Object-Oriented Programming Basics

Programming Paradigms

Imperative → **Structured** → **Procedural**

Imperative

Computation seen as a sequence of statements that directly change a program state

Global variables
Direct assignments
No modularity

Structured

More structure is added to imperative programming

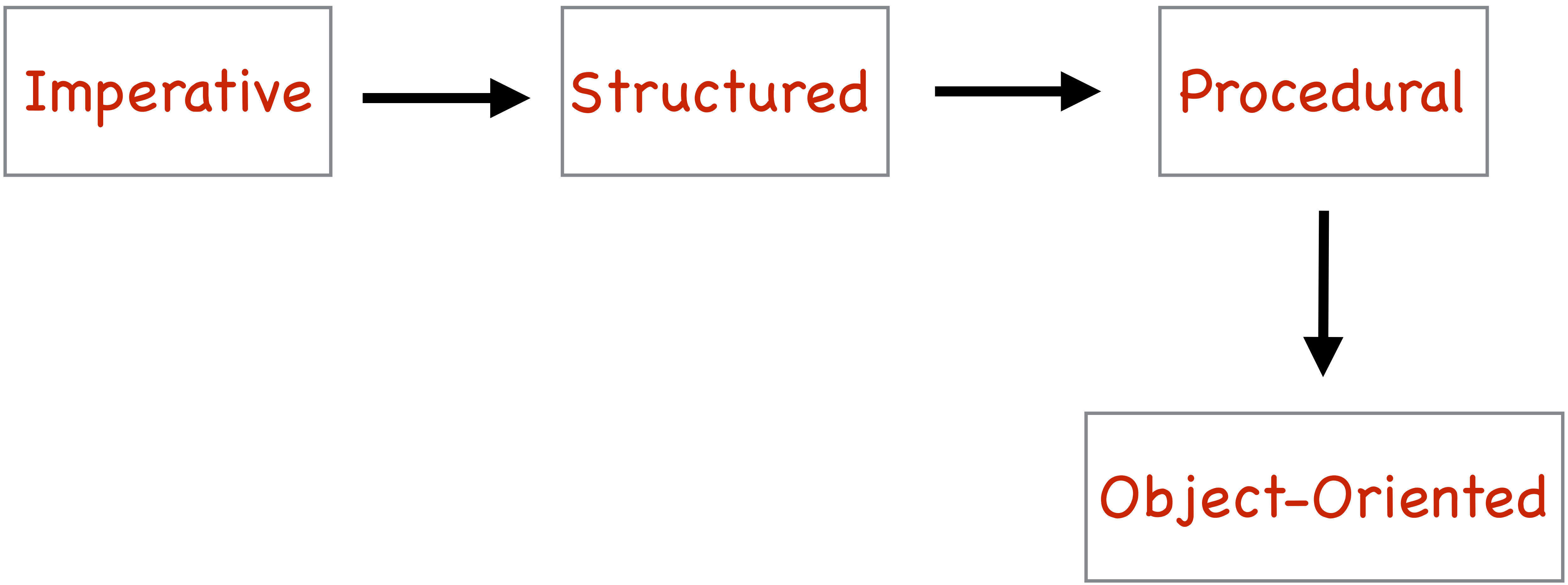
Almost no GOTO allowed: structure forced.
Introduction of indentation

