CSCI 262
Data Structures

20 – Inheritance

CS@Mines

Inheritance Overview

- Classes can *inherit* from other classes
  - Properties (variables)
  - Behavior (methods)
- Inheritance serves various functions
  - Modeling of class relationships
  - Code reuse
  - Subtyping/polymorphism

CS@Mines

Inheritance Example

Superclass aka "base" or "parent" class.

Subclass aka "derived" or "child" class.

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

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

class dog : public animal {
public:
      string breed;
};
```

CS@Mines

Inheritance: Modeling Relationships

```
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 these two independent properties, but in a square, they must be identical. So not every "is a" relationship in real life makes sense in C++!

CS@Mines

Inheritance: Properties

Note that animal defined a property:

```
string name;
```

This is inherited by dog and cat.

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

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

CS@Mines

Inheritance: Properties

Note that dog defines a new property,

```
string breed;
```

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

```
dog d;
cat c;
d.breed = "Dachshund";
c.breed = "Tabby";
```

error!

CS@Mines

Inheritance: Behavior

Behaviors can also be inherited, leading to very powerful code reuse.

E.g.,

```
void animal::print() {
      cout << "My name is " << name << ". ";
      cout << 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:

```
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:

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


Example So Far

```
…
dog d;
cat c;
d.name = "Rex";
d.breed = "Dachshund";
c.name = "Fluffy";
c.print();
d.print();
…
```

I encourage you to try these code snippets for yourself, and modify them to see what else you can do.

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:

```
…
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:

```
class animal {
public:
    string name;
    virtual void print();
};

print_animal(c);
print_animal(d);
```

This gives us the same output as:

```
c.print();
d.print();
```

Inheritance: Polymorphism III

Another approach, same output:

```
animal* A[2];
A[0] = &c;
A[1] = &d;
for (int j = 0; j < 2; j++) A[j]->print();
```

Note, how this is different:

```
animal a = d;  // default copy constructor called!
a.print();
```

Output is:

```
My name is Rex.
```

CS@Mines

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.

Polymorphism in C++ - tutorialspoint.com. (n.d.). Retrieved October 25, 2016, from https://www.tutorialspoint.com/cplusplus/cpp_polymorphism.htm

CS@Mines

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

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

Notation to designate as a pure virtual method.

CS@Mines

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:

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

CS@Mines

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:

```
animal::animal(string nm) { name = nm; }

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

Superclass constructor call

CS@Mines

Final Example

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

class dog : public animal {
public:
        string breed;
        void print();
        void speak() { cout << "Woof!"; }
};

class cat : public animal {
public:
        void speak() { cout << "Meow."; }
};
```

CS@Mines

Final Example II

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

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

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


Final Example III

```
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, December 5
  - Final Review