# Object-oriented programming

## Lecture #9: Polymorphism

# Outline

- Definition and Different Types of Polymorphism
- (Inclusion) Polymorphism and Dynamic Binding

# Literal Meaning

- Polymorphism:
  - *poly* ~ many/different
  - *morph* ~ forms/shapes

- *Definition: the ability to assign a different meaning or usage to something in different contexts*

# Types of Polymorphism

- Overloading
- Coercion
- Parametric Polymorphism
  - A concept originated from functional programming
  - Now popular in Java as a form of generic programming
- **Inclusion Polymorphism or *Overriding***
  - In the context of Object Oriented Programming (OOP), it is *the* Polymorphism

# Overloading

- We *overload* when we want to do “essentially the **same thing**”, but **with different parameters**

```
int add(int a, int b) { return a + b; }
float add(float a, float b} { return 1.0 + a + b; }
```

# Coercion

- An object or a primitive is (automatically) cast into another object or primitive type (more than just overloading)

```
int add(int a, int b) { return a + b; }
float add(float a, float b) { return 1.0 + a + b; }
int main() {
    std::cout << add(1, 1.0) << endl;
}
```

# Parametric Polymorphism

- Provides means to execute the same code for any type
- In C++ parametric polymorphism is implemented via templates.

# Inclusion Polymorphism

## Encapsulation

- Bundling data and associated functionalities
- Hide internal details and restricting access

## Inheritance

- Deriving a class from another, affording code reuse

## Abstraction

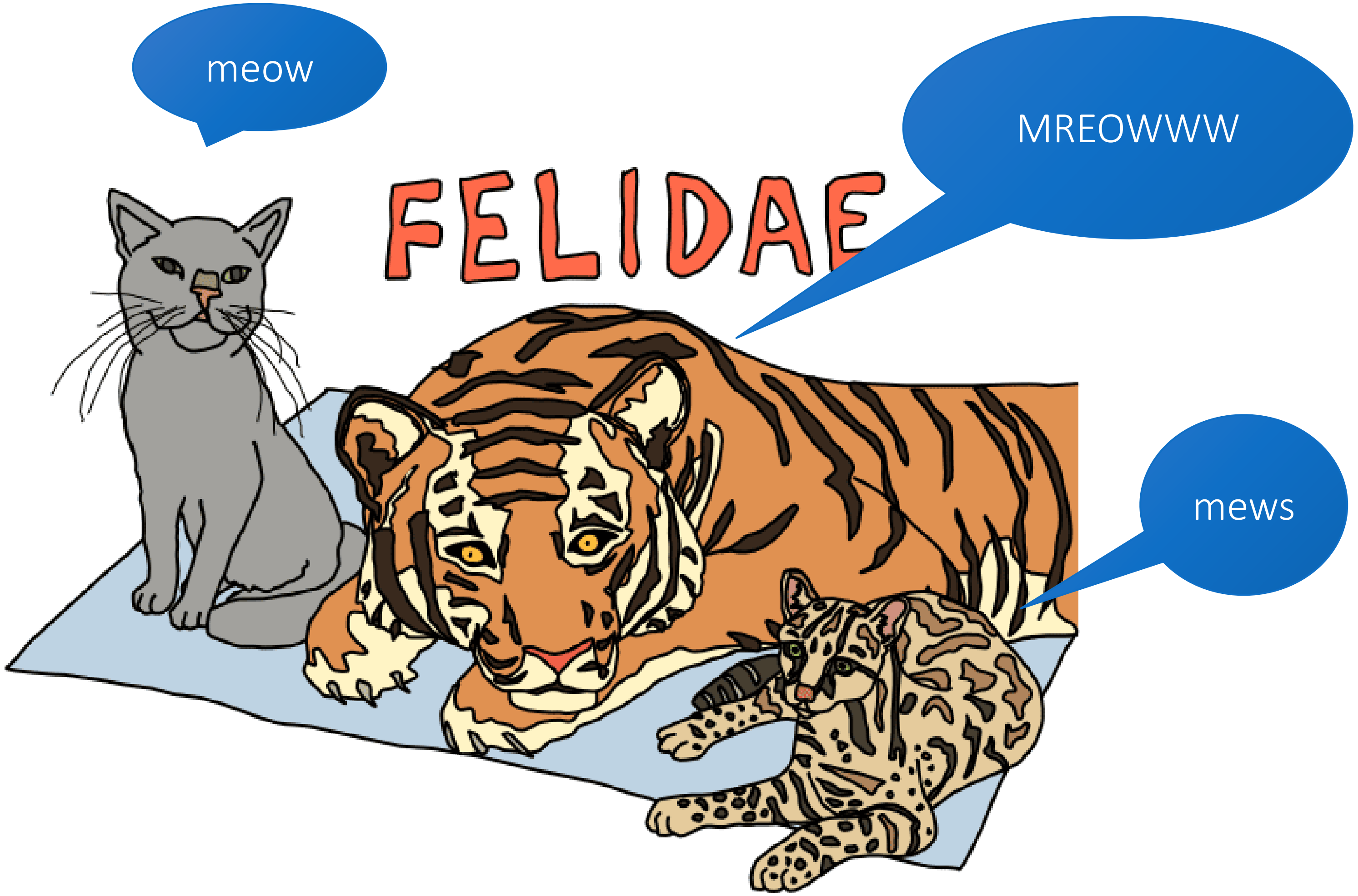
- Hiding the complexity of the implementation
- Focusing on the specifications and not the implementation details