Procedural

High Modularity

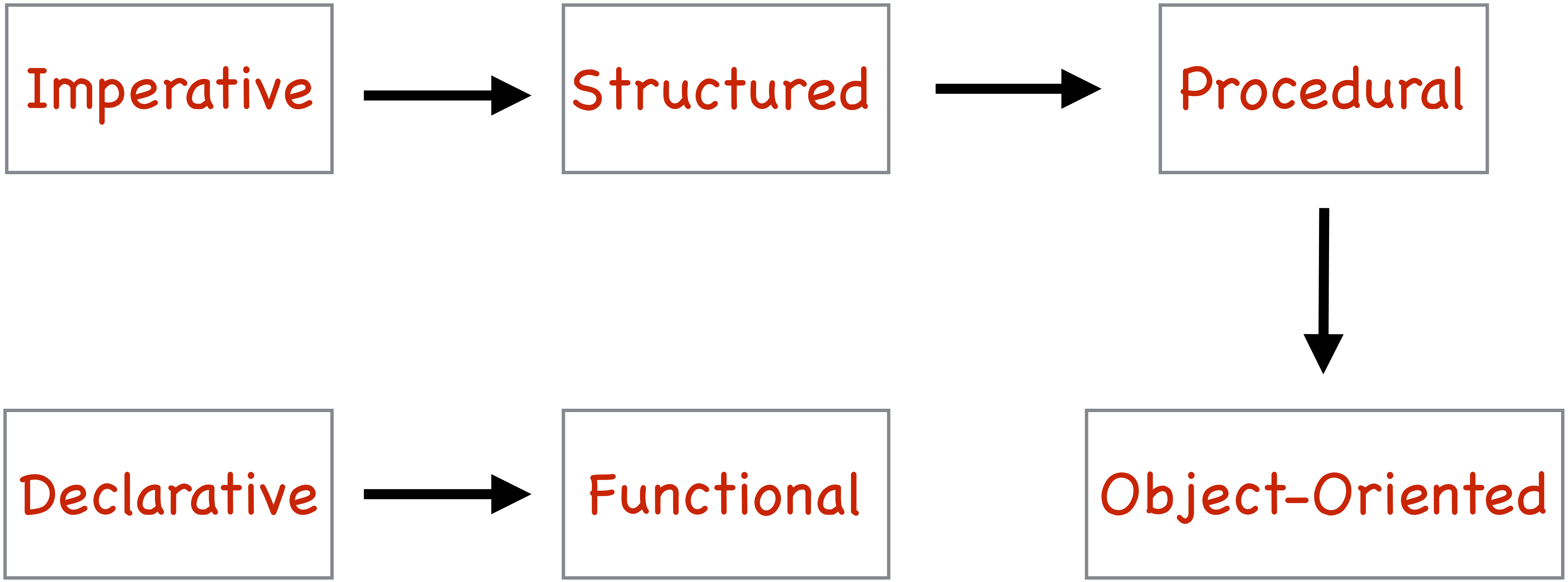
Local variables.
Globals avoided.
No GOTO: iteration statements.

Programming Paradigms



Objects and Methods.
Encapsulation.
Inheritance.
Polymorphism.

