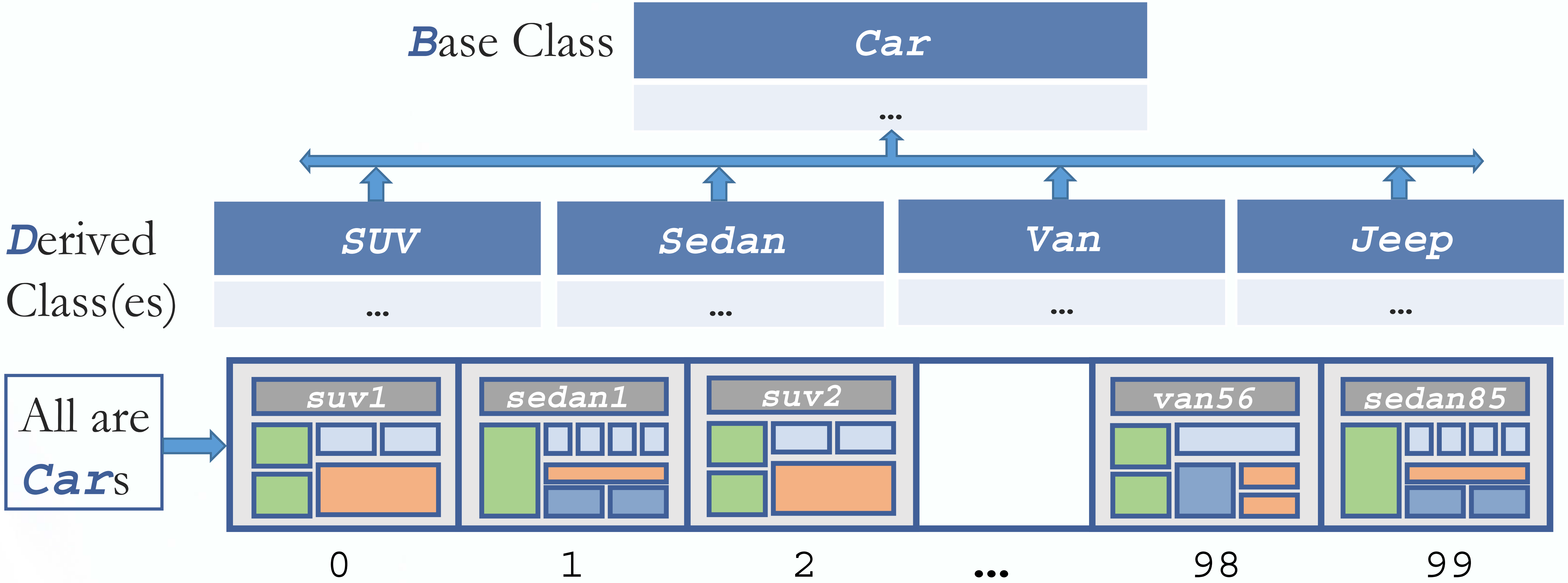
- *Note*: A 1-way relationship:

  A Derived Class Pointer cannot point to a Base Class Object.

Polymorphism & Inheritance

Supported through Pointers of Base Class-type:

➢ Problem: Implement a single catalog of different types of Cars available for rental.



Polymorphism & Inheritance

Supported through Pointers of Base Class-type:

➢ Problem: Implement a single catalog of different types of Cars available for rental.

? Multiple arrays, one for each Child Class type.

? Combine all Child Classes into one giant Class with redundant useless info for every kind of car.

Accomplished with a single array of Base Class Pointers !

➢ Polymorphism in the Data Structure.

➢ Inheritance enables this.

Note:

➢ Only *Array* of Base Class *Pointers* !

(not array of Objects because different Derived Classes require different allocation, and arrays don't do that).

➢ Dynamic Structures too (later on `std::vector` etc.)

Polymorphism & Inheritance

Base Class-type Pointers :

- A pointer of a Parent Class type can point to an Object of a Child Class type.

```
SUV   suv1;
Sedan sedan1, sedan2;
Jeep  jeep1;

Car * suv1_Pt    = &suv1;
Car * sedan1_Pt = &sedan1, * sedan2_Pt = &sedan2;
Car * jeep1_Pt  = &jeep1;
```

```
Car *cars_Pt_arr[4];
cars_Pt_arr[0] = suv1_Pt;
cars_Pt_arr[1] = sedan1_Pt;
cars_Pt_arr[2] = sedan2_Pt;
cars_Pt_arr[3] = jeep1_Pt;
```

Note:

- *Array* of Base Class *Pointers* (not Objects) !