## Polymorphism or Overriding

# Polymorphism

- Different types of objects respond differently to the same function call (and same parameter types)
- This is achieved by *overriding*
- We *override* an inherited function when we want to do something *slightly different* than what in the base class
- Note that this highly connected to the concept of inheritance

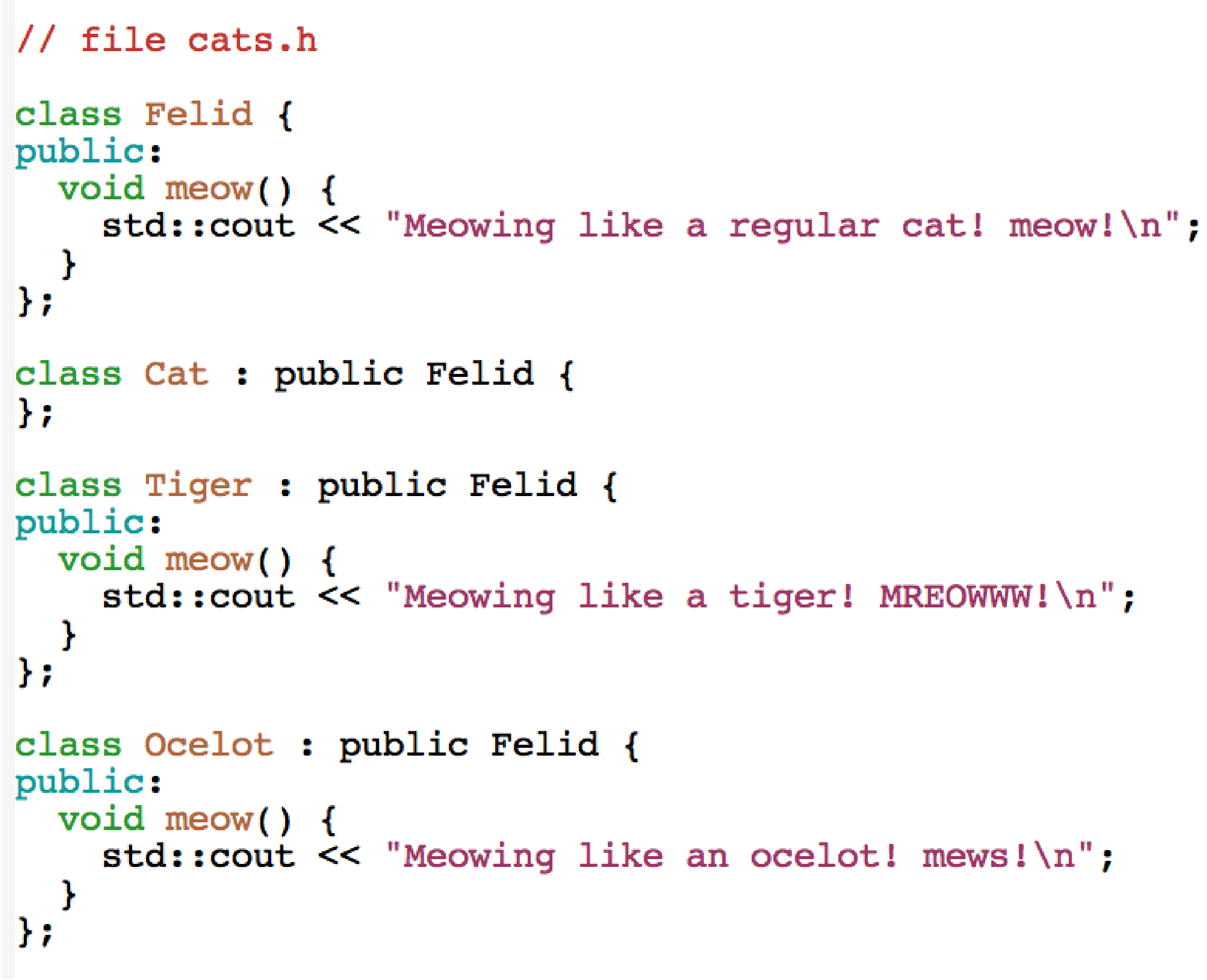
# Illustration for (Inclusion) Polymorphism