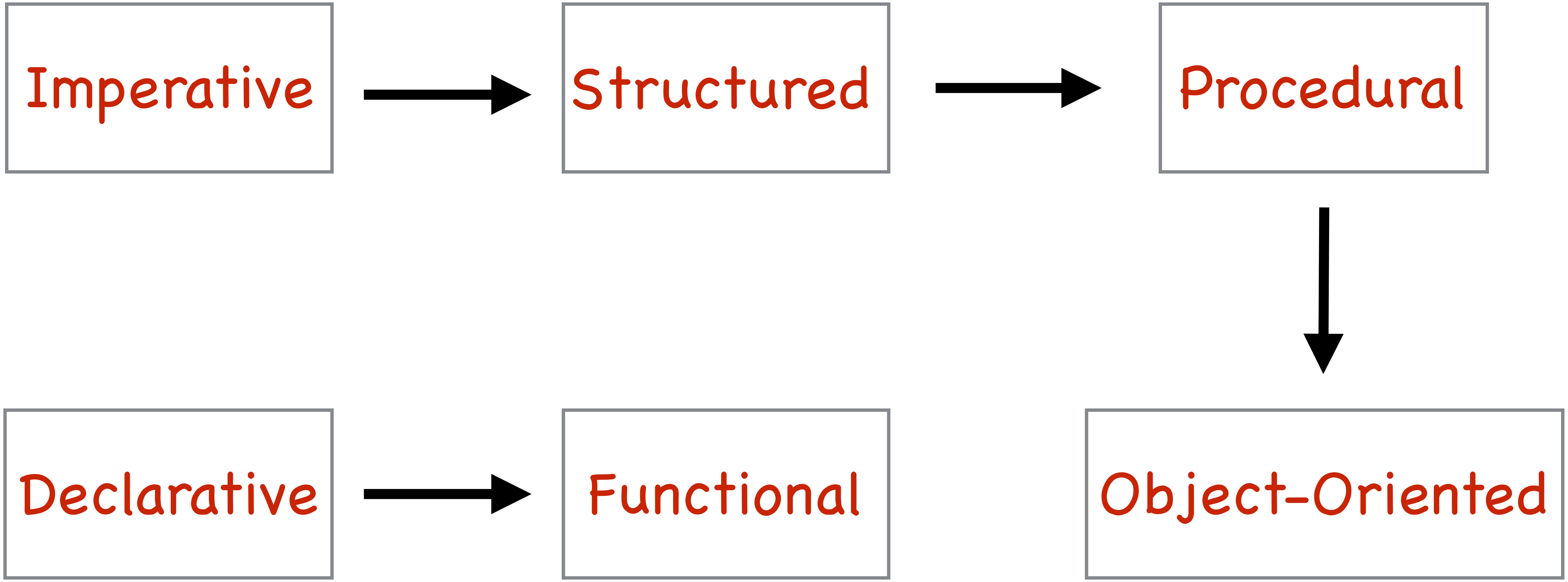
Programming Paradigms



Defines the logic instead of the flow.

Sees computation as the evaluation of mathematics-like functions.
Coding by "expressions"

Programming Paradigms



- (Partial) Overlaps of the paradigms are common in programming languages.
- **C++, C#, Java** are Imperative, Structured, Procedural and OO languages.
- **SQL** is an example of declarative language.
- **Python** is closer to the functional approach.
- Some modern languages are “multiparadigm”: e.g. C# with some extensions.

Object-Oriented Programming

1) Objects and Methods

2) Encapsulation

3) Inheritance

4) Polymorphism

Objects and Methods

An **Object** is a **data structure** containing data and functions.

The data is usually called **fields** or **attributes**.
The functions are called **methods**.

In the following, we concentrate on OO languages like Java, C++, C#.
In these languages, an object is an **instance** of a **class**.

The Class

Example:

Let's define a **class** (at this point we disregard the specific language and some details we will discuss later):

```
class Car {
    int NCylinders;
    int HP;
    int Npassengers;
    float max_speed;
    float price;
    float weight;
    float Specific_Power();
};
```

The Class

Example:

Let's define a **class** (at this point we disregard the specific language and some details we will discuss later):

```
class Car {
    int NCylinders;
    int HP;
    int Npassengers;
    float max_speed;
    float price;
    float weight;
    float Specific_Power();
};
```

← Data (attributes)

The Class

Example:

Let's define a **class** (at this point we disregard the specific language and some details we will discuss later):

```
class Car {
   int NCylinders;
   int HP;
   int Npassengers;
   float max_speed;
   float price;
   float weight;
   float Specific_Power();     ← Method
};

float Specific_Power(){
   return this.HP/this.weight;
}
```

The Object

The object is an instance of a class.
The class is similar to a new TYPE (like int, float, double, ...): as you declare a variable as e.g. `int a;` , an object is instantiated as:

```
Car Toyota;
```

Now we have an object called "Toyota" which is an instance of the class "Car". We can access (with restrictions we will see later) the data and methods as:

```
Toyota.HP = 100;

float specificHP;
specificHP = Toyota.Specific_Power();
```

C++ as Object-Oriented Language

Let's now turn to a specific OO language: C++.
Its syntax is quite close to Java or C#.
Rewriting the previous class:

```
class Car {
   public:
     int NCylinders;
     int HP;
     int Npassengers;
     float max_speed;
     float price;
     float weight;
     float Specific_Power();
};

float Specific_Power(){
   return this.HP/this.weight;
}
```

C++ as Object-Oriented Language

```
class Car {
  public:
    int NCylinders;
    int HP;
    int Npassengers;
    float max_speed;
    float price;
    float weight;
    float Specific_Power();
};

float Specific_Power(){
  return this.HP/this.weight;
}
```

C++ as Object-Oriented Language

```
class Car {
  private:
      int NCylinders;
      int HP;
      int Npassengers;
      float max_speed;
      float price;
      float weight;
  public:
      float Specific_Power();
};

float Specific_Power(){
   return HP/weight;
}
```

Full Example

```
#include <iostream>

using namespace std;

class Car {
private:
  int NCylinders;
  int HP;
  int Npassengers;
  float max_speed;
  float price;
  float weight;
public:
  Car();
  float Specific_Power();
  void SetCylinders(int c);
  void SetWeight(float w);
  void SetHP(int hp);
};

Car::Car(){}

void Car::SetCylinders(int c){
  NCylinders = c;
}

void Car::SetWeight(float w){
  weight = w;
}

void Car::SetHP(int hp){
  HP=hp;
}

float Car::Specific_Power(){
  return HP/weight;
}
```

In this example, we can notice:

- Encapsulation
- Class Definition
- Methods
- Object Instantiation

```
int main(){

  cout << "Hello World!" << endl;

  Car Toyota;
  Toyota.SetWeight(1200.1);
  Toyota.SetHP(200);

  cout << "Specific Power = " << Toyota.Specific_Power() << endl;

  return 0;

}
```

The "this" pointer

```
class Box
{
   public:
     Box(double l=2.0, double b=2.0, double h=2.0)
      {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
      }
      double Volume()
      {
         return length * breadth * height;
      }
      int compare(Box box)
      {
         return this->Volume() > box.Volume();
      }
   private:
      double length;
      double breadth;
      double height;
};
```

In this example:

- "this" pointer
- "inline" method construction
- default parameters

(Multiple) Inheritance and Multiple Constructors

```
class Shape
{
   public:
      Shape();
      Shape(int w, int h);
      void setWidth(int w)
      {
         width = w;
      }
      void setHeight(int h)
      {
         height = h;
      }
   private:
      int width;
      int height;
};


class Rectangle: public Shape
{
   public:
      int getArea()
      {
         return (width * height);
      }
};
```

Polymorphism

Polymorphism ("multiple form") occurs when there is a hierarchy of classes generated by inheritance.

Polymorphism implies that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Example:

```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      int area()
      {
         cout << "Parent class area" <<endl;
         return 0;
      }
};
```

```
class Rectangle: public Shape{
   public:
      Rectangle( int a=0, int b=0):Shape(a, b) { }
      int area ()
      {
         cout << "Rectangle class area :" <<endl;
         return (width * height);
      }
};
```

Polymorphism

```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      int area()
      {
         cout << "Parent class area" <<endl;
         return 0;
      }
};
```

```
class Rectangle: public Shape{
   public:
      Rectangle( int a=0, int b=0):Shape(a, b) { }
      int area ()
      {
         cout << "Rectangle class area :" <<endl;
         return (width * height);
      }
};
```

Notice that the “area” function is present in the parent and in the derived class. When we instantiate a “rectangle”, which area function will be called?

```
int main( ){
  Shape *shape;
  Rectangle rec(10,7);

  shape = &rec;
  shape->area();
}
```

If we store the address of rec in a shape pointer, the “area” of shape will be called.

Polymorphism solution

```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      virtual int area()
      {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
```

A virtual function is a function in a base class that is declared using the keyword **virtual**. Defining a virtual function with another version in a derived class, signals to the compiler that we don't want **static linkage**. What we do want is choosing the function to be called at a given point in the program to be based on the kind of object for which is called. This is referred to as **dynamic linkage** (or late binding).

Purely Virtual Functions

```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      // pure virtual function
      virtual int area() = 0;
};
```

Purely virtual functions are defined in a base class without any implementation. An implementation is required in the derived classes.