Polymorphism & Methods

Refers to the ability to associate many meanings to one function name,

- by means of *Late Binding* !

Associating many meanings to one function:

- Fundamental principle of Object-Oriented Programming.
- `virtual` functions provide this capability.

Polymorphism

Virtual

Property that indicates something that is not concretely defined.

- It does exist, but in "essence".
- Not necessary to exist "in fact" at the very first point we refer to it …

`virtual` Function

- A function that "should be there".
- We can "use it / call it" – virtually , before it is even "fully defined" !

Virtual

What is the purpose?

- In effect, implements an incomplete behavior.
- If it indeed “used”, resulting code exhibits incomplete behavior.

Power of Programming Abstraction:

- Incomplete is not concrete.

but

- Incomplete is also modular !

Polymorphism

Virtual

By-Example:

Classes for several kinds of ***GeometricFigure***s:

- ***Rectangle***s, ***Circle***s, ***Oval***s, etc.

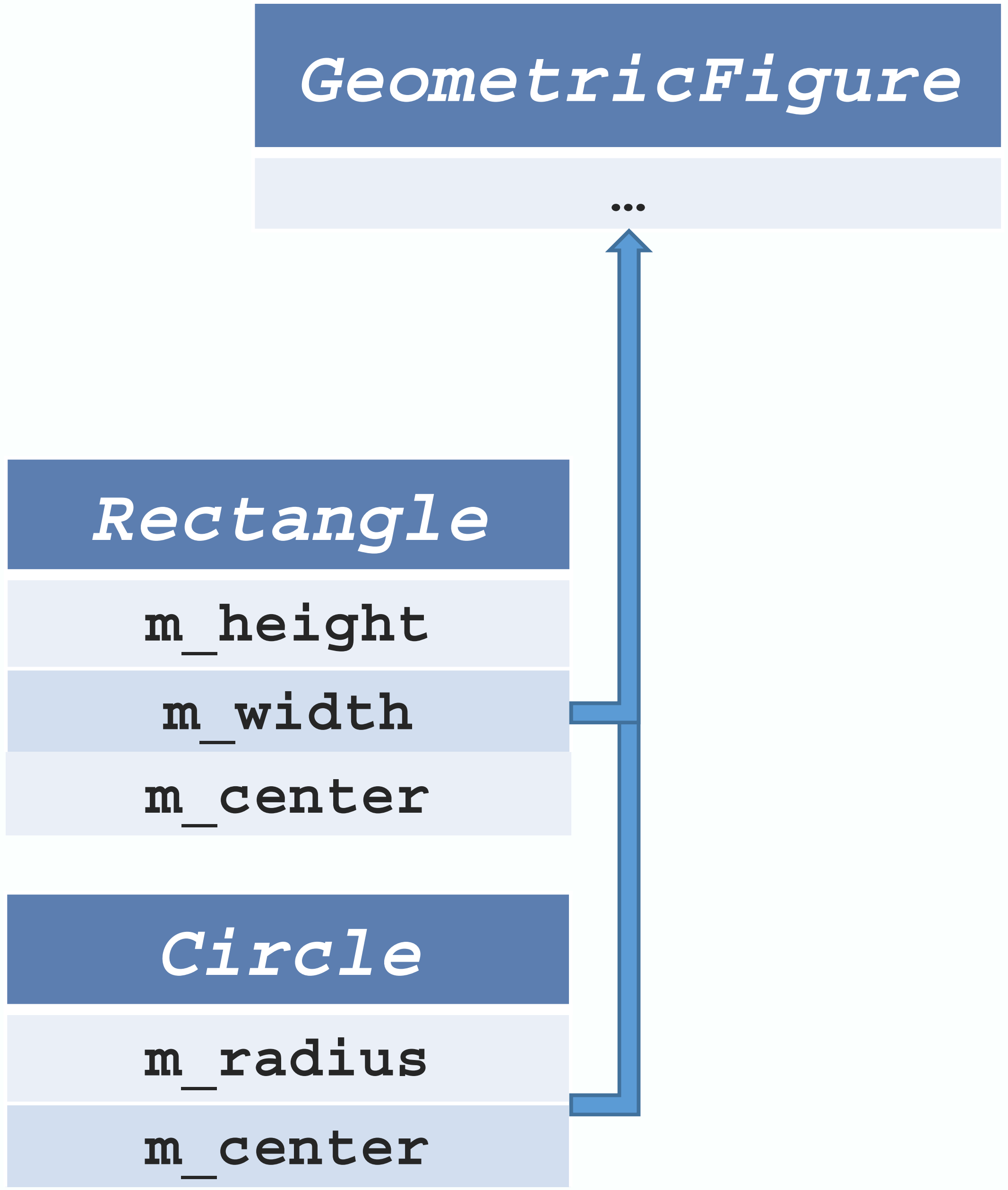
Each figure is an Object of different Class:

class ***Rectangle*** data: height, width, center point.

class ***Circle*** data: center point, radius.