# Binding

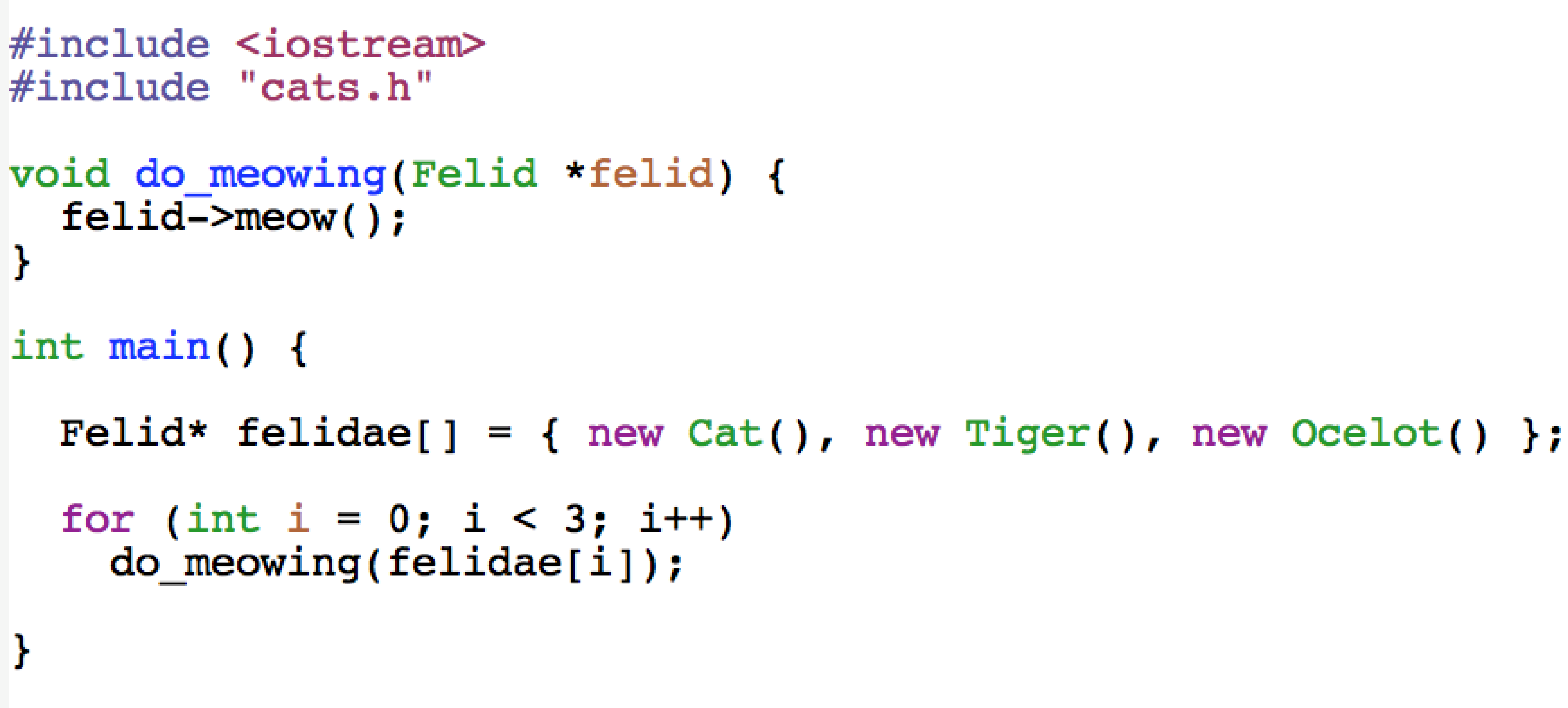
- Connecting a method call to a method body is called *binding*
- When binding is performed before the program is run, it is called *static*/*early* binding
  - e.g. in C compilers
- Binding occurs at run-time, based on the type of object, is called *dynamic*/*late* binding
- Polymorphism is a concept; Dynamic binding is a description of how that concept is implemented

