SRS Setup
Login: student.turningtechnologies.com
Session ID: 20220404<A|D>

Replace <A|D> with this section's letter

Inheritance Practice

—

CS 2124: Object Oriented Programming
Darryl Reeves, Ph.D.

Agenda

- Inheritance review
- In-class problems
- The `final` keyword

Inheritance review

Inheriting methods

```
class Animal {};

class Lion : public Animal {};

class Tiger : public Animal {};

class Bear : public Animal {};
```

Inheriting methods

```
class Animal {
public:
    void eat() { cout << "Animal eating\n"; }
};

class Lion : public Animal {};

class Tiger : public Animal {};

class Bear : public Animal {};

int main() {
    Bear yogi;
    yogi.eat();
}
```

inherited from base class

no eat() method defined

```
Animal eating
```

Redefining methods

```
class Animal {
public:
    void eat() { cout << "Animal eating\n"; }
};
class Lion : public Animal {};
class Tiger : public Animal {
public:
    void eat() { cout << "Tiger eating\n"; }
};
class Bear : public Animal {};

int main() {
    Bear yogi;
    yogi.eat();

    Tiger tigger;
    tigger.eat();
}
```

```
Animal eating
Tiger eating
```

Inheritance assignment rules

- derived class instance to base class instance ✔
- base class instance to derived class instance ✘
- address of derived class instance to base class pointer ✔
- address of base class instance to derived class pointer ✘

Inheritance assignment rules

- derived class instance to base class instance ✔
- base class instance to derived class instance ✘
- address of derived class instance to base class pointer ✔
- address of base class instance to derived class pointer ✘

All are compile time considerations

Dynamic binding using the `virtual` keyword

```
class Animal {
public:
    void eat() { cout << "Animal eating\n"; }
};
class Lion : public Animal {
public:
    void eat() { cout << "Lion eating\n"; }
};
class Tiger : public Animal {
public:
    void eat() { cout << "Tiger eating\n"; }
};
class Bear : public Animal {
public:
    void eat() { cout << "Bear eating\n"; }
};
```

```
int main() {
    Lion fred;
    Tiger tigger;
    Bear pooh;

    vector<Animal*> animals;

    animals.push_back(&fred);
    animals.push_back(&tigger);
    animals.push_back(&pooh);

    for (size_t i = 0; i < animals.size(); ++i) {
        animals[i]->eat();
    }
}
```

observed

```
Animal eating
Animal eating
Animal eating
```

wanted

```
Lion eating
Tiger  eating
Bear  eating
```

Dynamic binding using the `virtual` keyword

method can be redefined in subclass and bound to object at **runtime**

```
class Animal {
public:
    virtual void eat() {
        cout << "Animal eating\n";
    }
};
class Lion : public Animal {
public:
    void eat() { cout << "Lion eating\n"; }
};
class Tiger : public Animal {
public:
    void eat() { cout << "Tiger eating\n"; }
};
class Bear : public Animal {
public:
    void eat() { cout << "Bear eating\n"; }
};
```

```
int main() {
    Lion fred;
    Tiger tigger;
    Bear pooh;

    vector<Animal*> animals;

    animals.push_back(&fred);
    animals.push_back(&tigger);
    animals.push_back(&pooh);

    for (size_t i = 0; i < animals.size(); ++i) {
        animals[i]->eat();
    }
}
```

wanted

```
Lion eating
Tiger  eating
Bear  eating
```

observed

Overriding a function properly

```
class Base {
public:
    virtual void where_am_i() const { cout << "Base\n"; }
};

class Derived : public Base {
public:
    void where_am_i() override { cout << "Derived\n"; }
};

void foo(Base& thing) {
    thing.where_am_i();
}

int main() {
    Base base;
    foo(base);
    Derived der;
    foo(der);
}
```

compilation error!