All Derived from one Parent Class:

- ***GeometricFigure***



Polymorphism

Virtual

By-Example:

Classes for several kinds of ***GeometricFigure***s:

- ***Rectangle***s, ***Circle***s, ***Oval***s, etc.

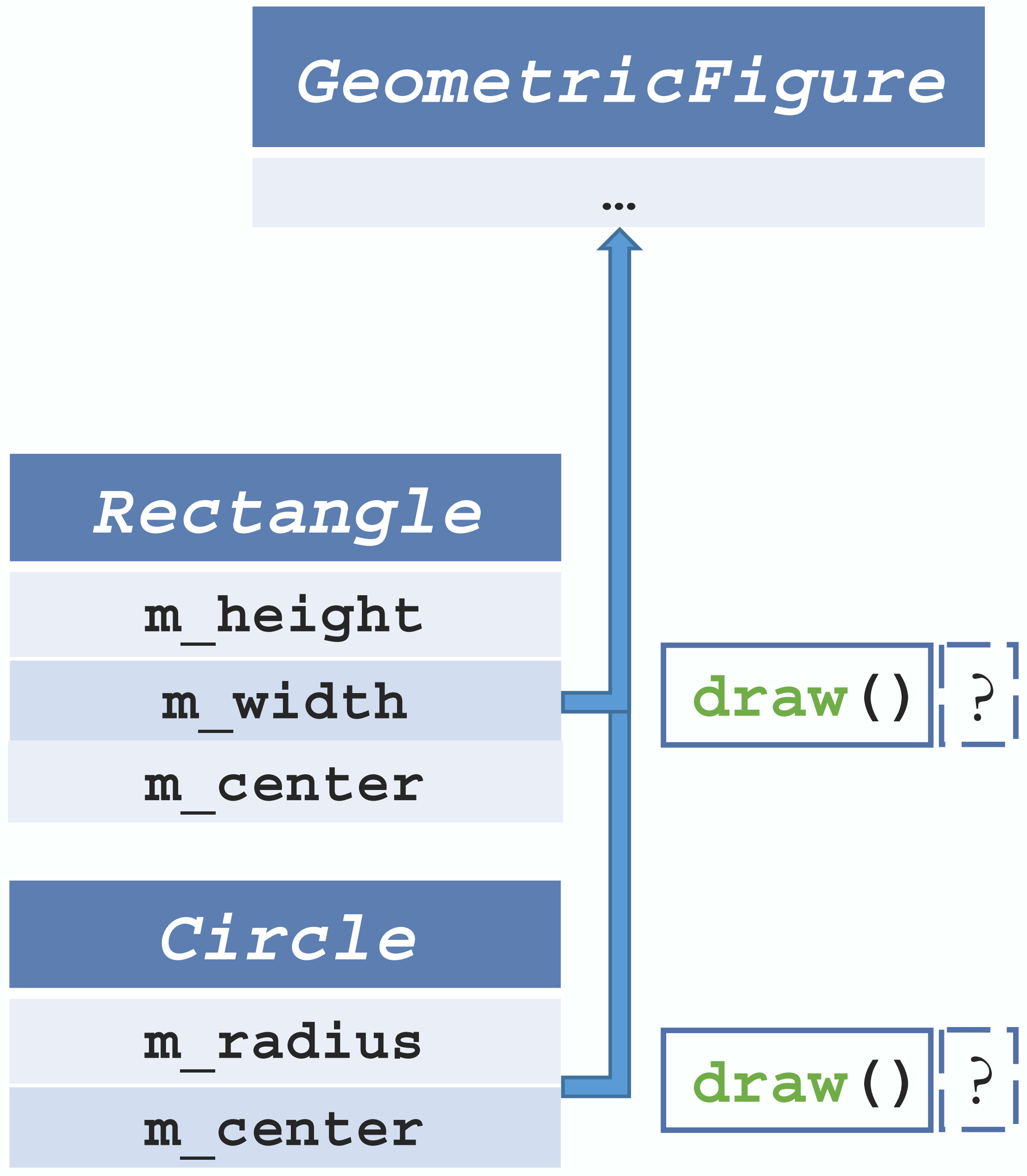
Each figure is an Object of different Class:

`class` ***Rectangle*** data: height, width, center point.

`class` ***Circle*** data: center point, radius.

All Require a function: `draw()`

- Different instructions for each figure type !



