

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 (8:00-12:00)	
	CLASS		CLASS	
PASS Session	PASS Session	Project DEADLINE	NEW Project	

Your 5th Project Deadline is this Wednesday 10/11.

- PASS Sessions held Monday-Tuesday, 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)

virtual Class Methods

Late Binding

- Static Binding *vs* Dynamic Binding

Polymorphism

Polymorphism & Inheritance

Means “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.

Only supported through Pointers of Base Class-type:

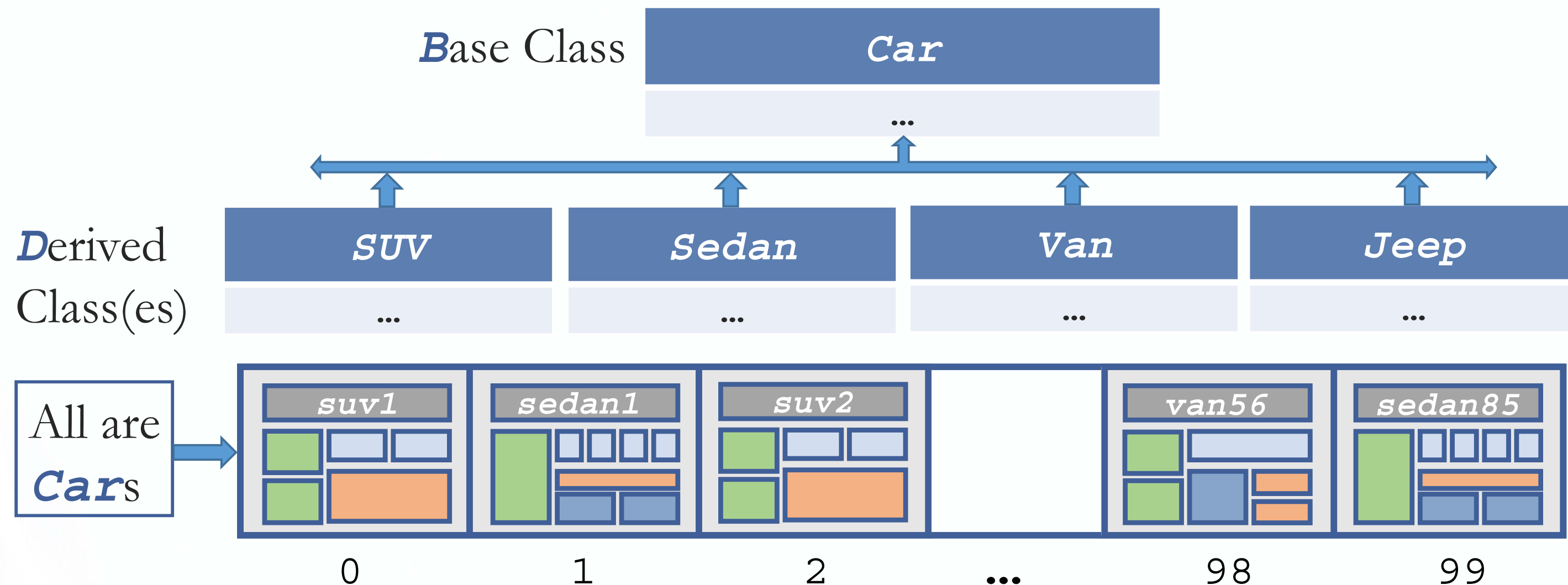
- The “common ancestor” to refer to in order to group behaviors.

Polymorphism

Polymorphism & Inheritance

Supported through Pointers of Base Class-type:

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



Polymorphism

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

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*;

This is valid: A Derived Class (*SUV*, *Sedan*, *Van*, *Jeep*) “is a” Base Class (*Car*).

- Remember: A 1-way relationship:


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

Polymorphism

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

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 initially necessary to exist “in fact”.

virtual Function

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

Polymorphism

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 *GeometricFigures*:

➤ *Rectangles*, *Circles*, *Ovals*, etc.