Polymorphism and function parameters

```
class Animal {
public:
    virtual void eat() {
        cout << "Animal eating\n";
    }
};

class Lion : public Animal {
public:
    void eat() { cout << "Lion eating\n"; }
};

class Bear : public Animal {
public:
    void eat() { cout << "Bear eating\n"; }
};
```

pass-by-value results in slicing

```
void feed_animal(Animal an) {
    cout << "Feeding the animal\n";
    an.eat();
}
```

```
int main() {
    Lion leo;

    feed_animal(leo);
}
```

```
Feeding the animal
Animal eating
```

Polymorphism and function parameters

```
class Animal {
public:
    virtual void eat() {
        cout << "Animal eating\n";
    }
};

class Lion : public Animal {
public:
    void eat() { cout << "Lion eating\n"; }
};

class Bear : public Animal {
public:
    void eat() { cout << "Bear eating\n"; }
};
```

pass-by-reference

```
void feed_animal(Animal& an) {
    cout << "Feeding the animal\n";
    an.eat();
}
```

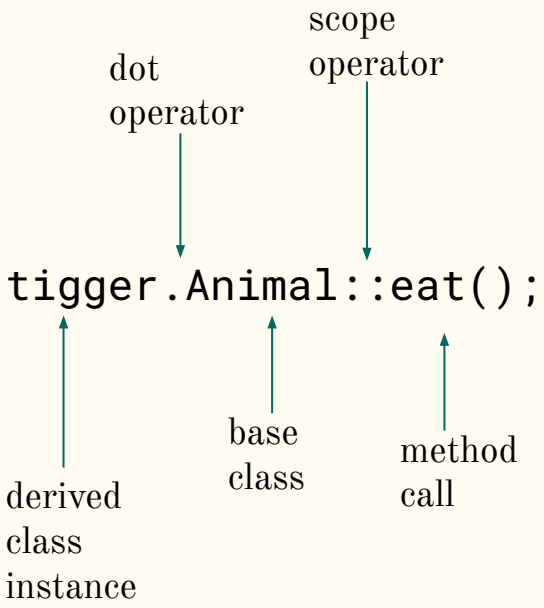
```
int main() {
    Lion leo;

    feed_animal(leo);
}
```

```
Feeding the animal
Lion eating
```

Calling a base class method (outside of class)

```
class Animal {
public:
    virtual void eat() {
        cout << "Animal eating\n";
    }
};

class Tiger : public Animal {
public:
    void eat() { cout << "Tiger eating\n"; }
};
```

```
int main() {
    Tiger tigger;
    tigger.eat();
    tigger.Animal::eat();
}
```

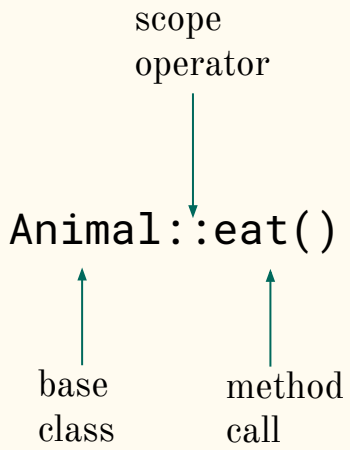



```
Tiger  eating
Animal eating
```

Calling a base class method (inside of class)

```
class Animal {
public:
    virtual void eat() {
        cout << "Animal eating\n";
    }
};

class Tiger : public Animal {
public:
    void eat() {
        cout << "Tiger!!!\n";
        Animal::eat();
    }
};
```

```
int main() {
    Tiger tigger;

    tigger.eat();
}
```




```
Tiger!!!
Animal eating
```

Method hiding

```
class Base {
public:
    void foo(int num) const { cout << "Base::foo(num)\n"; }
};

class Derived : public Base {
public:
    void foo() const { cout << "Derived::foo()\n"; }
};

int main() {
    Derived der;
    der.foo(17);  compilation error!
}
```

Method hiding

```
class Base {
public:
    void foo(int num) const { cout << "Base::foo(num)\n"; }
};

class Derived : public Base {
public:
    void foo() const { cout << "Derived::foo()\n"; }
    void foo(int num) const { Base::foo(num); }
};

int main() {
    Derived der;
    der.foo(17);
}
```

~~compilation error!~~

```
% g++ -std=c++11 hiding4.cpp -o hiding4.o
% ./hiding4.o
Base::foo(num)
```