# Static Binding in C++

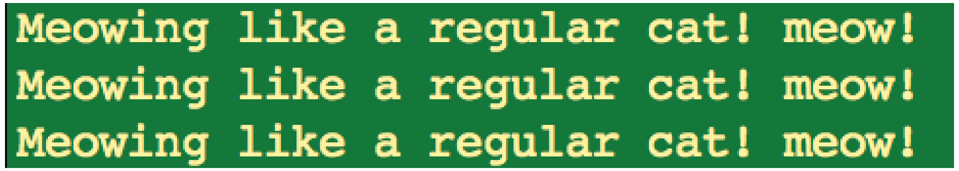


```cpp
// file cats.h

class Felid {
public:
  void meow() {
    std::cout << "Meowing like a regular cat! meow!\n";
  }
};

class Cat : public Felid {
};

class Tiger : public Felid {
public:
  void meow() {
    std::cout << "Meowing like a tiger! MREOWWW!\n";
  }
};

class Ocelot : public Felid {
public:
  void meow() {
    std::cout << "Meowing like an ocelot! mews!\n";
  }
};
```

# Static Binding in C++

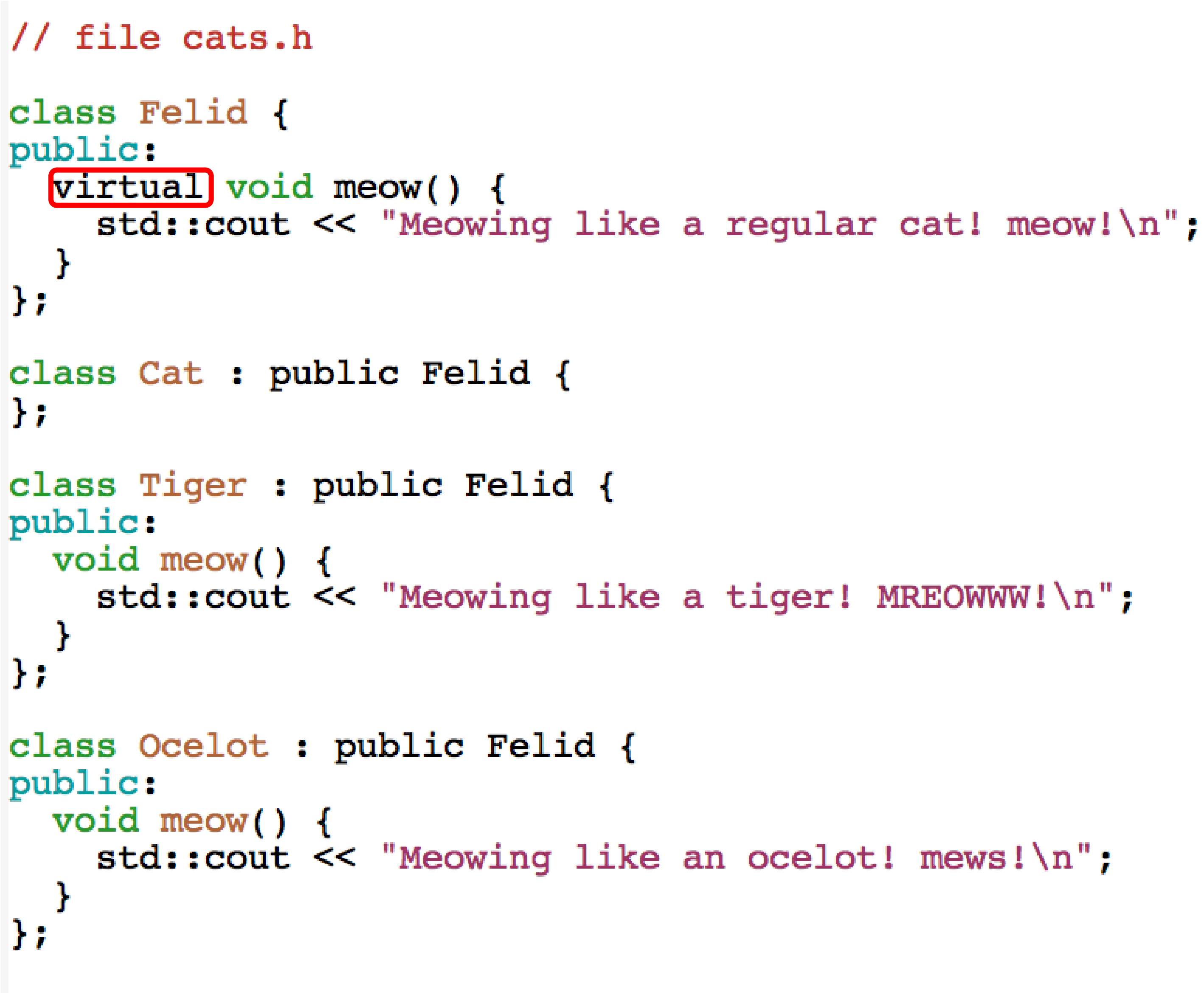


```cpp
#include <iostream>
#include "cats.h"

void do_meowing(Felid *felid) {
  felid->meow();
}

int main() {

  Felid* felidae[] = { new Cat(), new Tiger(), new Ocelot() };

  for (int i = 0; i < 3; i++)
    do_meowing(felidae[i]);

}
```



```
Meowing like a regular cat! meow!
Meowing like a regular cat! meow!
Meowing like a regular cat! meow!
```

# Dynamic Binding in C++



```
// file cats.h

class Felid {
public:
  virtual void meow() {
    std::cout << "Meowing like a regular cat! meow!\n";
  }
};

class Cat : public Felid {
};

class Tiger : public Felid {
public:
  void meow() {
    std::cout << "Meowing like a tiger! MREOWWW!\n";
  }
};

class Ocelot : public Felid {
public:
  void meow() {
    std::cout << "Meowing like an ocelot! mews!\n";
  }
};
```

# Dynamic Binding in C++

- In C++, methods are non-virtual by default. They can be made virtual by using ***virtual*** keyword
- This design decision is due to the fact that C++ was made to be *almost as efficient as* C
  - Dynamic binding is more costly than static binding
  - “If you don’t use it, you don’t pay for it”
- If a method cannot be overridden, it then can be statically bound (and/or made inline)
  - More compiler optimization techniques can be applied to such methods

# Dynamic Binding in C++

- Virtual Function
  - A non-static member function prefaced by the *virtual* keyword
  - It tells the compiler to generate code that selects the appropriate version of this function at run-time

