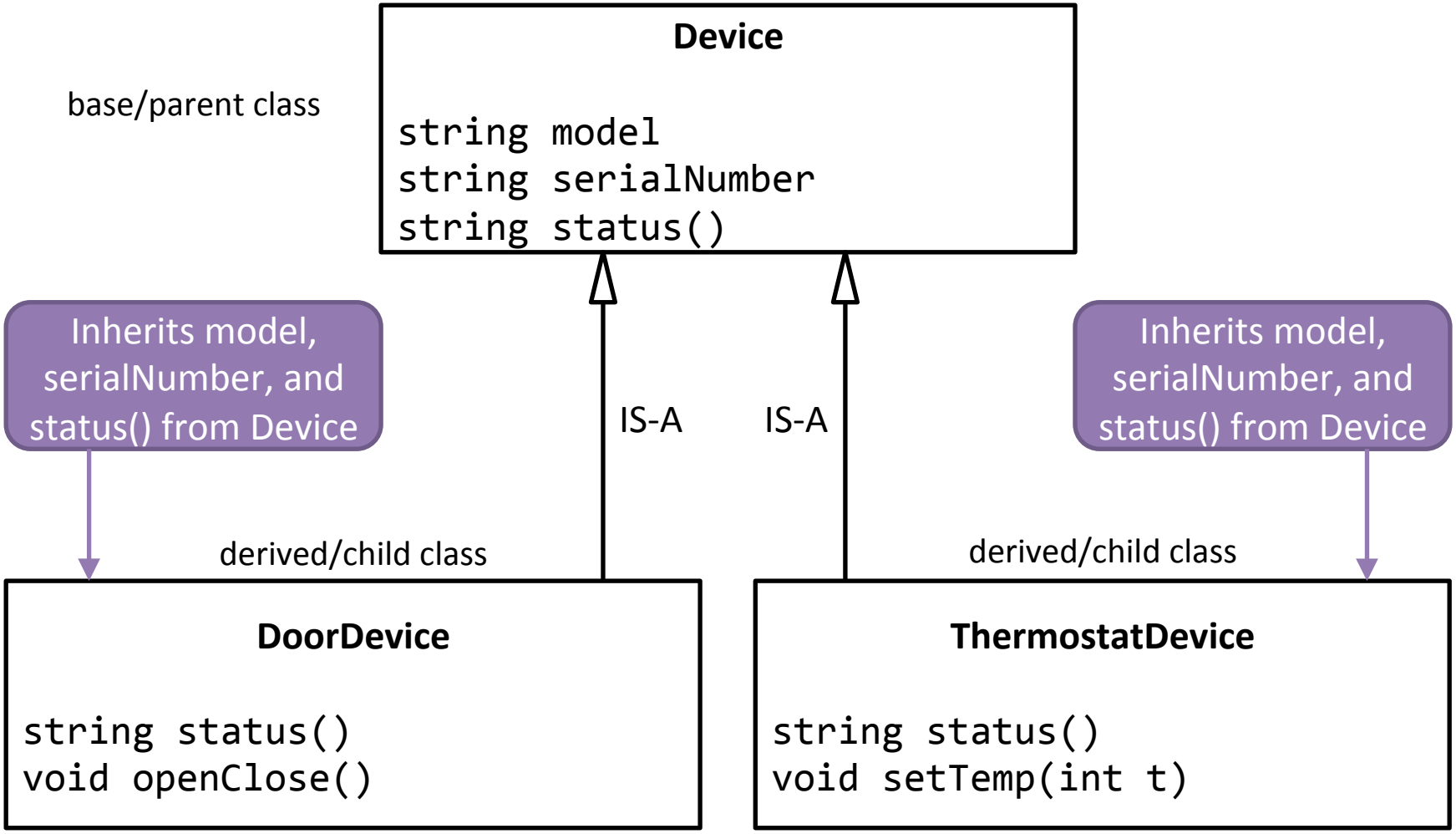
Inheritance & Polymorphism

Lecture 13

What is Inheritance?

