Inheritance and Polymorphism

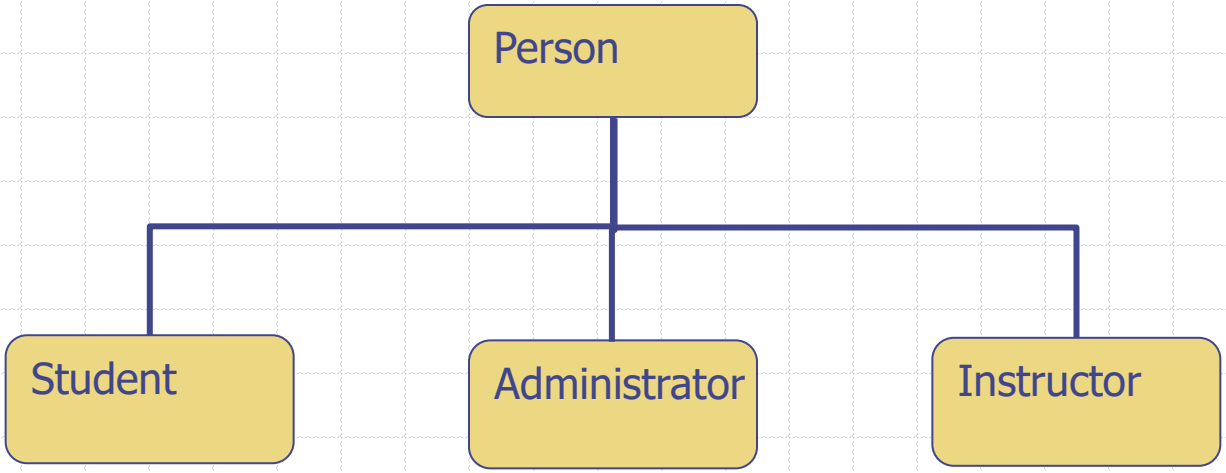
Sections 2.2.1-2.2.4

Inheritance and Polymorphism

- Two features of object-oriented languages which take advantage of hierarchical relationships to help provide code reuse and modularity.
- Inheritance is a mechanism that allows the design of general classes that can be specialized to (perhaps many) particular classes, each of which reuses the code from the general class.
- Polymorphism is a feature that allows a variable to represent different particular classes, provided they all share a common general class or interface.

Inheritance in Object-Oriented Languages

- Suppose we are designing a set of classes to represent people at a university. We'd have a general class Person, and specialized classes Student, Administrator, and Instructor.
- We can represent this with an IS-A hierarchy.



Inheritance in O-O languages

- The general class is known as a base class, a parent class, or a superclass.
- A specialized class is known as a derived class, a child class, or a subclass.
- A subclass is said to specialize or extend its base class, and to inherit the functions of the base class.

Inheritance in C++

```
class Person {
private:
    string      name;
    string      idNum;
public:
    // ...
    void print();
    string getName();
};
```

```
class Student: public Person {
private:
    string      major;
    int         gradYear;
public:
    // ...
    void print();
    void changeMajor(const
            string& newMajor);
};
```

Member Functions

```
Person mary("Mary", "12-345");
Student bob("Bob", "98-764", "Math", 2012);

cout << bob.getName() << endl;          // Person::getName()
mary.print();                           // Person::print()
bob.print();                            // Student::print()
mary.changeMajor("Physics");            // Error
bob.changeMajor("English");             // Student::changeMajor()
```

- :: is called the class scope operator in C++.

Using the class scope operator

```
void Person::print() {
    cout << "Name " << name << endl;
    cout << "IDnum " << idNum << endl;
}

void Student::print() {
    Person::print();
    cout << "Major " << major << endl;
    cout << "Year " << gradYear << endl;
}
```

Protected members

- A subclass does not inherit private members (data or functions) from its superclass.
- A subclass inherits public members, but every class can see or use such members.
- An inbetween option is to use protected members, which the subclasses inherit but other classes cannot see or use.
- protected is used just like private or public.

```
Class Something {
private:
   int a;
protected:
   int b;
public:
   int c;
}
```

