Object-Oriented Programming

- Object-oriented programming views a program as collection of agents, termed objects. Each object is responsible for specific tasks.
- An object is an encapsulation of state (data members) and behavior (operations).
- The behavior of objects is dictated by the object class.
- An object will exhibit its behavior by invoking a method (similar to executing a procedure).

- Objects and classes extend the concept of abstract data types by adding the notion of inheritance.

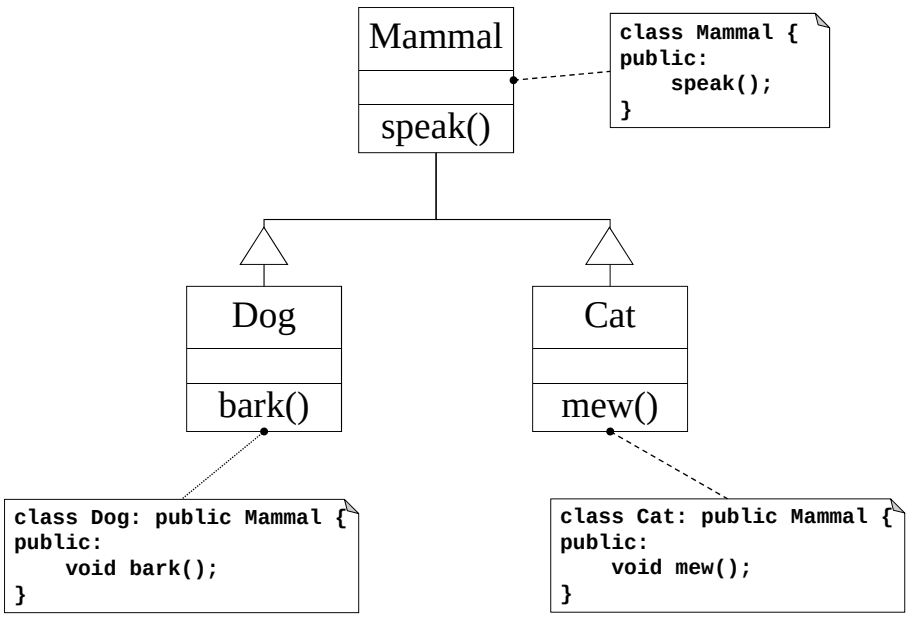
Inheritance in C++

`base / derived` class

`public / private` inheritance

`non-virtual / virtual / pure virtual` member functions

`concrete / abstract` classes



Public Inheritance, Non-Virtual Members

```
class Mammal { // base class
public:
    void speak() {cout << "can't speak" << endl;}
};

class Dog : public Mammal { // derived class
public:
    void speak() {cout << "wouf" << endl;}
    void bark() {cout << "wouf wouf" << endl;}
};

class Cat : public Mammal { // derived class
public:
    void mew() {cout << "mew mew" << endl;}
};
```

```
Mammal fred;
fred.speak();

Dog lassie;
lassie.speak();
lassie.bark();

Cat sue;
sue.speak();
sue.mew();

Mammal* x = new Dog();
x->speak();

x = &sue;
x->speak();
```

Public Inheritance, Virtual Members

```
class Mammal { // base class
public:
    virtual void speak() {cout << "can't speak"   << endl;}
};

class Dog : public Mammal { // derived class
public:
    void speak() {cout << "wouf" << endl;}
    void bark() {cout << "wouf wouf" << endl;}
};

class Cat : public Mammal { // derived class
public:
    void mew() {cout << "mew mew" << endl;}
};
```

Virtual member functions to support polymorphism.

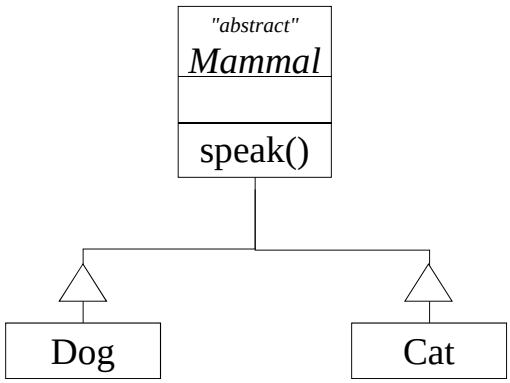
```
Mammal fred;
fred.speak();

Dog lassie;
lassie.speak();

Cat sue;
sue.speak();

Mammal* x = new Dog();
x->speak();

x = &sue;
x->speak();
```

Public Inheritance, Pure Virtual Members

```
class Mammal { // abstract base class
public:
    virtual void speak() = 0;
};

class Dog : public Mammal { // derived class
public:
    void speak() {cout << "wouf" << endl;}
    void bark() {cout << "wouf wouf" << endl;}
};

class Cat : public Mammal { // derived class
public:
    void speak() {cout << "mew" << endl;}
    void mew() {cout << "mew mew" << endl;}
};
```

Abstract classes serve as interfaces.

```
// Mammal fred;
// fred.speak();

Dog lassie;
lassie.speak();

Cat sue;
sue.speak();

Mammal* x[n];
x[0] = &lassie;
x[1] = &sue;

for(int i=0;i<n;i++) {
    x[i]->speak();
}

x[0]->bark(); // Error
```



Private Inheritance

```
class Mammal {
public:
    void speak() {cout << "can't speak" << endl;}
};

class Dog : private Mammal {
public:
    void bark() {speak();}
};

class Cat : public Mammal {
public:
    void mew() {cout << "mew mew" << endl;}
}
```

```
Mammal fred;
fred.speak();

Dog lassie;
lassie.bark();
lassie.speak(); // Error

Cat sue;
sue.speak();
```

Private inheritance to share implementation.

Public Inheritance, Virtual Members

```
class Mammal { // base class
public:
    virtual void speak() {cout << "can't speak"   << endl;}
};

class Dog : public Mammal { // derived class
public:
    void speak() {cout << "wouf" << endl;}
    void bark() {cout << "wouf wouf" << endl;}
};

class Cat : public Mammal { // derived class
public:
    void mew() {cout << "mew mew" << endl;}
};
```

Virtual member functions to support polymorphism.

```
Mammal fred;
fred.speak();

Dog lassie;
lassie.speak();

Cat sue;
sue.speak();

Mammal* x = new Dog();
x->speak();

x = &sue;
x->speak();
```

Access to Base Class Members

```
class One {
public:
    int element1;
protected:
    int element2;
private:
    int element3;
};
class Two : public One {

};

clients of Two
clients of Three
clients of Four
...
```

```
class Three : private One {

};

class Four : protected One {

};

class Five : public Three {

};

class Six: public Four {

};
```

dynamic_cast<T>(), typeinfo

```
#include <typeinfo>

Dog lassie;
Cat sue;

Mammal* m[3];
m[0] = &lassie;
m[1] = &sue;
...

for ( i=0;i<3;i++) cout << typeid(x[i]).name() << ", ";
cout << endl;

for ( i=0;i<3;i++){
    Dog* d = dynamic_cast<Dog *>(x[i]);
    if(d) d->bark();
}
```

Forms of Inheritance

Specialization: The derived class is a subtype, a special case of the base class.

Specification: The base class defines behavior that is implemented in the derived class.

Construction: The derived class makes use of the behavior provided by the base class, but is not a subtype of the base class.

Generalization: The derived class modifies some of the methods of the base.

Extension: The derived class adds new functionality to the base, but does not change any inherited behavior.

Limitation: The derived class restricts the use of some of the behavior inherited from the base.

Variance: The derived class and the base class are variants of each other, and the base/derived class relationship is arbitrary.

Combination: The derived class inherits from more than one base class.