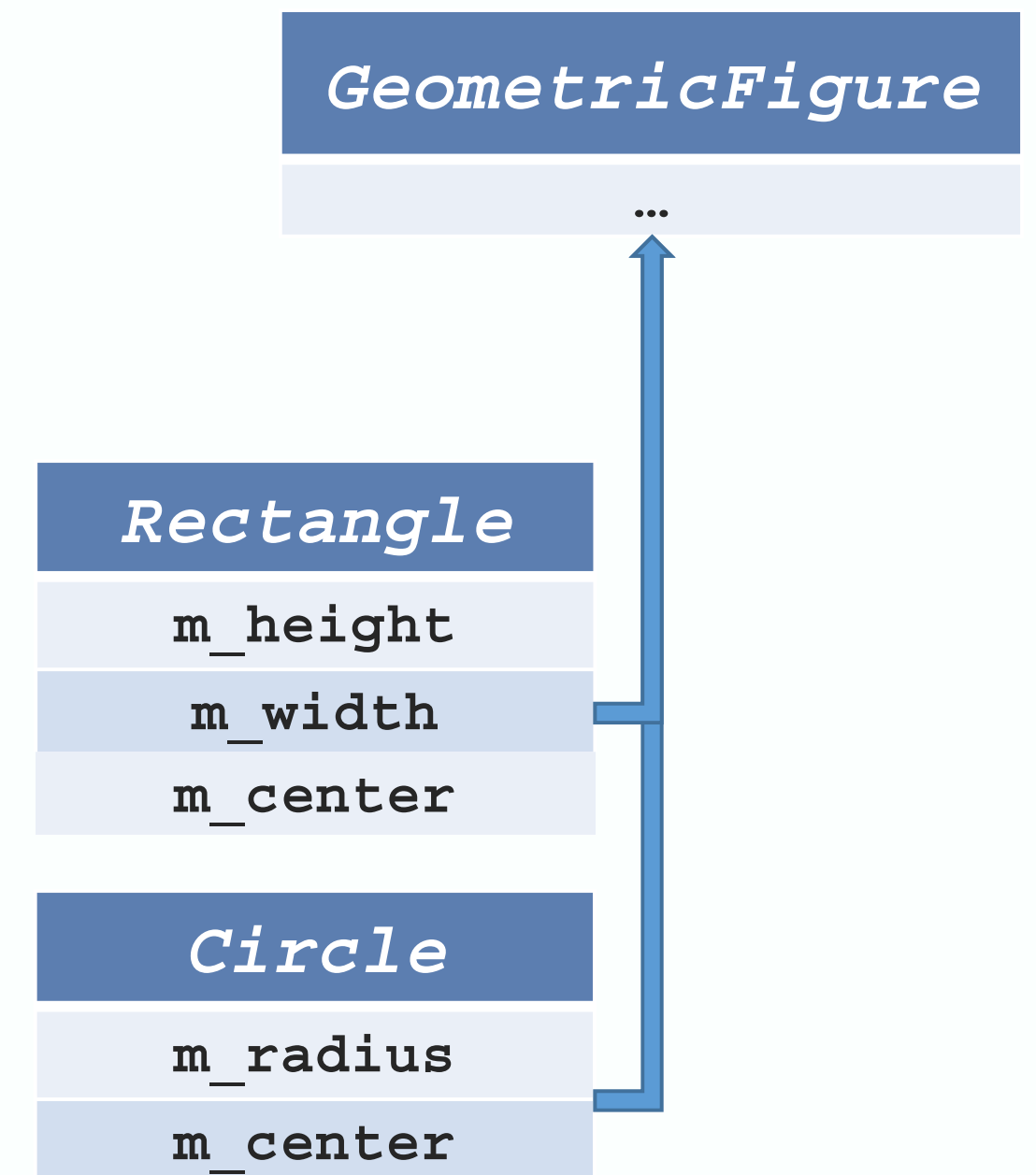
Each figure is an Object of different Class:

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 *GeometricFigures*:

➤ *Rectangles*, *Circles*, *Ovals*, etc.

