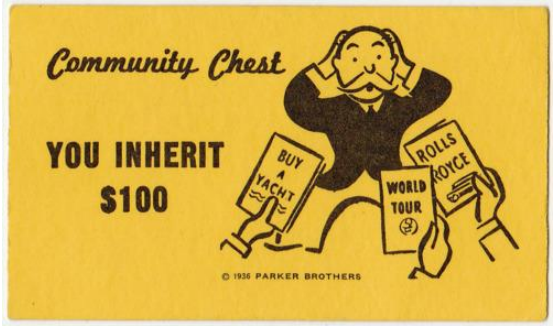
Programming Abstractions

CS106X

Cynthia Lee

Inheritance Topics



Inheritance

- The basics
  - › Example: Stanford GObject class
- **Polymorphism**
  - › Example: Expression trees (final project)

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

We will keep working with the Employee class:

- **Employees** have a name, years worked, salary, vacation, …
- **Lawyers** know how to sue and get paid 2x as much
- **Programmers** know how to write code and get bigger raises each year

- (Code is now on lectures page of website.)

Polymorphism

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

```
Employee   *neha    = new Programmer("Neha", 2);
Employee   *diane   = new Lawyer("Diane", "Stanford", 5);
Programmer *cynthia = new Programmer("Cynthia", 10);
```

- 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:

```
void doMonthlyPaycheck(Employee *employee) {
    cout << "You are now $" << employee->salary()/12 << " wealthier!" << endl;
}
```

Polymorphism

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

```
Employee   *neha    = new Programmer("Neha", 2);
Employee   *diane   = new Lawyer("Diane", "Stanford", 5);
Programmer *cynthia = new Programmer("Cynthia", 10);
```

- 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", "Stanford", 5);
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();

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

1. DerivedType’s implementation of method is called

BaseType * obj = new DerivedType();

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

Rules for “virtual”: pure virtual

If a method of a class looks like this:

- `virtual returntype method() = 0;`
- then this method is a called “**pure virtual**” function
- and the class is called an “**abstract class**”
- Abstract classes are like Java interfaces
- You cannot do “= new Foo();” if Foo is abstract (just like Java interfaces)
- ALSO, you cannot do “= new DerivedFoo();” if DerivedFoo extends Foo and DerivedFoo does not implement *all* the pure virtual methods of Foo

```
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; }
};
```

abstract

pure virtual

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; }
};
```

abstract

pure virtual

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();
```

((Siamese*))

(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

Stanford 1-2-3 Walkthrough

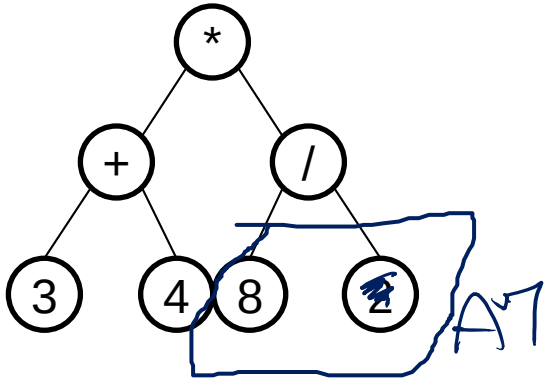
The `Expression` class

Walkthrough of Stanford 1-2-3

- Excel-like spreadsheet
- Among other things, it needs to parse expressions

  ```
  › = (3 + 4) * A7
  ```

- We saw something similar to this earlier in the quarter:



Evaluation of **CompoundExp**

```
CompoundExp::~CompoundExp() {
   delete lhs;
   delete rhs;
}

double CompoundExp::eval(EvaluationContext & context) const {
   double right = rhs->eval(context);
   double left = lhs->eval(context);
   if (op == "+") return left + right;
   if (op == "-") return left - right;
   if (op == "*") return left * right;
   if (op == "/") return left / right; // divide by 0.0 gives ±INF

   error("Illegal operator in expression.");
   return 0.0;
}
```