# Importance of Polymorphism

- When we have a collection of generic reference variables and each is assigned to a different type of object
  - We can step through the collection and for each element, calling a polymorphic function (e.g. `meow()`)
  - The runtime system picks out the appropriate method bodies that apply to the different types of objects

# Importance of Polymorphism

- When new derived class (subclass) is added, we do not need to change the existing code. Example:
  - We add a new class `Cheetah`
  - This should not affect the code defining the previous classes and the method `doMeowing()`

# Virtual Destructors

- See Lec9_ex1-VirtualDes.cpp
- Calling the wrong destructor could be disastrous, particularly if it contains a *delete* statement
- Destructors are not inherited
- Always make base classes' destructors virtual when they're meant to be manipulated polymorphically

# Abstract Classes

- In C++, an interface describes the **behavior or capabilities** of a class **without committing to a particular implementation of that class**.
- The C++ interfaces are implemented using **abstract classes** and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data
- A class is made abstract by declaring at least one of its functions as **pure virtual** function. *A pure virtual function is specified by placing "= 0" in its declaration*

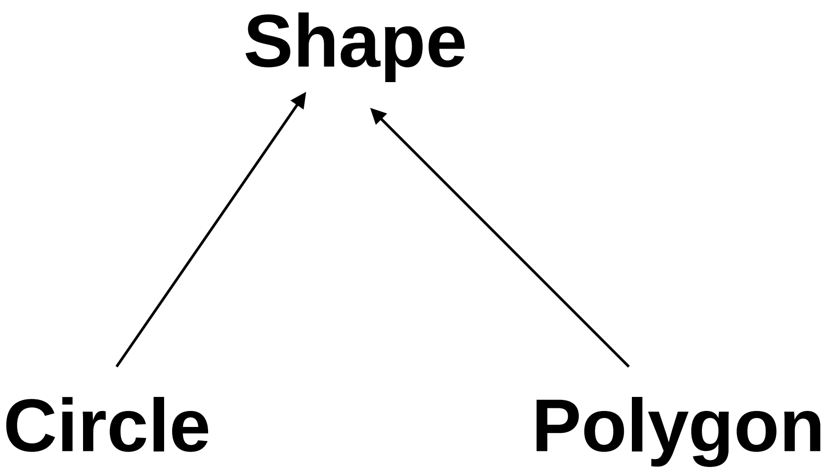
# Pure Virtual Function

```
class Box {
   public:
      // pure virtual function
      virtual double getVolume() = 0;
   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};
```

# Abstract Classes

- The purpose of an **abstract class** is to provide an appropriate base class from which other classes can inherit. Abstract classes often represent concepts for which objects cannot exist
- Abstract classes cannot be used to instantiate objects and serves only as an **interface**. Attempting to instantiate an object of an abstract class causes a compilation error
- In contrast, classes that can be used to instantiate objects are called **concrete classes**

# Example: Lec9_ex2-Shapes.cpp



Shape

Circle

Polygon

# An Abstract Derived Class

- If a pure virtual function is not defined in a derived class, the derived class is also considered an abstract class
- When a derived class does not provide an implementation of a virtual function the base class implementation is used
- It is possible to declare pointer variables to abstract classes

# “this” Keyword

- Within a member function of a class, “this” is the name of an implicit pointer to the current object which receives the message associated to this function
- “this” is really a ***polymorphic*** word which can mean any object in the C++ language