- The process by which a new class, called a **derived** class, is created from another class, called the **base** class
- A derived class automatically has all the member variables and functions of the base class
  - Depending on access level (more later)
- A derived class can have additional member variables and/or member functions
- The derived class is a *child* of the base, or *parent,* class; often referred to as an *IS-A* relationship or ancestor/descendent

Example Inheritance Hierarchy



Benefits of Inheritance

- Code re-use
  - Build layers of complex code without having to redefine common elements
  - For example, build thermostat devices without having to redefine model/serial number
- Work uniformly with multiple related types
  - Write code that depends upon a common interface, independent of implementation
  - For example ask for the status of any device

Example

```
class Device
{
public:
	string model;
	string serialNumber;
	string status() { return "???"; }
};

class DoorDevice: public Device
{
public:
	DoorDevice() { open = false; }
	void openClose() { open = !open; }

private:
	bool open;
};
```

```
int main()
{
	Device d;
	d.model = "model1";
	d.serialNumber = "sn1";

	DoorDevice dd;
	dd.model = "model2";
	dd.serialNumber = "sn2";
	dd.openClose();

	cout << d.model << " " << d.serialNumber
		<< " " << d.status() << endl;
	cout << dd.model << " " << dd.serialNumber
		<< " " << dd.status() << endl;

	return 0;
}
```

```
> ./d
model1 sn1 ???
model2 sn2 ???
```

Derived Classes

- To create a derived class, start like any other class, but after the name, add a colon and then the name of the base class

```
class DoorDevice: public Device
```

- The derived class automatically *inherits* all non-**`private`** member functions/variables
  - And can also add additional member functions/variables (e.g. **`openClose()`**)

Redefining Member Functions

- For all functions the derived class chooses to inherit, no re-declaration/definition is necessary

- However, a derived class may require a different *implementation* for an inherited member function

Example

```
class Device
{
public:
	string model;
	string serialNumber;
	string status() { return "???"; }
};

class DoorDevice: public Device
{
public:
	DoorDevice() { open = false; }
	void openClose() { open = !open; }
	string status()
	{
		if ( open ) return "Open";
		else return "Closed";
	}

private:
	bool open;
};
```

```
int main()
{
	Device d;
	d.model = "model1";
	d.serialNumber = "sn1";

	DoorDevice dd;
	dd.model = "model2";
	dd.serialNumber = "sn2";
	dd.openClose();

	cout << d.model << " " << d.serialNumber
		<< " " << d.status() << endl;
	cout << dd.model << " " << dd.serialNumber
		<< " " << dd.status() << endl;

	return 0;
}
```

```
> ./d
model1 sn1 ???
model2 sn2 Open
```

Exercise

Create a **`Shape2D`** class that has a public **`name`** member variable (string) and a single member function **`area`**, which takes no arguments and returns a double (0). Create a derived **`Circle`** class that allows for getting/setting its radius and redefines the **`area`** function. Now create a main function with both a **`Shape2D`** object and a **`Circle`** object – output their names and areas.

Answer

```
class Shape2D
{
public:
     double area() { return 0.0; };
     string name;
};

class Circle: public Shape2D
{
public:
     double area()
     {
          return 3.14159*radius*radius;
     };
     double getRadius() { return radius; }
     void setRadius(double r) { radius = r; }

private:
     double radius;
};
```

```
int main()
{
     Shape2D s;
     s.name = "shape";

     Circle c;
     c.name = "circle";
     c.setRadius( 10.0 );

     cout << s.name << " "
          << s.area() << endl;

     cout << c.name << " ("
          << c.getRadius() << ") "
          << c.area() << endl;

     return 0;
}
```

```
> ./shapes
shape 0
circle (10) 314.159
```

Cleaning Up Device

The Device base class has public member variables (*danger!*)

To fix this, we need to learn two items

1. How constructors work in derived classes
2. Initialization lists

Class Construction

- In C++, whenever an object of a class is created, its constructor is called
- But that's not all – the object's parent class constructor is called, as are the constructors for all objects that belong to the class
- By default, the constructors invoked are the default constructors
  - Moreover, all of these constructors are called before the class's own constructor is called

Example

```
class Base
{
public:
     Base()
     {
          cout << "Base" << endl;
     }
};

class Derived: public Base
{
public:
     Derived()
     {
          cout << "Derived" << endl;
     }
};
```

```
int main()
{
     Derived d;
     return 0;
}
```

```
> ./prog
Base
Derived
```

Construction Order

- Each class should initialize things that belong to it, not things that belong to other classes
  - Derived class should hand off the work of constructing the portion of it that belongs to the base class
- The child class may depend upon inherited member variables when initializing its own member variables
  - The base class constructor needs to be called before the derived class's constructor runs
- But what if the base class needs constructor arguments?
  - Enter *initialization lists*

Initialization Lists

- An initialization list immediately follows the constructor's signature, separated by a colon

- An initialization list is used to call a derived class constructor and/or initialize member variable(s)