Method hiding

```
class Base {
public:
    void foo(int num) const { cout << "Base::foo(num)\n"; }
};

class Derived : public Base {
public:
    void foo() const { cout << "Derived::foo()\n"; }
   void foo(int num) const { Base::foo(num); }
    using Base::foo;
};

int main() {
    Derived der;
    der.foo(17);
}
```

```
% g++ -std=c++11 hiding4.cpp -o hiding4.o
% ./hiding4.o
Base::foo(num)
```

Polymorphism with non-members

```
class Base {
public:
    virtual void display(ostream& os) const { os << "Base"; }
};

ostream& operator<<(ostream& os, const Base& rhs) {
    rhs.display(os);
    return os;
}

class Derived : public Base {
public:
    virtual void display(ostream& os) const { os << "Derived"; }
};

void func(const Base& base) {
    cout << base << endl;
}

int main() {
    Derived der;
    cout << der << endl;
    func(der);
}
```

observed

```
Derived
Derived
```

wanted

Inheriting member variables

```
class Animal {
public:
    Animal(const string& name) : name(name) {}
    void eat() { cout << "Animal eating\n"; }
private:
    string name;
};

class Lion : public Animal {};

class Tiger : public Animal {
public:
   void eat() { cout << "Tiger eating\n"; }
};

class Bear : public Animal {};
```

```
int main() {
    Tiger tigger("Tigger");
    tigger.eat();
}
```

compilation error!

constructors not inherited

Inheriting member variables

```
class Animal {
public:
    Animal(const string& name) : name(name) {}
    void eat() { cout << "Animal eating\n"; }
private:
    string name;
};

class Lion : public Animal {};

class Tiger : public Animal {
public:
   Tiger(const string& name) : name(name) {}
    void eat() { cout << "Tiger eating\n"; }
};

class Bear : public Animal {};
```

name is private to Animal class

compilation error!

```
int main() {
    Tiger tigger("Tigger");
    tigger.eat();
}
```

compilation error!

Inheriting member variables

```
class Animal {
public:
    Animal(const string& name) : name(name) {}
    void eat() { cout << "Animal eating\n"; }
private:
    string name;
};

class Lion : public Animal {};

class Tiger : public Animal {
public:
   Tiger(const string& name) : Animal(name) {}
   void eat() { cout << "Tiger eating\n"; }
};

class Bear : public Animal {};
```

compilation error!

```
int main() {
    Tiger tigger("Tigger");
    tigger.eat();
}
```

compilation error!

Inheritance and constructors

- derived constructor always invokes a base class constructor
- derived constructor initialization list
  - base class constructor ✔
  - member variables declared in derived class ✔
  - base class member variables ✘
- programmer can specify which base class constructor to use
  - must already exist

Polymorphism in constructors

Simple: polymorphism turned off inside of constructors

```
class Base {
public:
    Base() { foo(); }
    virtual void foo() const { cout << "Base\n"; }
    void display() { this->foo(); }
};

class Derived : public Base {
public:
    Derived(int val) : Base(), x_mem(val) {}
    void foo() const { cout << "Derived: x_mem == " << x_mem << endl; }
private:
    int x_mem;
};

int main() {
    Derived der(17);

}
```

Base

Polymorphism in constructors

calls class implementation (`virtual` or not)

```
class Base {
public:
    Base() { foo(); }
    virtual void foo() const { cout << "Base\n"; }
    void display() { this->foo(); }
};

class Derived : public Base {
public:
    Derived(int val) : Base(), x_mem(val) {}
    void foo() const { cout << "Derived: x_mem == " << x_mem << endl; }
private:
    int x_mem;
};

int main() {
    Derived der(17);
    der.display();
}
```

normal polymorphism rules apply

```
Base
Derived: x_mem == 17
```

`protected` mode

```
class Base {
    friend ostream& operator<<(ostream& os, const Base& base) {
        return os <<  "x: " << base.x_mem;
    }
public:
    Base(int x_val) : x_mem(x_val) {}
protected:
    // define a mutator method for modifying x_mem
private:
    int x_mem;
};

class Derived : public Base {
public:
    Derived(int x_val) : Base(x_val) {}
    void derived_setting_x() {
        x_mem = 42;
    }
};

int main() {
    Derived der(7);
    cout << der << endl;
    der.derived_setting_x();
    cout << der << endl;
}
```

private even for derived classes

compilation error!