# Summary

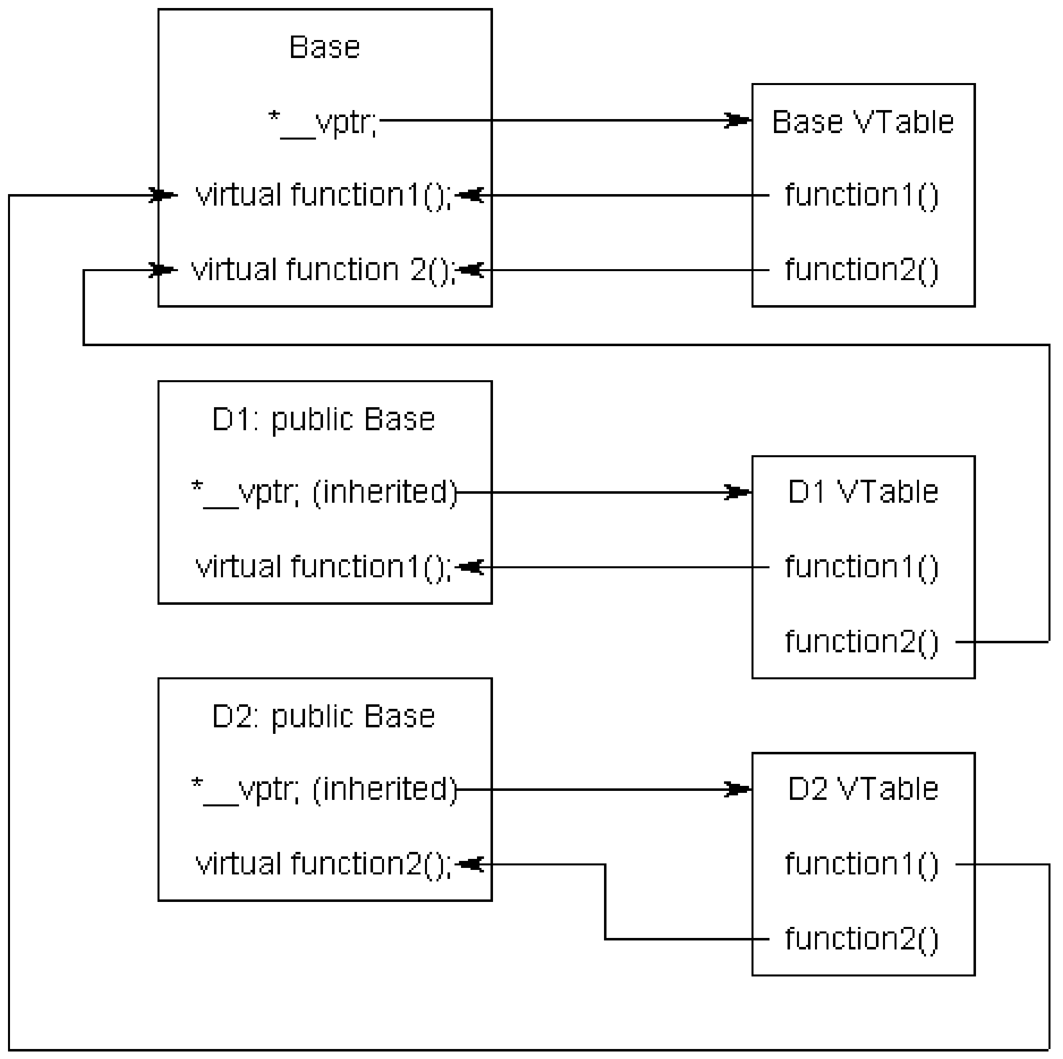
- Four types of Polymorphism
  - Runtime polymorphism
  - Parametric polymorphism (compile time polymer.)=>Templates
  - Ad-hoc polymorphism (Overloading)
  - Coercion polymorphism (Casting)
- (Inclusion) Polymorphism or Overriding
  - Why important?
  - Do not confuse with overloading
  - Requires dynamic binding
  - Static binding as default, virtual keyword

# Virtual Tables

- C++ (and Java) use the virtual table (`vtable`) mechanism to implement dynamic binding
  - A class with virtual member functions has a virtual table which contains the address of its virtual functions
  - An object of such a class has a pointer (`vptr`) to point to the virtual table of the class
  - Dynamic binding is done by

1. *Following the `vptr` to reach the virtual table of the class*
2. *Looking up the virtual table for the entry point of the appropriate function at run-time*

# Virtual Tables


Base
*__vptr;
virtual function1();
virtual function 2();
Base VTable
function1()
function2()
D1: public Base
*__vptr; (inherited)
virtual function1();
D1 VTable
function1()
function2()
D2: public Base
*__vptr; (inherited)
virtual function2();
D2 VTable
function1()
function2()