Inheritance Overview

- Classes can **inherit** from other classes
  - Properties (variables)
  - Behavior (methods)

- Inheritance serves various functions
  - Modeling of class relationships
  - Code reuse
  - Subtyping/polymorphism

Inheritance Example

```cpp
class animal {
    public:
        string name;
        void print();
};
class dog : public animal {
    public:
        string breed;
};
```

This signifies that dog inherits from animal. (The "public" just means all members have the same visibility in the subclass as in the superclass.)

Superclass aka "base" or "parent" class.
Subclass aka "derived" or "child" class.

Inheritance: Modeling Relationships

```cpp
class cat : public animal {
};
```

We say cat "is a" type of animal*

*This language can lead to bad modeling choices. E.g., a square "is a" type of rectangle. If we model this way in C++, a natural choice is to give rectangle properties of height and width. If square inherits from rectangle, it gets those two independent properties, but in a square, they must be identical. So not every "is a" relationship in real life makes sense in C++!

Inheritance: Properties

Note that animal defined a property:

```cpp
    string name;
```

This is inherited by dog and cat.

We can use name in dog and cat because it was defined by the superclass:

```cpp
dog d;
cat c;
d.name = "Rex";
c.name = "Fluffy";
```

Inheritance: Properties

Note that dog defines a new property,

```cpp
    string breed;
```

This is unique to dog; we can't use it in animal or cat:

```cpp
dog d;
cat c;
d.breed = "Dachshund";
c.breed = "Tabby";   // error!
```
Inheritance: Behavior

Behaviors can also be inherited, leading to very powerful code reuse.
E.g.,
```cpp
void animal::print() {
    cout << "My name is " << name << "." << endl;
}
```
Defines a reasonable print behavior for cat and dog.

Inheritance: Overrides

If we don't like the superclass behavior, we can change it in the subclass:
```cpp
class dog : public animal {
    public:
        string breed;
        void print();
}
void dog::print() {
    cout << "My name is " << name << "." << endl;
    cout << "I am a " << breed << "." << endl;
}
```
You cannot:
- Override properties
- Change the return type of methods

Inheritance: Calling on the Super

We can improve our print() method slightly by reusing the superclass behavior:
```cpp
dog::print() {
    animal::print();
    cout << "I am a " << breed << "." << endl;
}
```

Example So Far

```cpp
dog d;
cat c;
d.name = "Rex";
d.breed = "Dachshund";
c.name = "Fluffy";
c.print();
d.print();
```
Output is:
My name is Fluffy.
My name is Rex.
I am a Dachshund.

Inheritance: Polymorphism I

Note we can now use dogs and cats wherever we would use an animal:
```cpp
... 
    void print_animal(animal &a) { a.print(); }
    print_animal(c);
    print_animal(d);
...
```
What does this output?
(Hint: it is different from previous page!)

Inheritance: Polymorphism II

Let's fix this:
```cpp
class animal {
    public:
        string name;
        virtual void print();
};
print_animal(c);
print_animal(d);
```
This gives us the same output as:
```cpp
c.print();
d.print();
```
Polymorphism

- The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.
- C++ polymorphism means 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.

Inheritance: Abstract Classes

An abstract class is one which:
- Contains at least one “pure virtual” method
- Cannot be instantiated
- Can only be used via inheritance

```cpp
class animal {
public:
    string name;
    virtual void print();
    virtual void speak() = 0;
};
```

Notation to designate as a pure virtual method.

Abstract Classes II

Pure virtual methods *are not defined in the abstract class.*

(Non-abstract) children of abstract classes **must** implement any pure virtual methods.

However, we can use pure virtual methods in the abstract class:
```cpp
void animal::print() {
    cout << "My name is " << name << ". ";
    speak();
    cout << endl;
}
```

Inheritance: Constructors

- Normally, a subclass calls the default constructor (i.e. no parameters) of the superclass before executing its own constructor.
- You can force the subclass to call a different constructor using this form in the definition:

```cpp
animal::animal(string nm) { name = nm; }
dog::dog(string n, string b) : animal(n) {
    breed = b;
}
```

Final Example

```cpp
class animal {
public:
    string name;
    virtual void print();
    virtual void speak() = 0;
};
class dog : public animal {
public:
    string breed;
    virtual void print();
    virtual void speak() { cout << "Woof!"; }
};
class cat : public animal {
public:
    virtual void speak() { cout << "Meow."; }
};
```
Final Example II

```cpp
void animal::print() {
    cout << "My name is " << name << "; " << endl;
    speak();
    cout << endl;
}

void dog::print() {
    animal::print();
    cout << "I am a " << breed << "; " << endl;
}

void print_animal(animal& a) { a.print(); }
```

Final Example III

```cpp
int main() {
    dog d;
    cat c;
    d.name = "Rex";
    d.breed = "Dachshund";
    c.name = "Fluffy";
    print_animal(c);
    print_animal(d);
    return 0;
}
```

Final Example Output

My name is Fluffy. Meow.
My name is Rex. Woof!
I am a Dachshund.

Up Next

- Wednesday, March 14
  - Recursion
  - Review Chapter 9
- Friday, March 16
  - Lab 9 – Queues Revisited
- Monday, March 19
  - Analysis of Algorithms 1
  - Read Chapter 15