Polymorphism

Virtual

By-Example:

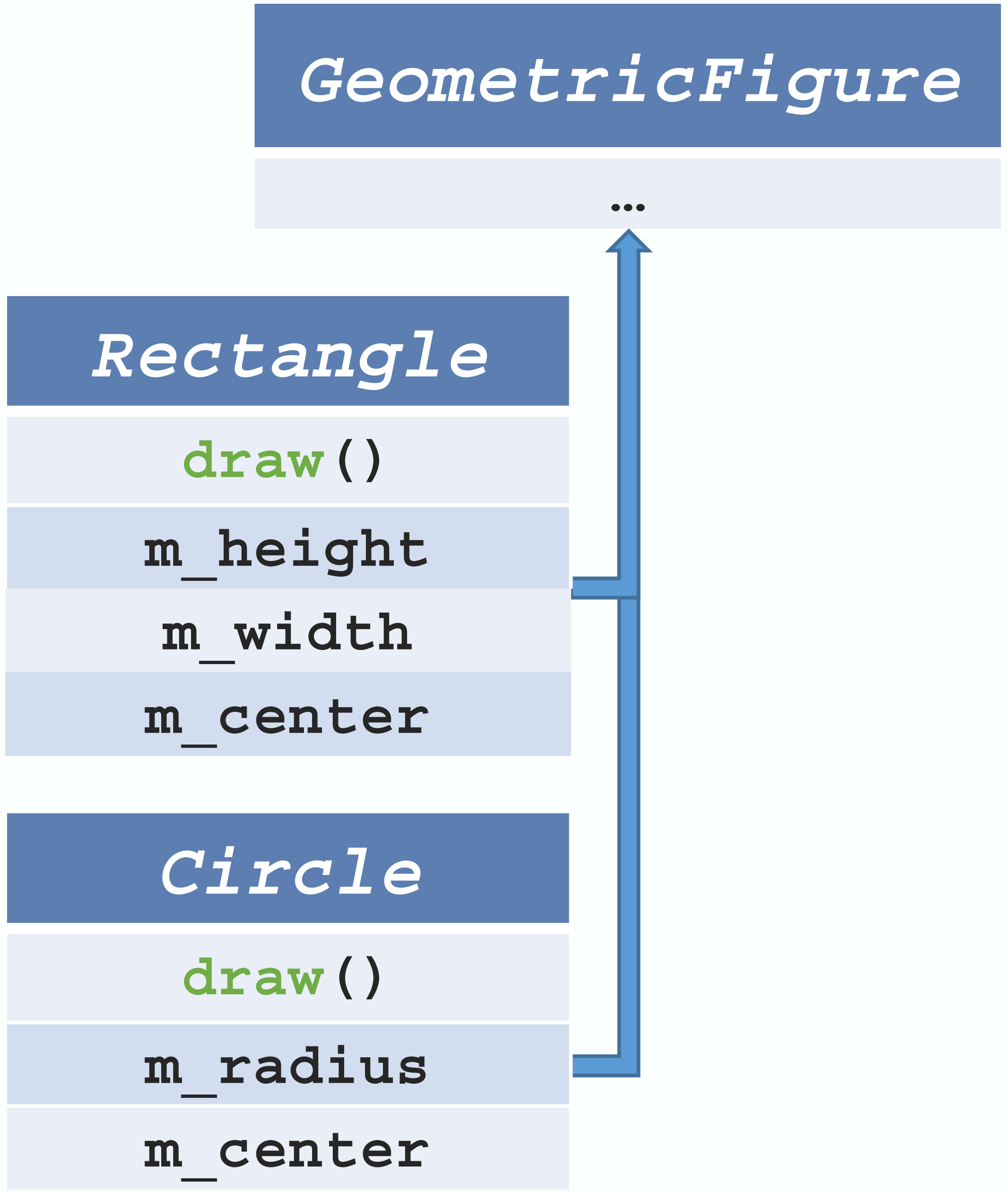
Each class needs different drawing function.

➢ Can be called `Draw()` in each different Class:

```
Rectangle r;
Circle c;

r.draw();  //Calls Rectangle class's draw()
c.draw();  //Calls Circle class's draw()
```

Nothing new here (yet).