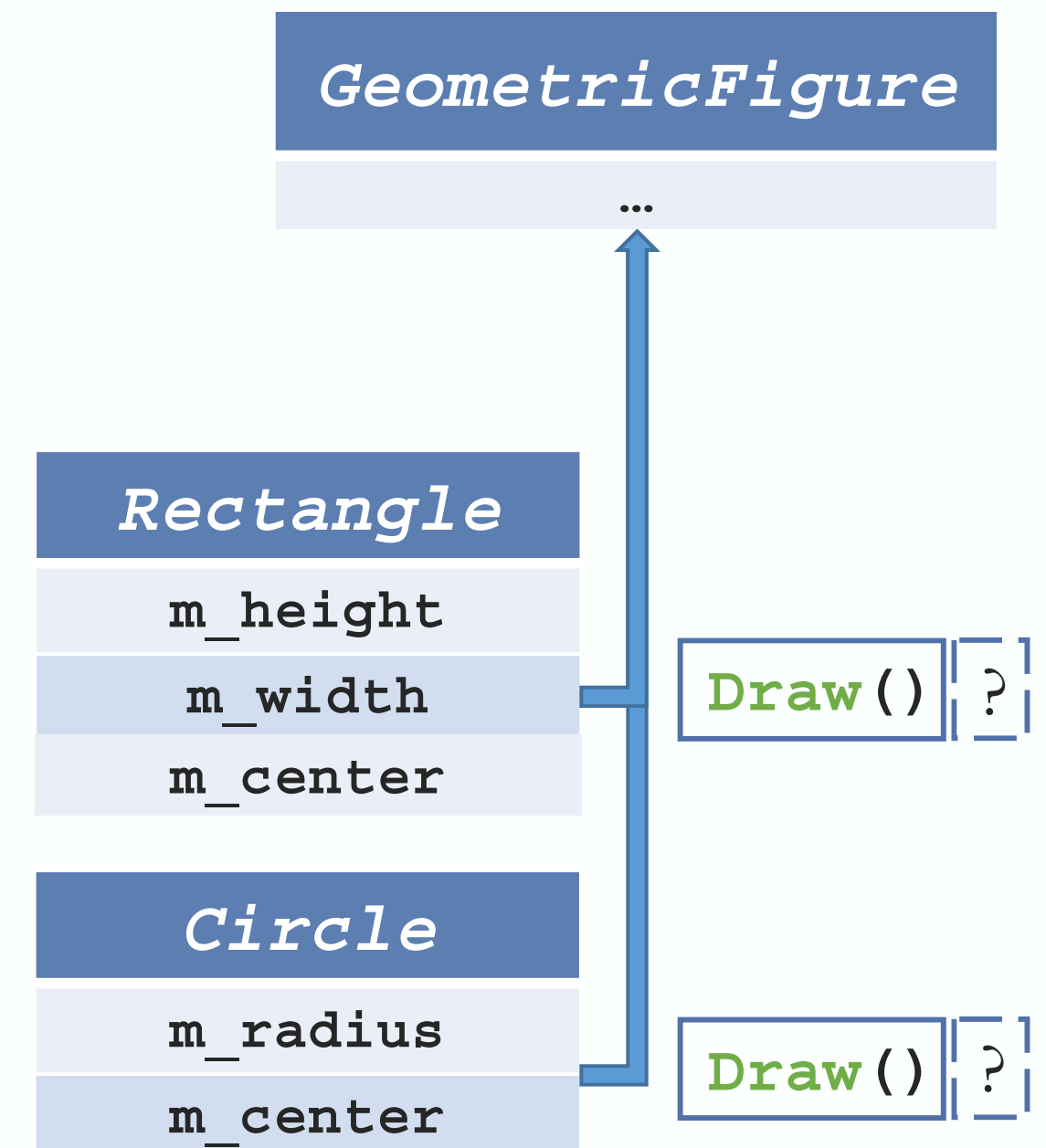
Each figure is an Object of different Class:

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:

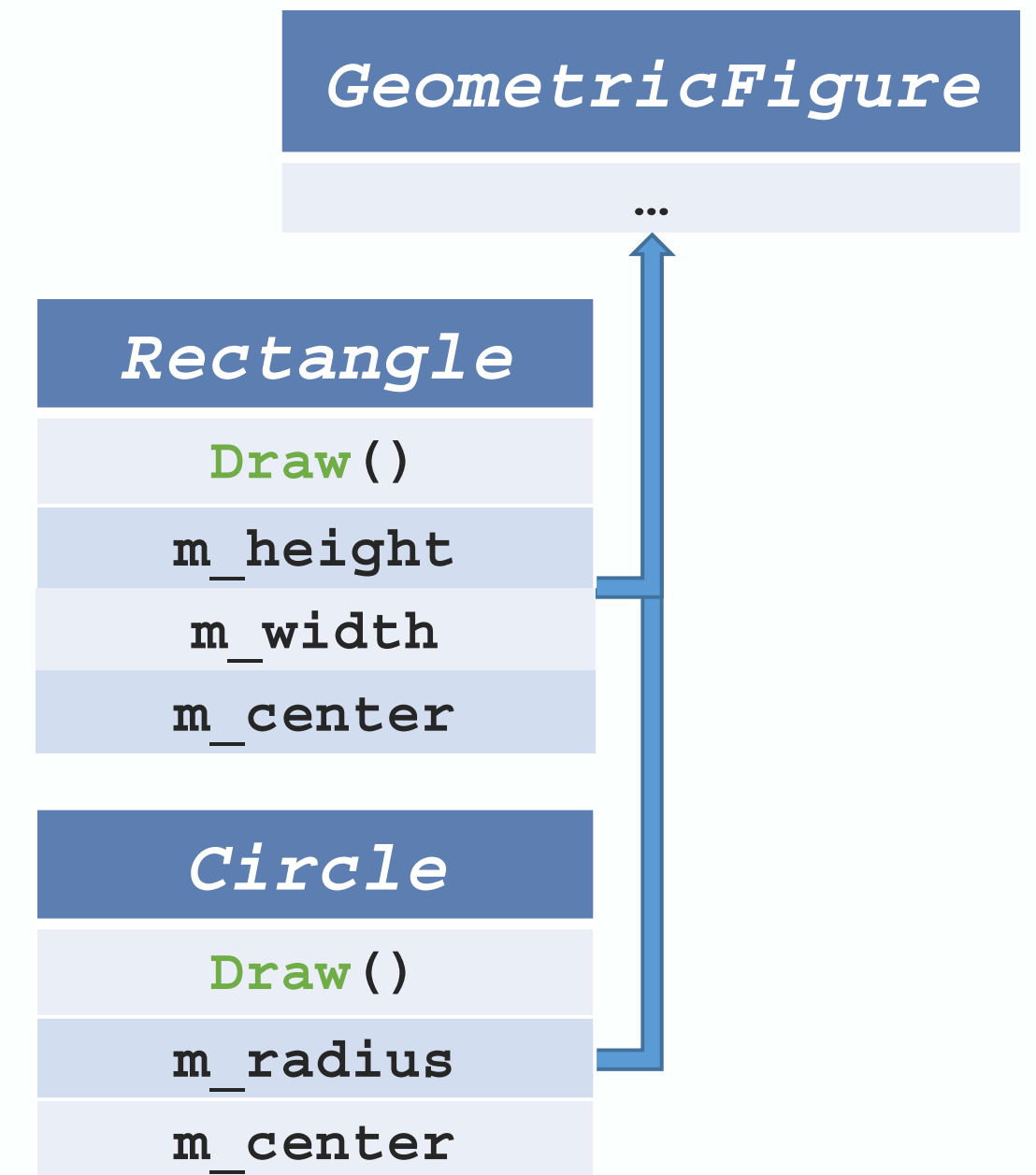
```
Rectangle r;
```

```
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.

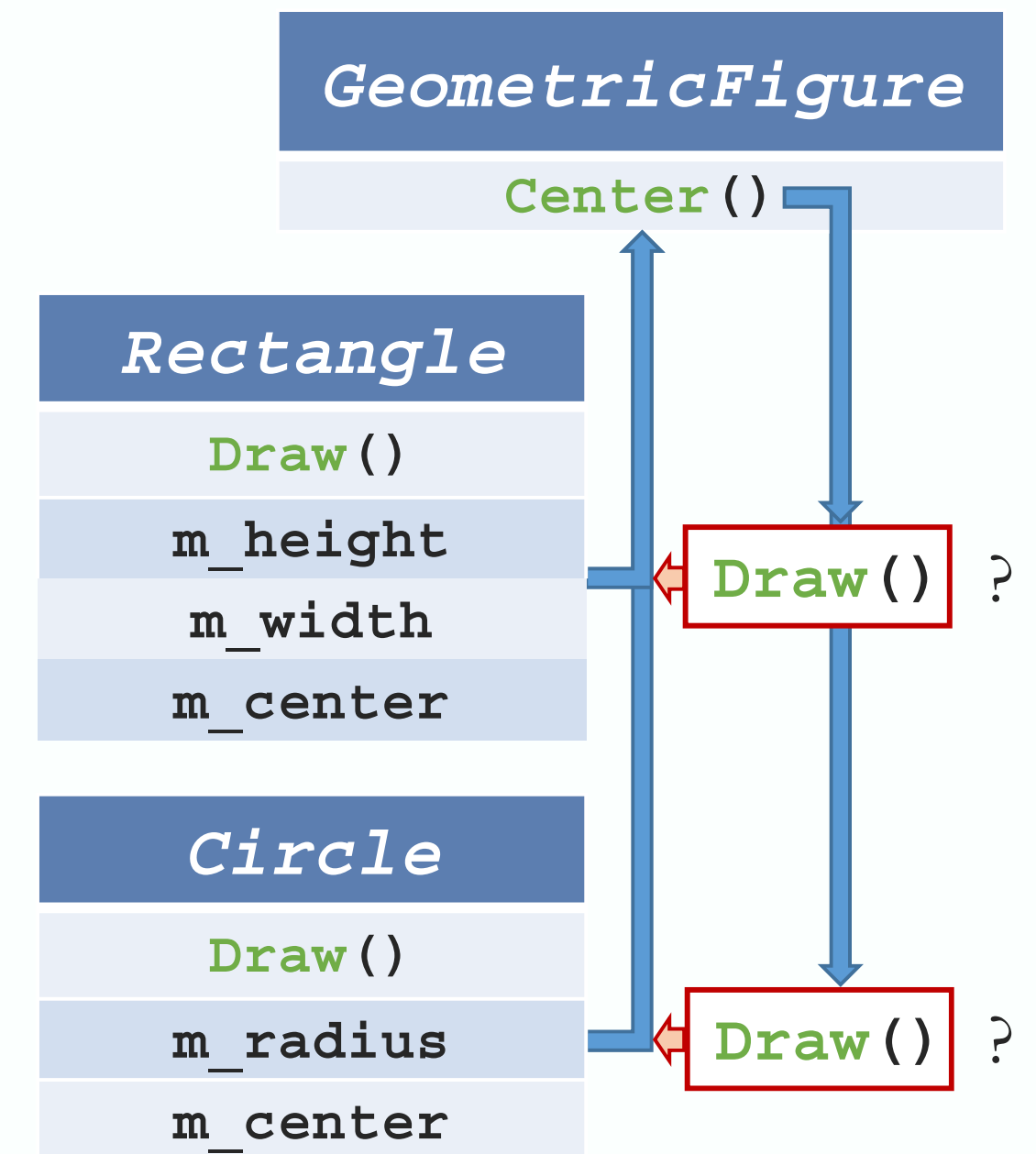
- Erases 1st, then re-draws Object.

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

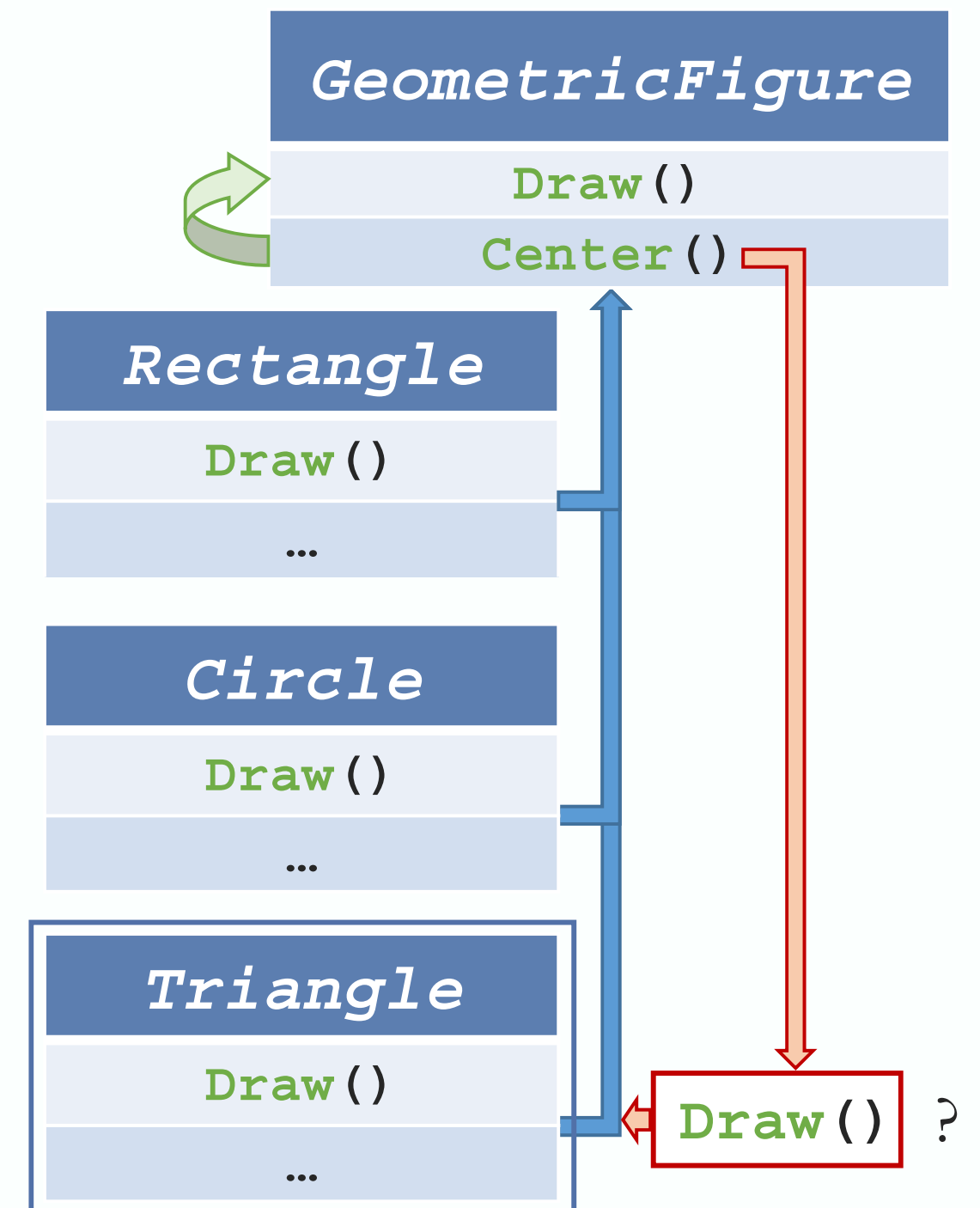
By-Example:

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

Function **Center()** is Inherited.

➤ Will it work for *Triangles*?

It uses **Draw()**, which is different for each figure!



Polymorphism

Virtual

By-Example:

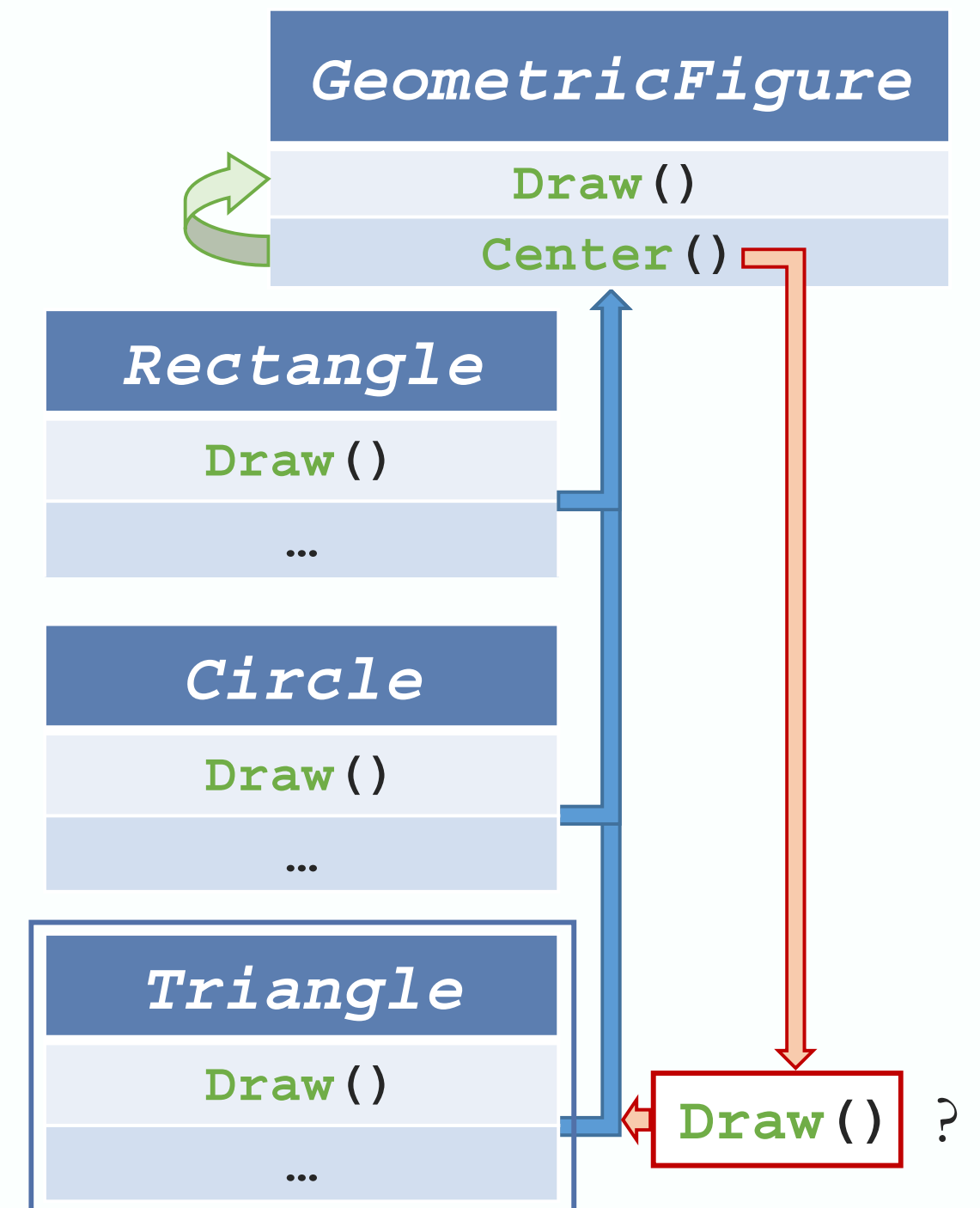
Consider a new kind of figure comes *later* into play:
class *Triangle*: Derived from *GeometricFigure*.

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.



Polymorphism

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*.

Polymorphism

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.

Polymorphism

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()**.

Polymorphism

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.

Polymorphism

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*).

Polymorphism

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.

Polymorphism

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!

Polymorphism

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.

Polymorphism

Keyword **virtual**

COVARIANT TYPES !

Polymorphism

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;  
    }  
};
```

	<pre>int main() { Animal animal; Lion lion; animal.Eat(); lion.Eat(); Animal *animal_Pt = &animal; animal_Pt->Eat(); Animal *animalLion_Pt = &lion; animalLion_Pt->Eat(); return 0; }</pre>	<pre>// Food // Meat // Food // Food</pre>
Objects		
Object Pointers		

Static Binding

Polymorphism

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;  
}
```

	<pre>int main() { Animal animal; Lion lion; animal.Eat(); lion.Eat(); Animal *animal_Pt = &animal; animal_Pt->Eat(); Animal *animalLion_Pt = &lion; animalLion_Pt->Eat(); return 0; }</pre>	<pre>// Food // Meat</pre>
Objects		
Object Pointers		<pre>//Food //Meat</pre>

Dynamic Binding

Polymorphism

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 !