class members defined as `protected` can be modified from outside of the class by derived classes

protected mode

```
class Pet {
public:
    Pet(const string& name) : name(name) {}
protected:
    string get_name() const { return name; }
private:
    string name;
};

class Dog : public  Pet {
public:
    Dog(const string& name) : Pet(name) {}
};

class Cat : public Pet {
public:
    Cat(const string& name) : Pet(name) {}
    void display() const { cout << get_name() << endl; }
    void display_dog(const Dog& a_dog) const {
        cout << a_dog.get_name() << endl;
    }
};
```

get_name() method only accessible for current object

```
int main() {
    Cat felix("Felix");
    felix.display();

    Dog fido("Fido");
    felix.display_dog(fido);
}
```

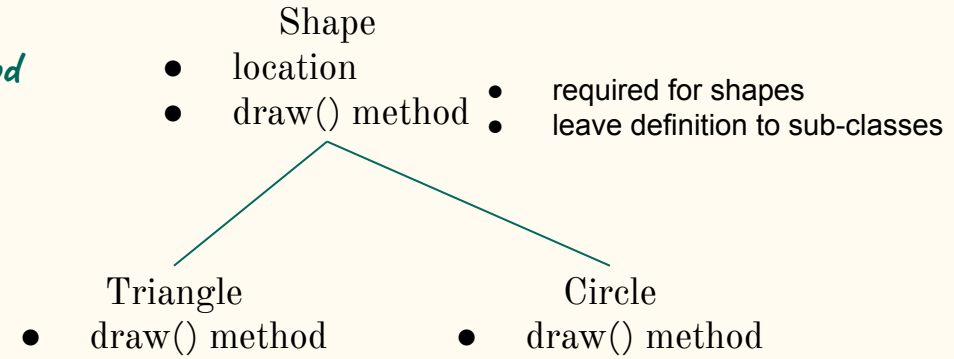
Implementing an interface

```
class Shape {
public:
    Shape(int x, int y) : x(x), y(y) {}
    virtual void draw() = 0;
private:
    int x, y;
};

class Triangle : public Shape {
public:
    Triangle(int x, int y) : Shape(x,y) {}
    void draw() {
        /* stuff to draw triangle */
        cout << "Drawing a triangle\n";
    }
};

class Circle : public Shape {
public:
    Circle(int x, int y) : Shape(x,y) {}
    void draw() {
        /* stuff to draw a circle */
        cout << "Drawing a circle\n";
    }
};
```

abstract class

abstract/pure virtual method

prevents class from being instantiated



Implementing an interface

```
class Shape {
public:
    Shape(int x, int y) : x(x), y(y) {}
    virtual void draw() = 0;
private:
    int x, y;
};

void Shape::draw() { cout << "Default stuff... "; }

class Triangle : public Shape {
public:
    Triangle(int x, int y) : Shape(x,y) {}
    void draw() {
        Shape::draw();
        /* stuff to draw triangle */
        cout << "Drawing a triangle\n";
    }
};

class Circle : public Shape {
public:
    Circle(int x, int y) : Shape(x,y) {}
    void draw() {
        Shape::draw();
        /* stuff to draw a circle */
        cout << "Drawing a circle\n";
    }
};
```

abstract class

code reuse

```
int main() {
    Triangle tri(3,4);
    tri.draw();

    Circle circ(10,10);
    circ.draw();

    Shape a_shape(5,4);
}
```

compilation error!

what if some common behavior exists for drawing shapes?