Polymorphism

Virtual

By-Example:

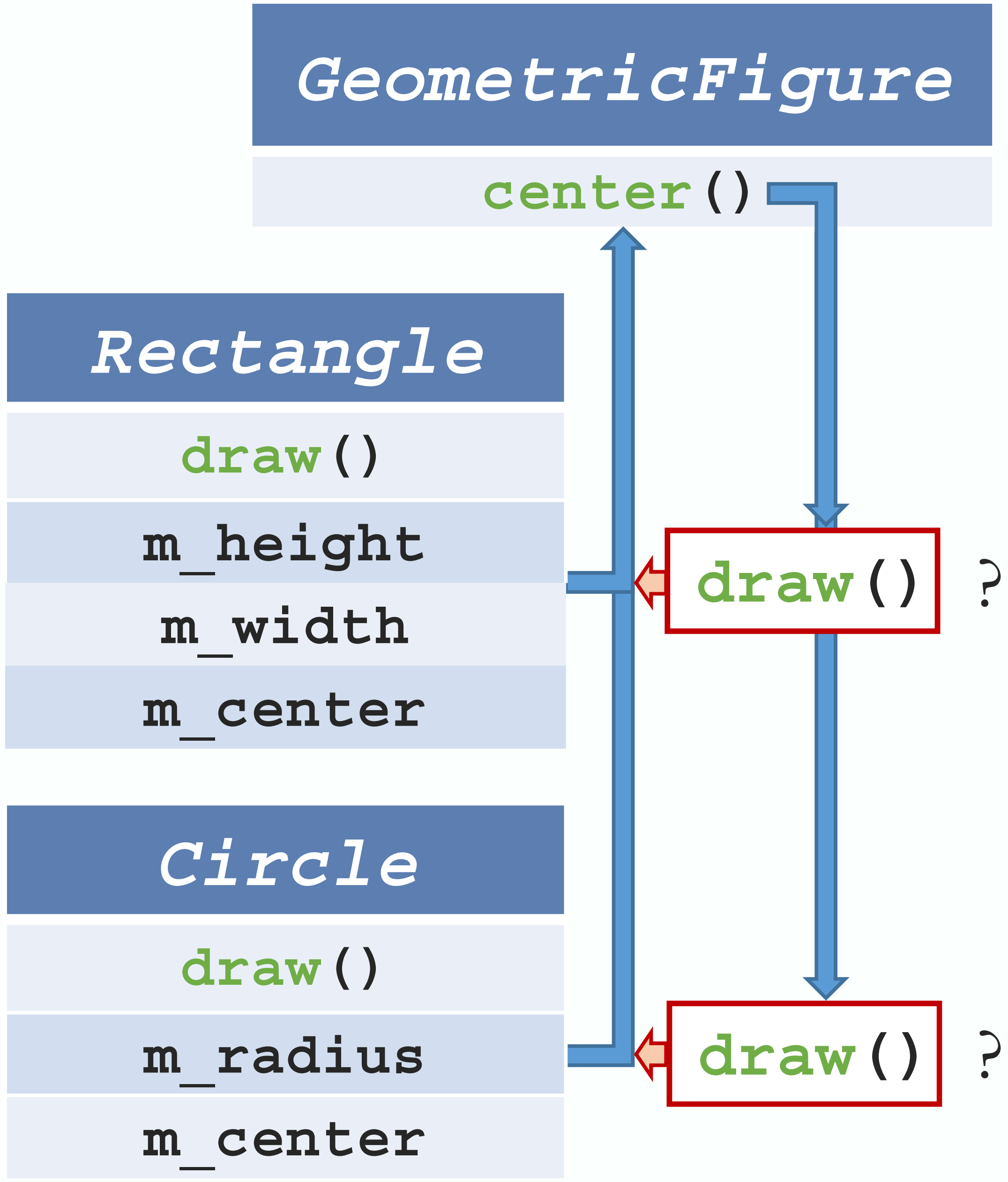
Parent Class `GeometricFigure` contains functions that apply to "all" figure types:

`center()`: moves a figure to center of screen.

- Erases 1st, then re-draws Object.

So `GeometricFigure::center()` would:

- *have to use* function `draw()` !



Complications:

- Which `draw()` function, from which Class, since we are implementing a Base Class behavior ?
- If `draw()` is defined in the Base Class it will use that one!