Example

```
class Base
{
public:
     Base(int x): y( x+1 )
     {
          cout << "Base: y="
               << y << endl;
     }

     int y;
};

class Derived: public Base
{
public:
     Derived(): Base( 7 )
     {
          cout << "Derived: y="
               << y << endl;
     }
};
```

```
int main()
{
     Derived d;
     return 0;
}
```

```
> ./prog
Base: y=8
Derived: y=8
```

Improvements to Device

```
class Device
{
public:
        Device(string model, string serialNumber):
        model( model ), serialNumber( serialNumber ) {}

        string status()
        {
                return ( model + " (" + serialNumber + "): ???" );
        }

        string getModel() { return model; }
        string getSN() { return serialNumber; }

private:
        string model;
        string serialNumber;
};

class DoorDevice: public Device
{
public:
        DoorDevice(string model, string serialNumber):
        Device( model, serialNumber ), open( false ) {}

        void openClose() { open = !open; }
        string status()
        {
                if ( open )
                        return ( getModel() + " (" + getSN() + "): Open" );
                else
                        return ( getModel() + " (" + getSN() + "): Closed" );
        }

private:
        bool open;
};
```

```
int main()
{
        Device d( "model1", "sn1" );

        DoorDevice dd( "model2", "sn2" );
        dd.openClose();

        cout << d.status() << endl;
        cout << dd.status() << endl;

        return 0;
}
```

```
> ./d
model1 (sn1): ???
model2 (sn2): Open
```

Exercise

Update your **`Shape2D`** and **`Circle`** classes to use constructors and private member variables.

Answer

```
class Shape2D
{
public:
      Shape2D(string name): name( name ) {}
      string getName() { return name; }
      double area() { return 0.0; };

private:
      string name;
};

class Circle: public Shape2D
{
public:
      Circle(string name, double radius):
      Shape2D( name ), radius( radius ) {}

      double area() { return 3.14159*radius*radius; };
      double getRadius() { return radius; }
      void setRadius(double r) { radius = r; }

private:
      double radius;
};
```

```
int main()
{
      Shape2D s( "shape" );
      Circle c( "circle", 10.0 );

      cout << s.getName() << " "
           << s.area() << endl;
      cout << c.getName() << " ("
           << c.getRadius() << ") "
           << c.area() << endl;

      return 0;
}
```

```
> ./shapes
shape 0
circle (10) 314.159
```

All in the Family

- Note that derived classes do not have access to **`private`** member variables/ functions of base classes
  - **`Circle`** cannot access **`Shape2D::name`**
  - **`DoorDevice`** cannot access **`Device::model`**

- The **`protected`** modifier allows you to specify member functions/variables that can be accessed by derived classes, but not outside code

Family-Friendly Device

```
class Device
{
public:
        Device(string model, string serialNumber):
        model( model ), serialNumber( serialNumber ) {}

        string status()
        {
                return ( model + " (" + serialNumber + "): ???" );
        }

        string getModel() { return model; }
        string getSN() { return serialNumber; }

protected:
        string model;
        string serialNumber;
};

class DoorDevice: public Device
{
public:
        DoorDevice(string model, string serialNumber):
        Device( model, serialNumber ), open( false ) {}

        void openClose() { open = !open; }
        string status()
        {
                if ( open )
                        return ( model + " (" + serialNumber + "): Open" );
                else
                        return ( model + " (" + serialNumber + "): Closed" );
        }

private:
        bool open;
};
```

```
int main()
{
        Device d( "model1", "sn1" );

        DoorDevice dd( "model2", "sn2" );
        dd.openClose();

        cout << d.status() << endl;
        cout << dd.status() << endl;

        return 0;
}
```

```
> ./d
model1 (sn1): ???
model2 (sn2): Open
```

Polymorphism

- The ability to associate multiple meanings to one function name
- One of the key components of OOP
- Implemented in C++ via **`virtual`** functions