Constructors

- When a derived class is constructed, it is the responsibility of this class's constructor to call the appropriate constructor for its base class.

```
Person::Person(const string& nm, const string& id)
    : name(nm),
      idNum(id) { }

Student::Student(const string& nm, const string& id,
                                   const string& maj, int year)
    : Person(nm, id),
      major(maj),
      gradYear(year) { }
```

Constructors

Alternatively:

```
Person::Person(const string& nm, const string& id) {
      name = nm;
      idNum = id;
}

Student::Student(const string& nm, const string& id,
                                   const string& maj, int year)
    : Person(nm, id) {
      major = maj;
      gradYear = year;
}
```

Destructors

- Classes are destroyed in reverse order from their construction—subclasses before superclasses.
- Subclass destructors do not need to call superclass destructors; it is done automatically

```
Person::~Person() { }
Student::~Student() { }

Student* s = new Student("Carol", "34-927", "Physics", 2014);
delete s;                          // calls ~Student() then ~Person()
```

Static Binding

```
Person *pp[100];
pp[0] = new Person(…);
pp[1] = new Student(…);

cout << pp[1]->getName() << '\n';
pp[0]->print();              // calls Person::print()
pp[1]->print();              // calls Person::print()
pp[1]->changeMajor("English");          // Error
```

- C++ by default uses static binding—when determining which member function to call, it considers the object's declared type, not its actual type.

Dynamic Binding

- In computing science, static ("not moving") means at compile time. Dynamic ("moving") means at run time.
- So static binding means that the binding (determination of which member function to call) happens at compile time.
- In contrast, dynamic binding determines which function to call at run time.
- We can force C++ to do dynamic binding by adding the keyword **virtual** to a function's declaration.

Dynamic Binding

```
class Person {
    virtual void print() { … }
    // …
}
```

```
class Student: public Person {
    virtual void print() { … }
    // …
}
```

```
Person *pp[100];
pp[0] = new Person(…);
pp[1] = new Student(…);
pp[0]->print();          // calls Person::print()
pp[1]->print();          // calls Student::print()
```

Virtual Destructors

- There are no virtual constructors; the concept makes no sense.
- When we delete an element of our array pp[], we may need to delete a Student and may need to delete a Person.
- Therefore we need to call a destructor based on the actual run-time type of the element.
- This is done by declaring a virtual destructor, e.g.:

  virtual ~Person();

  for the Person class, and similar for the Student class.

Virtual Destructors

- Important rule:

> If a base class defines any virtual functions, it should define a virtual destructor, even if that destructor is empty.

Polymorphism

- Literally, polymorphism means "many forms".
- For computing science, it means the ability of a variable or a function to take different types.
- The array variable pp[] in our previous example is a polymorphic variable.
- A variable p declared as a pointer to some class S implies that the variable p can point to any object belonging to any subclass T of S.
- If T and S both define a virtual member function a, which is called when we invoke p->a?

Polymorphism

- If T and S both define a virtual member function a, which is called when we invoke p->a?
  - If p points to an object of class T, then it calls T::a. In this case, T is said to override the function S::a.
  - If p points to an object of class S, then it calls S::a.
- If p points to a class object with at least one virtual function, p is called polymorphic.
- Inheritance, polymorphism, and function overriding support reusable software.

Specialization and Extension

- The two primary ways of using inheritance are for specialization and extension.
- In specialization, a subclass inherits some functions of the superclass but overrides others. The overrides provide a special way the subclass does the general function.
- In extension, a subclass inherits the functions of the superclass and adds other functions. These added functions extend the capabilities of the superclass.

Example of Specialization

```
class Shape {
  public:
    virtual void draw();
  // ...
}

class BitMap: public Shape {
  public:
    virtual void draw();
  // ...
}
```

```
class Circle: public Shape {
  public:
    virtual void draw();
  // ...
}

Shape* shapes[10];
// ... initialize shapes ...

for(int i=0; i<10; i++) {
    shapes[i]->draw();
}
```

Example of Extension

```
class Dog {
  public:
    void bark();
    double getWeight();
  // ...
}

class BorderCollie: public Dog {
  public:
    void herd();
  // ...
}
```

```
BorderCollie* lassie =
    new BorderCollie(...);

lassie->bark();
cout << lassie->getWeight()
        << endl;
lassie->herd();
```