# Programming Abstractions

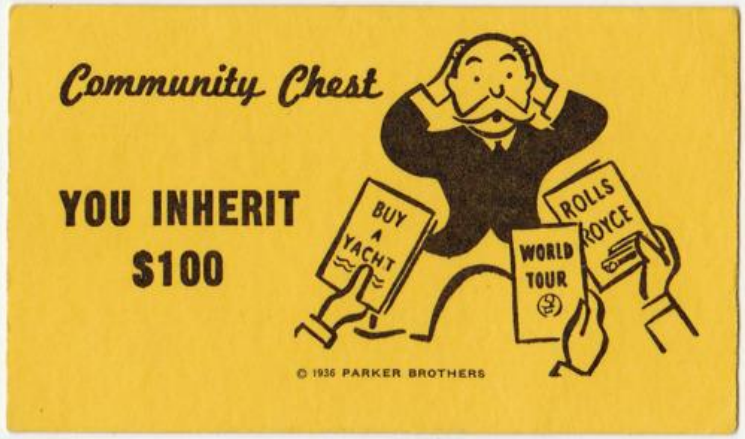
CS106B

Cynthia Lee

# Inheritance Topics

**Inheritance**

- Continue our lawyer example from Wednesday
- New: Polymorphism and the “virtual” keyword



# Polymorphism

Start with *how*

# Polymorphism

**polymorphism**: Ability for the **same code** to be used with **different types** of objects and **behave differently with each**.

- Templates provide a kind of *compile-time* polymorphism.
  - `Grid<int>` or `Grid<string>` will output different things for `myGrid[0][0]`, but we can predict at compile time which it will do

- Inheritance provides *run-time* polymorphism.
  - `someEmployee->salary()` will behave differently at runtime depending on what type of employee—may not be able to predict at compile time which it is

# Polymorphism

**Fun fact!** A pointer of type *T* can point to any subclass of *T*.

```
Employee   *neha    = new Employee("Neha", 3);
 Employee   *pedro   = new Programmer("Pedro", 3);
Employee   *diane   = new Lawyer("Diane", 3, "Stanford");
Programmer *cynthia = new Programmer("Cynthia", 3);
```

- Why would you do this?
  - Handy if you want to have a function that works on any Employee, but takes advantage of custom behavior by specific employee type:

```
Vector<Employee*> staff;
staff.add(neha);
staff.add(pedro);
staff.add(diane);
staff.add(cynthia);
for (int i = 0; i < staff.size(); i++) {
    Employee *person = staff[i];
    cout << "Congratulations, " << person->name() << "! You are now $"
         << person->salary()/12 << " wealthier!" << endl;
}
```

# Polymorphism

A pointer of type *T* can point to any subclass of *T*.

```
Employee   *neha    = new Employee("Neha", 3);
Employee   *pedro   = new Programmer("Pedro", 3);
Employee   *diane   = new Lawyer("Diane", 3, "Stanford");
Programmer *cynthia = new Programmer("Cynthia", 3);
```

- When a member function is called on `diane`, it behaves as a `Lawyer`.
  - `diane->salary();`
  - (This is because all the employee functions are declared `virtual`.)
- You can *not* call any `Lawyer`-only members on `diane` (e.g. `sue`).
  - `diane->sue();` `// will NOT compile!`
- You *can* call any `Programmer`-only members on `cynthia` (e.g. `code`).
  - `cynthia->code("Java");` `// ok!`

# Polymorphism examples

You can use the object's extra functionality by casting.

```
Employee   *diane   = new Lawyer("Diane", 5, "Stanford");
diane->vacationDays();                          // ok
diane->sue("Cynthia");                          // compiler error
((Lawyer*) diane)->sue("Cynthia");              // ok
```

**Pro Tip: you should not cast a pointer into something that it is not!**

- It will compile, but the code will crash (or behave unpredictably) when you try to run it.

```
Employee   *carlos  = new Programmer("Carlos", 3);
carlos->code();                                 // compiler error
((Programmer*) carlos)->code("C++");            // ok
((Lawyer*) carlos)->sue("Cynthia");             // No!!! Compiles but crash!!
```

# Rules for “virtual”: runtime calls

**DerivedType* obj = new DerivedType();  // Lawyer* bob = new Lawyer();**

If we call a method like this: obj->method(), only one thing could happen:

1. **DerivedType**’s implementation of method is called (or most specific implementation available in **DerivedType** doesn’t have one)

**BaseType* obj = new DerivedType(); // Employee* bob = new Lawyer();**

If we call a method like this: obj->method(), two different things could happen:

1. If method is **not virtual**, then **BaseType**’s implementation of method is called
2. If method is **virtual**, then **DerivedType**’s implementation of method is called (or most specific implementation available in **DerivedType** doesn’t have one)