Motivating Example

```
class Animal
{
public:
        Animal(string species, string name):
        species( species ), name( name ) {}

        string speak() { return "???"; };
        string getSpecies() { return species; }
        string getName() { return name; }

private:
        string species;
        string name;
};

class Dog: public Animal
{
public:
        Dog(string name): Animal( "dog", name ) {}
        string speak() { return "Woof"; }
};

class Cat: public Animal
{
public:
        Cat(string name): Animal( "cat", name ) {}
        string speak() { return "Meow"; }
};

class Cow: public Animal
{
public:
        Cow(string name): Animal( "cow", name ) {}
        string speak() { return "Moo"; }
};
```

```
int main()
{
        vector< Animal* > farm;
        farm.push_back( new Animal( "Animal", "Annie" ) );
        farm.push_back( new Dog( "Spot" ) );
        farm.push_back( new Cat( "Kitty" ) );
        farm.push_back( new Cow( "Happy" ) );

        for ( int i=0; i<farm.size(); i++ )
        {
                cout << farm[i]->getName() << " the "
                     << farm[i]->getSpecies() << " says "
                     << farm[i]->speak() << endl;
                delete farm[i];
        }

        return 0;
}
```

```
> ./farm
Annie the Animal says ???
Spot the dog says ???
Kitty the cat says ???
Happy the cow says ???
```

Virtual Functions

- When you make a function virtual, you are telling the compiler to wait until it is used in a program, and then get the implementation from the object instance
  - It uses the most derived form of the function
- Must be used at the base class
  - Optional in derived class (recommended)
- More expensive, so use when necessary
  - Implemented via *late binding*

Functioning Farm

```
class Animal
{
public:
        Animal(string species, string name):
        species( species ), name( name ) {}

        virtual string speak() { return "???"; };
        string getSpecies() { return species; }
        string getName() { return name; }

private:
        string species;
        string name;
};

class Dog: public Animal
{
public:
        Dog(string name): Animal( "dog", name ) {}
        virtual string speak() { return "Woof"; }
};

class Cat: public Animal
{
public:
        Cat(string name): Animal( "cat", name ) {}
        virtual string speak() { return "Meow"; }
};

class Cow: public Animal
{
public:
        Cow(string name): Animal( "cow", name ) {}
        virtual string speak() { return "Moo"; }
};
```

```
int main()
{
        vector< Animal* > farm;
        farm.push_back( new Animal( "Animal", "Annie" ) );
        farm.push_back( new Dog( "Spot" ) );
        farm.push_back( new Cat( "Kitty" ) );
        farm.push_back( new Cow( "Happy" ) );

        for ( int i=0; i<farm.size(); i++ )
        {
                cout << farm[i]->getName() << " the "
                     << farm[i]->getSpecies() << " says "
                     << farm[i]->speak() << endl;
                delete farm[i];
        }

        return 0;
}
```

```
> ./farm
Annie the Animal says ???
Spot the dog says Woof
Kitty the cat says Meow
Happy the cow says Moo
```

Abstract Classes

- An abstract class has one or more **pure virtual** functions

```
virtual ret_type name(t args) = 0;
```

- Commonly used for classes that support code re-use and polymorphism, but should not be instantiated

Abstracting the Farm

```
class Animal
{
public:
        Animal(string species, string name):
        species( species ), name( name ) {}

        virtual string speak() = 0;
        string getSpecies() { return species; }
        string getName() { return name; }

private:
        string species;
        string name;
};

class Dog: public Animal
{
public:
        Dog(string name): Animal( "dog", name ) {}
        virtual string speak() { return "Woof"; }
};

class Cat: public Animal
{
public:
        Cat(string name): Animal( "cat", name ) {}
        virtual string speak() { return "Meow"; }
};

class Cow: public Animal
{
public:
        Cow(string name): Animal( "cow", name ) {}
        virtual string speak() { return "Moo"; }
};
```

```
int main()
{
        vector< Animal* > farm;
        farm.push_back( new Animal( "Animal", "Annie" ) );
        farm.push_back( new Dog( "Spot" ) );
        farm.push_back( new Cat( "Kitty" ) );
        farm.push_back( new Cow( "Happy" ) );

        for ( int i=0; i<farm.size(); i++ )
        {
                cout << farm[i]->getName() << " the "
                     << farm[i]->getSpecies() << " says "
                     << farm[i]->speak() << endl;
                delete farm[i];
        }

        return 0;
}
```

```
In function 'int main()':
error: cannot allocate an object of abstract type 'Animal'
  farm.push_back( new Animal( "Animal", "Annie" ) );
                                                  ^
note:   because the following virtual functions are pure
within 'Animal':
 class Animal
       ^
note:  virtual std::string Animal::speak()
  virtual string speak() = 0;
```