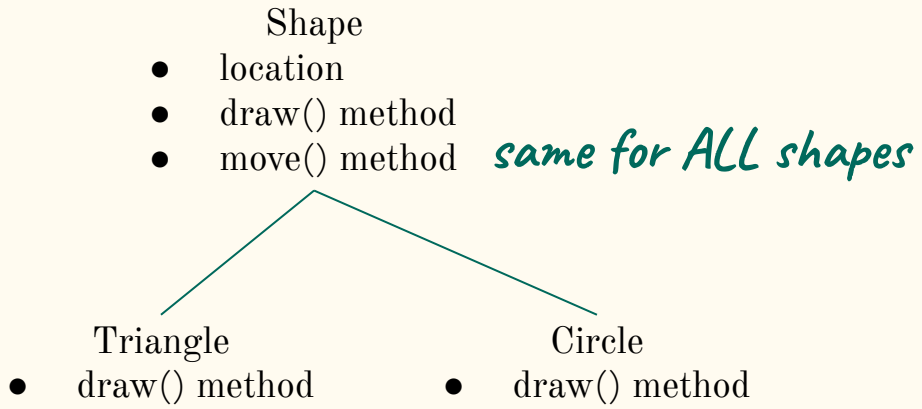
Implementing an interface

```
class Shape {
public:
    Shape(int x, int y) : x(x), y(y) {}
    virtual void draw() = 0;
    void move(int x, int y) {
        this->x = x;
        this->y = y;
    }
private:
    int x, y;
};
void Shape::draw() { cout << "Default stuff... "; }

class Triangle : public Shape {
public:
    Triangle(int x, int y) : Shape(x,y) {}
    void draw() {
        Shape::draw();
        /* stuff to draw triangle */
        cout << "Drawing a triangle\n";
    }
};

class Circle : public Shape {
public:
    Circle(int x, int y) : Shape(x,y) {}
    void draw() {
        Shape::draw();
        /* stuff to draw a circle */
        cout << "Drawing a circle\n";
    }
};
```

abstract class

only 1 method needs to be pure virtual/abstract

```
int main() {
    Triangle tri(3,4);
    tri.draw();

    Circle circ(10,10);
    circ.draw();
}
```



Overriding vs overloading

```
class Parent {
public:
    virtual void whereami() const {
        cout << "Parent" << endl;
    }
};
class Child : public Parent {
public:
    void whereami() const {
        cout << "Child!!!" << endl;
    }
};
class Grandchild : public Child {
public:
    void whereami() const {
        cout << "Grandchild!!!" << endl;
    }
};
void func(const Parent& base) { cout << "func(Parent)\n"; }
void func(const Child& derived) { cout << "func(Child)\n"; }

void other_func(const Parent& base) {
    func(base);
    base.whereami();
}
```

overriding --
choice made at runtime

overloading --
choice made at compile-time

In-class problem I

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat {

};
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat ___ ___ ___ {

};
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat ___ ___ _1_ {

};
```

TurningPoint

SRS Setup
Login: student.turningtechnologies.com
Session ID: 20220404<A|D>

Replace <A|D> with this section's letter

Which base class will the Cat class be derived from (replacing blank #1)?

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat ___ ___ _1_ {

};
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat ___ ___ Pet {

};
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat ___ _2_ Pet {

};
```

Which inheritance type replaces blank #2 to ensure that all public members of the `Pet` class remain public in the `Cat` class?

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat ___ _2_ Pet {

};
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat ___ public Pet {

};
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat _3_ public Pet {

};
```

Which symbol replaces blank #3 to separate the declaration of the `Cat` class from its parent class?

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat _3_ public Pet {

};
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
};
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat eating";
    }
};
```

works but not reusing code

```
int main() {
    Cat felix;
    felix.eat();
}
```

```
% g++ -std=c++11 inheritance1.cpp -o inheritance1.o
% inheritance1.o
Cat eating
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        ---
    }
};
```

```
int main() {
    Cat felix;
    felix.eat();
}
```

```
% g++ -std=c++11 inheritance1.cpp -o inheritance1.o
% inheritance1.o
Cat eating
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        ---
    }
};
```

```
int main() {
    Cat felix;
    felix.eat();
}
```

```
% g++ -std=c++11 inheritance1.cpp -o inheritance1.o
% inheritance1.o
Cat eating
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        _4_
    }
};
```

```
int main() {
    Cat felix;
    felix.eat();
}
```

```
% g++ -std=c++11 inheritance1.cpp -o inheritance1.o
% inheritance1.o
Cat eating
```

Which function call (replacing blank #4) will allow for reusing the code from the `eat()` method defined in the `Pet` class?