# Rules for “virtual”: pure virtual

If a method of a class looks like this, then this method is a called “**pure virtual**” function:

```
virtual returntype method() = 0; // does this instead of defining
```

- It means that the class is sort of declining to provide a definition of that function, but expecting that there will be derived classes and that they will implement it.
- Such a class is called an “**abstract class**” meaning that you can’t create objects of that class. It can only serve as a base class for other classes that do provide a definition of the function.

```
class Mammal {
public:
  virtual void makeSound() = 0;
  string toString() { return "Mammal"; }
};
class Cat : public Mammal {
public:
  virtual void makeSound() { cout << "rawr" << endl; }
  string toString() { return "Cat"; }
};
class Siamese : public Cat {
public:
  virtual void makeSound() { cout << "meow" << endl; }
  string toString() { return "Siamese"; }
  virtual void scratchCouch() { cout << "scraaaatch" << endl; }
};
```

**What is printed?**

```
Siamese * s = new Mammal;
cout << s->toString();
```

(A) "Mammal"
(B) "Cat"
(C) "Siamese"
(D) Gives an error (identify compiler or crash)
(E) Other/none/more

```
class Mammal {
public:
  virtual void makeSound() = 0;
  string toString() { return "Mammal"; }
};
class Cat : public Mammal {
public:
  virtual void makeSound() { cout << "rawr" << endl; }
  string toString() { return "Cat"; }
};
class Siamese : public Cat {
public:
  virtual void makeSound() { cout << "meow" << endl; }
  string toString() { return "Siamese"; }
  virtual void scratchCouch() { cout << "scraaaatch" << endl; }
};
```

**What is printed?**

```
Siamese * s = new Siamese;
cout << s->toString();
```

(A)"Mammal"
(B)"Cat"
(C)"Siamese"
(D) Gives an error (identify compiler or crash)
(E) Other/none/more

```
class Mammal {
public:
  virtual void makeSound() = 0;
  string toString() { return "Mammal"; }
};
class Cat : public Mammal {
public:
  virtual void makeSound() { cout << "rawr" << endl; }
  string toString() { return "Cat"; }
};
class Siamese : public Cat {
public:
  virtual void makeSound() { cout << "meow" << endl; }
  string toString() { return "Siamese"; }
  virtual void scratchCouch() { cout << "scraaaatch" << endl; }
};
```

**What is printed?**

```
Mammal * m = new Mammal;
cout << m->toString();
```

(A)"Mammal"
(B)"Cat"
(C)"Siamese"
(D) Gives an error (identify compiler or crash)
(E) Other/none/more

```
class Mammal {
public:
  virtual void makeSound() = 0;
  string toString() { return "Mammal"; }
};
class Cat : public Mammal {
public:
  virtual void makeSound() { cout << "rawr" << endl; }
  string toString() { return "Cat"; }
};
class Siamese : public Cat {
public:
  virtual void makeSound() { cout << "meow" << endl; }
  string toString() { return "Siamese"; }
  virtual void scratchCouch() { cout << "scraaaatch" << endl; }
};
```

**What is printed?**

```
Mammal * m = new Siamese;
cout << m->toString();
```

(A) "Mammal"
(B) "Cat"
(C) "Siamese"
(D) Gives an error (identify compiler or crash)
(E) Other/none/more

```
class Mammal {
public:
  virtual void makeSound() = 0;
  string toString() { return "Mammal"; }
};
class Cat : public Mammal {
public:
  virtual void makeSound() { cout << "rawr" << endl; }
  string toString() { return "Cat"; }
};
class Siamese : public Cat {
public:
  virtual void makeSound() { cout << "meow" << endl; }
  string toString() { return "Siamese"; }
  virtual void scratchCouch() { cout << "scraaaatch" << endl; }
};
```

**What is printed?**

```
Mammal * m = new Siamese;
m->scratchCouch();
```

(A)"Mammal"
(B)"Cat"
(C)"Siamese"
(D) Gives an error (identify compiler or crash)
(E) Other/none/more

```
class Mammal {
public:
  virtual void makeSound() = 0;
  string toString() { return "Mammal"; }
};
class Cat : public Mammal {
public:
  virtual void makeSound() { cout << "rawr" << endl; }
  string toString() { return "Cat"; }
};
class Siamese : public Cat {
public:
  virtual void makeSound() { cout << "meow" << endl; }
  string toString() { return "Siamese"; }
  virtual void scratchCouch() { cout << "scraaaatch" << endl; }
};
```

**What is printed?**

```
Cat * c = new Siamese;
c->makeSound();
```

(A) "rawr"
(B) "meow"
(C) "Siamese"
(D) Gives an error (identify compiler or crash)
(E) Other/none/more