Proper Farm

```
class Animal
{
public:
        Animal(string species, string name):
        species( species ), name( name ) {}

        virtual string speak() = 0;
        string getSpecies() { return species; }
        string getName() { return name; }

private:
        string species;
        string name;
};

class Dog: public Animal
{
public:
        Dog(string name): Animal( "dog", name ) {}
        virtual string speak() { return "Woof"; }
};

class Cat: public Animal
{
public:
        Cat(string name): Animal( "cat", name ) {}
        virtual string speak() { return "Meow"; }
};

class Cow: public Animal
{
public:
        Cow(string name): Animal( "cow", name ) {}
        virtual string speak() { return "Moo"; }
};
```

```
int main()
{
        vector< Animal* > farm;
        farm.push_back( new Dog( "Spot" ) );
        farm.push_back( new Cat( "Kitty" ) );
        farm.push_back( new Cow( "Happy" ) );

        for ( int i=0; i<farm.size(); i++ )
        {
                cout << farm[i]->getName() << " the "
                     << farm[i]->getSpecies() << " says "
                     << farm[i]->speak() << endl;
                delete farm[i];
        }

        return 0;
}
```

```
> ./farm
Spot the dog says Woof
Kitty the cat says Meow
Happy the cow says Moo
```

Smart Home

```
class Device
{
public:
        Device(string model, string sn): model( model ), sn( sn ) {}
        string getModel() { return model; }
        string getSN() { return sn; }
        string status() { return ( model + " (" + sn + "): " + getStatus() ); }

protected:
        virtual string getStatus() = 0;
        string model;
        string sn;
};

class DoorDevice: public Device
{
public:
        DoorDevice(string model, string sn): Device( model, sn ), open( false ) {}
        void openClose() { open = !open; }

protected:
        virtual string getStatus() { return ((open)?("Open"):("Closed")); };

private:
        bool open;
};

class ThermostatDevice: public Device
{
public:
        ThermostatDevice(string model, string sn, int tempC):
        Device( model, sn ), tempC( tempC ) {}

protected:
        virtual string getStatus() { return ( to_string( tempC ) + "C" ); };

private:
        int tempC;
};
```

```
int main()
{
        Device* devices[3];

        devices[0] = new DoorDevice( "m1", "sn1" );

        devices[1] = new DoorDevice( "m2", "sn2" );
        ((DoorDevice*) devices[1])->openClose();

        devices[2] = new ThermostatDevice( "m3", "sn3", 14 );

        for (  int i=0; i<3; i++ )
        {
                cout << devices[i]->status() << endl;
                delete devices[i];
        }

        return 0;
}
```

```
> ./home
m1 (sn1): Closed
m2 (sn2): Open
m3 (sn3): 14C
```

Exercise

Extend your shapes. Make **`Shape2D`** abstract and **`area`** virtual. Add a **`Rectangle`** class. Populate a **`vector`** of **`Shape2D`** pointers and output their areas.

Answer

```
class Shape2D
{
public:
        Shape2D(string name): name( name ) {}
        string getName() { return name; }
        virtual double area() = 0;

private:
        string name;
};

class Circle: public Shape2D
{
public:
        Circle(double radius):
        Shape2D( "circle" ), radius( radius ) {}

        virtual double area() { return 3.14159*radius*radius; };

private:
        double radius;
};

class Rectangle: public Shape2D
{
public:
        Rectangle(double l, double w):
        Shape2D( "rectangle" ), l( l ), w( w ) {}

        virtual double area() { return l*w; }

private:
        double l, w;
};
```

```
int main()
{
        vector< Shape2D* > shapes;

        shapes.push_back( new Circle( 10.0 ) );
        shapes.push_back( new Rectangle( 3.0, 17.0 ) );

        for ( int i=0; i<shapes.size(); i++ )
        {
                cout << shapes[i]->getName() << ": "
                     << shapes[i]->area() << endl;
                delete shapes[i];
        }

        return 0;
}
```

```
> ./shapes
circle: 314.159
rectangle: 51
```

Notes on Inheritance

- You can assign a derived object to a base object, but only base member variables/functions will be accessible (*slicing*)

```
Base obj = derivedObj; // can't access derived
                       // variables/functions
```

  - Solution: use pointers

- You should always make destructors virtual to clean up after the base and derived classes (similar to constructors)

- In C++, a class can derive from multiple base classes and derived classes can themselves serve as base classes

Wrap Up

- Inheritance is a key OOP method to support code re-use and polymorphism
  - Derived classes inherit variables/functions from a base class
  - Derived classes can redefine functions and add their own member variables/functions
  - The protected access level allows derived classes to access variables/functions that are hidden from external code
- Initialization lists allow for member variable initialization, as well as derived class construction
- Polymorphism allows code to apply to different types that implement a common base class
  - In C++, virtual functions implement late binding
    - Pure virtual functions implement abstract classes