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        _4_
    }
};
```

```
int main() {
    Cat felix;
    felix.eat();
}
```

```
% g++ -std=c++11 inheritance1.cpp -o inheritance1.o
% inheritance1.o
Cat eating
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
```

code reuse ✔

```
int main() {
    Cat felix;
    felix.eat();
}
```

```
% g++ -std=c++11 inheritance1.cpp -o inheritance1.o
% inheritance1.o
Cat eating
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
```

```
int main() {
    Cat felix;
    felix.eat();
}
```

```
% g++ -std=c++11 inheritance1.cpp -o inheritance1.o
% inheritance1.o
Cat eating
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};

void poly(___ a_pet) {
     cout << "passed in a pet" << endl;
     a_pet.eat();
}
```

```
int main() {
    Cat felix;
    Pet peeve;

    poly(peeve);

}
```

```
% g++ -std=c++11 inheritance1.cpp -o inheritance1.o
% inheritance1.o
passed in a pet
eating
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};

void poly(___ a_pet) {
     cout << "passed in a pet" << endl;
     a_pet.eat();
}
```

```
int main() {
    Cat felix;
    Pet peeve;

    poly(peeve);
    poly(felix);

}
```

```
% g++ -std=c++11 inheritance1.cpp -o inheritance1.o
% inheritance1.o
passed in a pet
eating

passed in a pet
Cat eating
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};

void poly(___ a_pet) {
    cout << "passed in a pet" << endl;
    a_pet.eat();
}
```

```
int main() {
    Cat felix;
    Pet peeve;

    poly(peeve);
    poly(felix);

}
```

```
% g++ -std=c++11 inheritance1.cpp -o inheritance1.o
% inheritance1.o
passed in a pet
eating
passed in a pet
Cat eating
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};

void poly(_5_ a_pet) {
    cout << "passed in a pet" << endl;
    a_pet.eat();
}
```

```
int main() {
    Cat felix;
    Pet peeve;

    poly(peeve);
    poly(felix);

}
```

```
% g++ -std=c++11 inheritance1.cpp -o inheritance1.o
% inheritance1.o
passed in a pet
eating
passed in a pet
Cat eating
```

Which type replaces blank #5 for the `a_pet` parameter to enable the desired program output?

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};

void poly(_5_ a_pet) {
    cout << "passed in a pet" << endl;
    a_pet.eat();
}
```

```
int main() {
    Cat felix;
    Pet peeve;

    poly(peeve);
    poly(felix);

}
```

```
% g++ -std=c++11 inheritance1.cpp -o inheritance1.o
% inheritance1.o
passed in a pet
eating
passed in a pet
Cat eating
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};

void poly(const Pet& a_pet) {
    cout << "passed in a pet" << endl;
    a_pet.eat();
}
```

object cannot be passed by value

```
int main() {
    Cat felix;
    Pet peeve;

    poly(peeve);
    poly(felix);

}
```

```
% g++ -std=c++11 inheritance1.cpp -o inheritance1.o
% inheritance1.o
passed in a pet
eating
passed in a pet
Cat eating
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};

void poly(const Pet& a_pet) {
    cout << "passed in a pet" << endl;
    a_pet.eat();
}
```

```
int main() {
    Cat felix;
    Pet peeve;

    poly(peeve);
    poly(felix);

    Slug sluggo;
    Roach archie;

    vector<___> pets;
    pets.push_back(___); // add felix
    pets.push_back(___); // add peeve
    pets.push_back(___); // add sluggo
    pets.push_back(___); // add archie
    for (size_t i = 0; i < pets.size(); ++i) {
        // invoke pet's eat method
    }
}
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};

void poly(const Pet& a_pet) {
    cout << "passed in a pet" << endl;
    a_pet.eat();
}
```

```
int main() {
    Cat felix;
    Pet peeve;

    poly(peeve);
    poly(felix);

    Slug sluggo;
    Roach archie;

    vector<_6_> pets;
    pets.push_back(___); // add felix
    pets.push_back(___); // add peeve
    pets.push_back(___); // add sluggo
    pets.push_back(___); // add archie
    for (size_t i = 0; i < pets.size(); ++i) {
       // invoke pet's eat method
    }
}
```

Which type replaces blank #6 so that each pet's most specific `eat()` method will be invoked within the for loop?