Polymorphism

Virtual

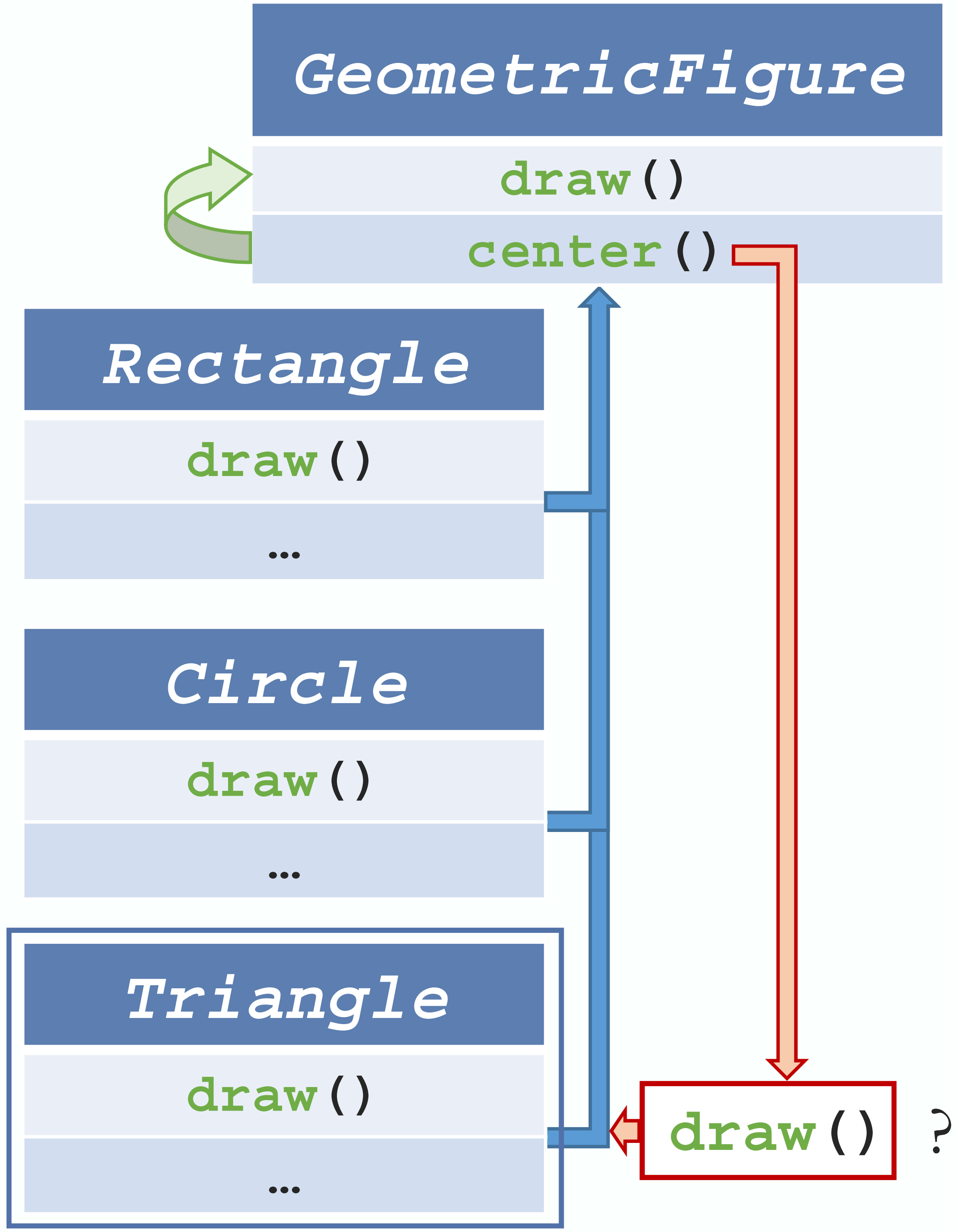
By-Example:

Consider a new kind of figure comes later into play:
`class` ***Triangle***: Derived from ***GeometricFigure***.

Function `center()` is Inherited.

➢ Will it work for ***Triangle***s?

It uses `draw()`, which is different for each figure!



Polymorphism

Virtual

By-Example:

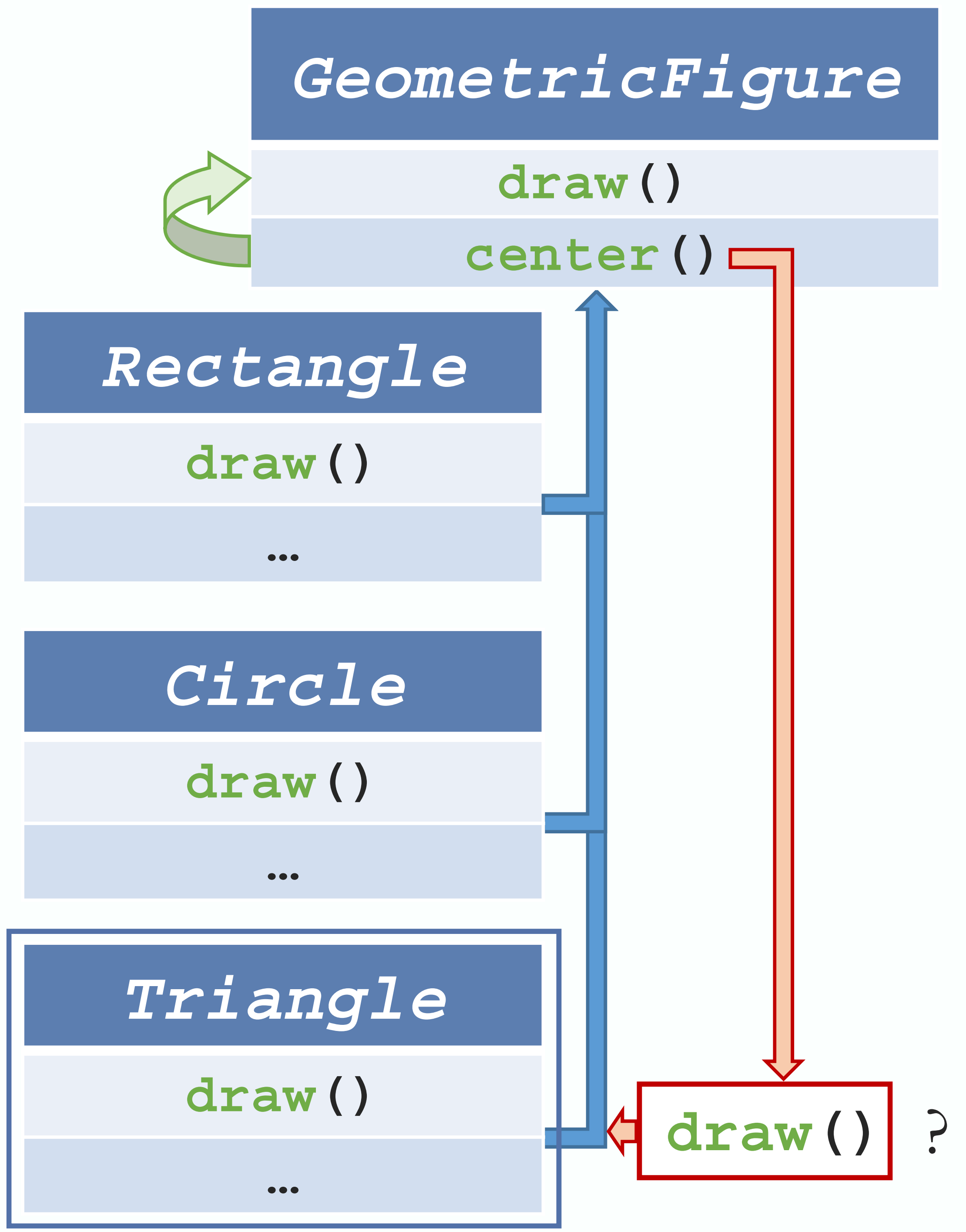
Consider a new kind of figure comes *later* into play:
`class Triangle`: Derived from `GeometricFigure`.

It will use `GeometricFigure::draw()`:

- ➢ won't work for `Triangles`.

Want Inherited function `center()` to use function `Triangle::draw()`.

But Class `Triangle` wasn't even written when `GeometricFigure::center()` was.



Virtual

How? `virtual` functions are the answer.

Tells the C++ compiler:

- "Don't know" how function is implemented (but not "don't really care at the moment"!)
- Wait until it is used in the program (at *runtime*)
- Then get the specific implementation from the specific Object instance that called it!

Polymorphism

Virtual

Virtual functions are the answer.

Tells the C++ compiler to:

- Get the implementation from the specific Object instance at *runtime*.

Called *Late Binding* or *Dynamic Binding*.

Virtual functions implement *Late Binding*.

Virtual

By-Example (a larger application problem):

Record-keeping program module for automotive parts store:
- Sales tracking.

Issue:
- Don’t know all types of sales yet.
  Initially just working with regular retail sales,
  but later on, discount sales, mail-order sales, etc. might come along.

These might additionally depend on other factors besides just price, tax.

Virtual

By-Example (a larger application problem):

Program must:
- Compute daily gross sales.
- Calculate largest/smallest sales of day.
- Average daily sales.

All will come from individual bills.
- But many functions for computing bills will be added "later" !

Function for computing a bill will be `virtual` !

Polymorphism

Keyword `virtual`

By-Example (a larger application problem):

```
class Sale {
 public:
   Sale();
   Sale(double price);
   double getPrice() const;
   virtual double bill() const;
   double savings(const Sale&
                   other) const;
 private:
   double m_price;
};
```

A general Super-class:

- Represents sales of a single item with no added discounts or charges.