Summary

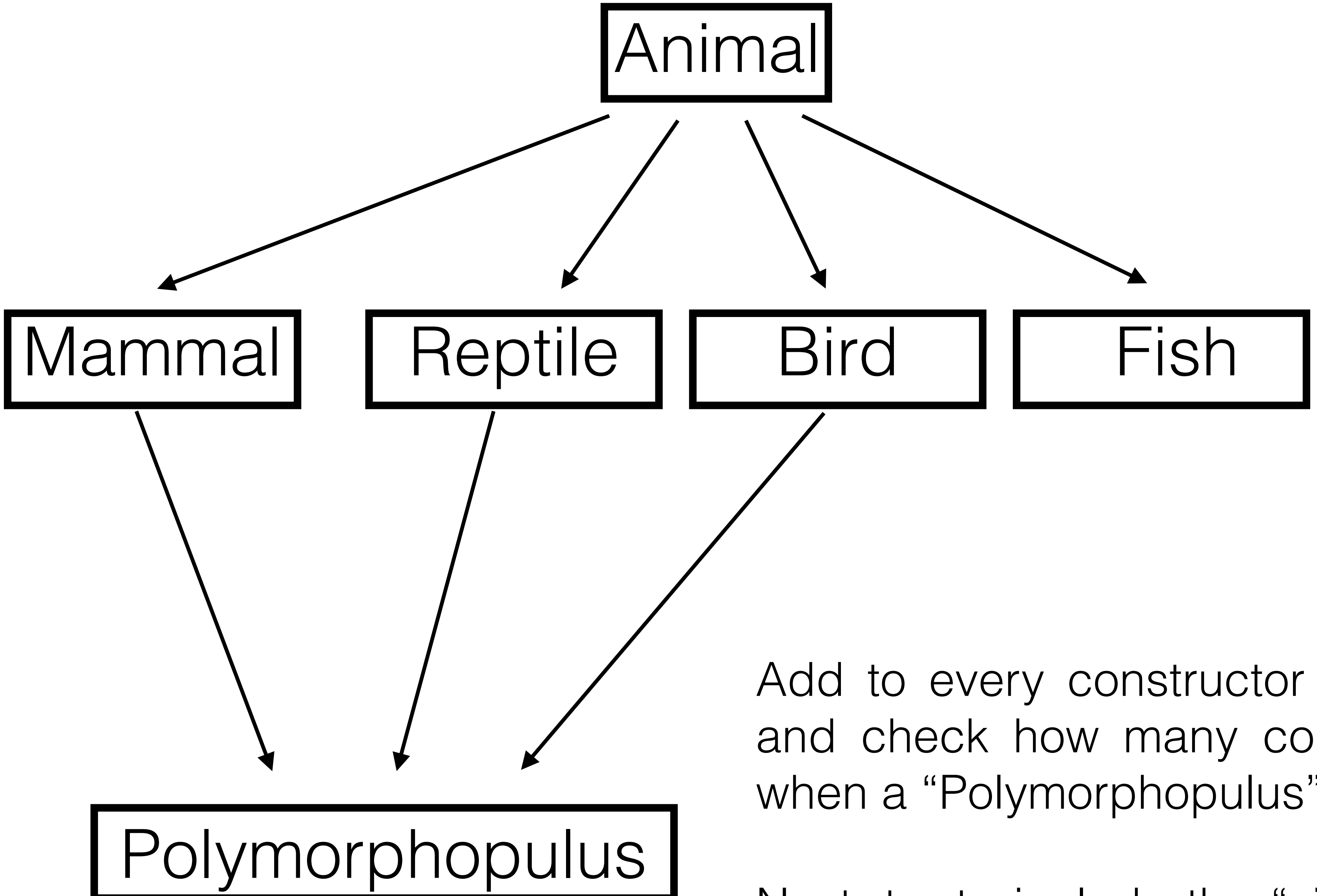
- Programming Paradigms

With C++ as example:

- Classes and Objects: Instantiation
- Methods
- Encapsulation
- Inheritance
- Polymorphism

EXERCISES

1) Write a Class implementing the type "Animal" with the protected data: name, age, size.
2) Add the corresponding constructor
3) Add an additional constructor with data initialization
4) Add the following methods:
   Set/Get methods.
   A purely virtual method which prints the animal's group: Group()
5) Derive the following classes:
   - Mammal (add IsVegetarian data/methods)
   - Reptile (add Nlegs data/method),
   - Bird (add Colour data/method),
   - Fish (add Sea data/method),
   - Amphibian.
   For all of them, define the corresponding Group() method.
5) Instantiate the class and test it!
6) Create the class corresponding to an animal which is at the same time mammal, reptile and Bird. Solve the so-called "<u>diamond problem</u>" of multiple inheritance.



Add to every constructor a print-out message and check how many constructors are called when a “Polymorphopulus” is instantiated.

Next, try to include the “virtual” keyword in the inheritance statement, eg:

```
class Reptile:virtual Animal { … };
```

The “virtual” keyword

In C++, “virtual” is used in different contexts:

1) A pure virtual function can be overridden by a function with the same name in a derived class. This is the case when we create **abstract base classes**:

```
class AbstractClass {
    public:
        virtual void AbsClassMethod() = 0;
};
```

This forces all the derived classes to implement such a method.

2) See example before (the “diamond problem”): a class at a lower level of inheritance when instantiated calls only one instance of the original base class.

Virtual destructors

When a delete statement is used to delete a pointer of type base class which actually points to derivedclass, what is really destroyed?
The solution is to use a virtual destructor in the base class:

```
class BaseClass {
….
virtual ~BaseClass(); //virtual destructor

};
```

Copy Constructors

For ensuring a deep copy of an object, C++ does not provide ways to define virtual constructors, so a virtual “clone” functions has to be specified.

```
class BaseClass
{
   public:
      virtual BaseClass* Clone() const = 0;
}

class DerivedClass : public BaseClass
{
   DerivedClass* Clone(){
      return new DerivedClass (*this);
   }
};
```