This is in the implementation of **CompoundExp**—let's take a look at the **.h** file to see what **op**, **lhs**, and **rhs** are

```
class CompoundExp : public Expression {

public:

/** ...*/

   CompoundExp(const std::string& op, const Expression *lhs, const Expression *rhs);

/* Prototypes for the virtual methods overridden by this class */

   virtual ~CompoundExp();
   virtual double eval(EvaluationContext& context) const;
   virtual std::string toString() const;
   virtual ExpressionType getType() const;

/* Prototypes of methods specific to this class */
   std::string getOperator() const;
   const Expression *getLHS() const;
   const Expression *getRHS() const;

private:
   std::string op;              /* The operator string (+, -, *, /)  */
   const Expression *lhs, *rhs; /* The left and right subexpression  */
};
```

`Expression` (base class)

```
class Expression {

public:

   Expression();
   virtual ~Expression();

   virtual double eval(EvaluationContext& context) const = 0;
   virtual std::string toString() const = 0;
   virtual ExpressionType getType() const = 0;
};
```

Note: you cannot actually create an `Expression` object
These methods are never implemented (note the "`= 0`")

- "pure virtual"

`Expression` exists solely to provide a base class to others

- "abstract"

Another Derived class: **DoubleExp**

```
class DoubleExp : public Expression {

public:

   DoubleExp(double value);

/* Prototypes for the virtual methods overridden by this class */
   double eval(EvaluationContext& context) const;
   std::string toString() const;
   ExpressionType getType() const;

/* Prototypes of methods specific to this class */
   double getDoubleValue() const;

private:
   double value;                    /* The value of the constant */
};
```

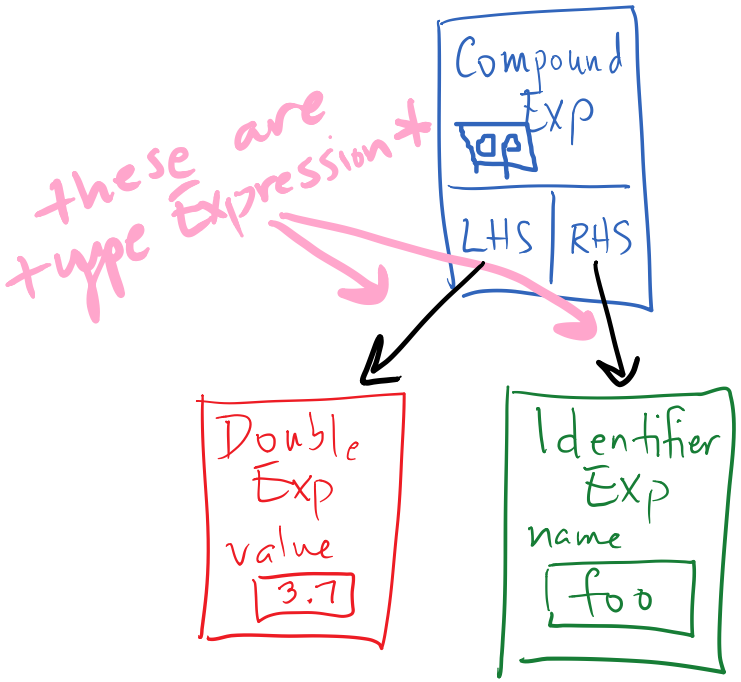
Another Derived class: **IdentifierExp**

```
class IdentifierExp : public Expression {

public:

   IdentifierExp(const std::string& name);

/* Prototypes for the virtual methods overridden by this class */
   double eval(EvaluationContext& context) const;
   std::string toString() const;
   ExpressionType getType() const;

/* Prototypes of methods specific to this class */
   std::string getIdentifierName() const;

private:
   std::string name;              /* The name of the identifier */
};
```

Stanford 1-2-3 `eval`

Because of the "is a" relationship: `lhs/rhs of compoundExp can be any of:`

- `CompoundExp`
- `DoubleExp`
- `IdentifierExp`



Stanford 1-2-3 **`eval`**

`CompoundExp` doesn't want to care exactly what type its **`lhs`** and **`rhs`** are

Just calls **`eval()`** on whatever they are and gets the right value!