A `virtual` Member Function:

- Impact: *Later*, Derived classes of `Sale` can define their own versions of Function `Bill()`.
- Other Member Functions of `Sale` will use version based on Object of that Derived class !
- They won't automatically use `Sale`'s version!

Polymorphism

Keyword `virtual`

By-Example (a larger application problem):

A Member Function of Super-Class *Sale*.

```
double Sale::savings(const Sale& other) const
{
    return (bill() - other.bill());
}
```

A non-Member Function – Operator (<).

```
bool operator<(const Sale& first,
               const Sale& second)
{
    return (first.bill() < second.bill());
}
```

Both use the `virtual` Member Function `bill()`.

Keyword `virtual`

By-Example (a larger application problem):

```
class DiscountSale : public Sale {
 public:
   DiscountSale();
   DiscountSale(double price,
                double discount);
   double getDiscount() const;
   void setDiscount(double newDisc);
   virtual double bill() const;
 private:
   double m_discount;
};
```

The Derived Sub-class:

- Represents more specialized sales type.

The Base class' `virtual` Function:

- Automatically `virtual` in Derived Class.
- Not required to have `virtual` keyword but typically included for readability.
- The Derived class will implement its own version of the `virtual` Function.

Keyword `virtual`

By-Example (a larger application problem):

Derived Class' more particular implementation of the `virtual` Member Function.

```
double DiscountSale::bill() const
{
   double fraction = m_discount/100;
   return (1 - fraction)*getPrice();
}
```

`virtual` qualifier does not appear in implementation (only declaration).

`DiscountSale`'s Member Function `bill()` implemented differently than *`Sale`*'s:

- Member Function `savings()` and non-Member `operator<(…)` will use this definition of `bill()` for all Objects of *`DiscountSale`* Class (instead of defaulting to version defined in *`Sales`*).

Virtual

Remember:

- Base Class `Sale` written before Derived Class `DiscountSale`.
- Member Function `savings()` and non-Member `operator<(…)` compiled before even having the concept of a `DiscountSale` Class.

Yet consider the call:

```
DiscountSale d1, d2;
d1.savings(d2);
```

- Powerful !

  The call inside `savings()` to function `bill()` has no problem to work with the definition of `bill()` from a `DiscountSale` Class.

Virtual

Remember:

- Base Class `Sale` written before Derived Class `DiscountSale`.
- Member Function `savings()` and non-Member `operator<(…)` compiled before even having the concept of a `DiscountSale` Class.

- Powerful !
  Even non-Member Functions can be Polymorphic.

Can pass a Pointer or Reference to a Base Class Object.

- Subsequent Method calls are *Dynamically Bound.*
- Old code is dynamically calling new code when new Derived Classes are defined!

Static *vs* Dynamic Binding

What is the difference with Method Overriding as known to us by now?

- Determining which Method in Hierarchy to call.

Static Binding (overriding as of yet)
Compiler –at "*compile-time*"– determines binding.

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

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)

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)

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

Static Binding

```
class Animal {
 public:
  void eat(){
   cout<<"Food"<<endl;
  }
};
class Lion : public Animal{
 public:
  void eat(){
   cout<<"Meat"<<endl;
  }
};
```

Objects

Object Pointers

```
int main(){
  Animal animal;
  Lion lion;

  animal.eat();                          // Food
  lion.eat();                            // Meat

  Animal * animal_Pt = &animal;
  animal_Pt->eat();                      //Food

  Animal * animalLion_Pt = &lion;
  animalLion_Pt->eat();                  //Food

  return 0;
}
```

Static Binding

Remember: Static *vs* Dynamic Binding

Dynamic Binding

```
class Animal {
 public:
  virtual void eat();
};
void Animal::eat(){
 cout<<"Food"<<endl;
}
class Lion : public Animal{
 public:
  virtual void eat();
};
void Lion::eat(){
 cout<<"Meat"<<endl;
}
```

Objects

Object Pointers

```
int main(){
  Animal animal;
  Lion lion;

  animal.eat();                                  // Food
  lion.eat();                                    // Meat

  Animal * animal_Pt = &animal;
  animal_Pt->eat();                              //Food

  Animal * animalLion_Pt = &lion;
  animalLion_Pt->eat();                          //Meat

  return 0;
}
```

Dynamic Binding

Virtual

Consider:
In C++ programs, nothing happens by "magic".
- Explanation involves *Late Binding*.

Tell C++ compiler to wait until function is used in program, and decide which definition to use based on the current calling Object.

Disadvantages:
- Overhead – Late binding is "on the fly" at runtime.
- Use of more storage – "Hidden elements" in how it is implemented.

If `virtual` functions are not necessary, they should be avoided.

CS-202

Time for Questions !