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};

void poly(const Pet& a_pet) {
    cout << "passed in a pet" << endl;
    a_pet.eat();
}
```

```
int main() {
    Cat felix;
    Pet peeve;

    poly(peeve);
    poly(felix);

    Slug sluggo;
    Roach archie;

    vector<_6_> pets;
    pets.push_back(___); // add felix
    pets.push_back(___); // add peeve
    pets.push_back(___); // add sluggo
    pets.push_back(___); // add archie
    for (size_t i = 0; i < pets.size(); ++i) {
       // invoke pet's eat method
    }
}
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};

void poly(const Pet& a_pet) {
    cout << "passed in a pet" << endl;
    a_pet.eat();
}
```

```
int main() {
    Cat felix;
    Pet peeve;

    poly(peeve);
    poly(felix);

    Slug sluggo;
    Roach archie;

    vector<Pet*> pets;
    pets.push_back(___); // add felix
    pets.push_back(___); // add peeve
    pets.push_back(___); // add sluggo
    pets.push_back(___); // add archie
    for (size_t i = 0; i < pets.size(); ++i) {
       // invoke pet's eat method
    }
}
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};

void poly(const Pet& a_pet) {
    cout << "passed in a pet" << endl;
    a_pet.eat();
}
```

```
int main() {
    Cat felix;
    Pet peeve;

    poly(peeve);
    poly(felix);

    Slug sluggo;
    Roach archie;

    vector<Pet*> pets;
    pets.push_back(_7_); // add felix
    pets.push_back(___); // add peeve
    pets.push_back(___); // add sluggo
    pets.push_back(___); // add archie
    for (size_t i = 0; i < pets.size(); ++i) {
       // invoke pet's eat method
    }
}
```

How do we add a pointer (replacing blank #7) to the `Cat felix` to the `pets` vector?

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};

void poly(const Pet& a_pet) {
    cout << "passed in a pet" << endl;
    a_pet.eat();
}
```

```
int main() {
    Cat felix;
    Pet peeve;

    poly(peeve);
    poly(felix);

    Slug sluggo;
    Roach archie;

    vector<Pet*> pets;
    pets.push_back(_7_); // add felix
    pets.push_back(___); // add peeve
    pets.push_back(___); // add sluggo
    pets.push_back(___); // add archie
    for (size_t i = 0; i < pets.size(); ++i) {
       // invoke pet's eat method
    }
}
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};

void poly(const Pet& a_pet) {
    cout << "passed in a pet" << endl;
    a_pet.eat();
}
```

```
int main() {
    Cat felix;
    Pet peeve;

    poly(peeve);
    poly(felix);

    Slug sluggo;
    Roach archie;

    vector<Pet*> pets;
    pets.push_back(&felix); // add felix
    pets.push_back(___); // add peeve
    pets.push_back(___); // add sluggo
    pets.push_back(___); // add archie
    for (size_t i = 0; i < pets.size(); ++i) {
       // invoke pet's eat method
    }
}
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};

void poly(const Pet& a_pet) {
    cout << "passed in a pet" << endl;
    a_pet.eat();
}
```

```
int main() {
    Cat felix;
    Pet peeve;

    poly(peeve);
    poly(felix);

    Slug sluggo;
    Roach archie;

    vector<Pet*> pets;
    pets.push_back(&felix); // add felix
    pets.push_back(_8_); // add peeve
    pets.push_back(___); // add sluggo
    pets.push_back(___); // add archie
    for (size_t i = 0; i < pets.size(); ++i) {
       // invoke pet's eat method
    }
}
```

How do we add a pointer (replacing blank #8) to the `Pet peeve` to the `pets` vector?

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};

void poly(const Pet& a_pet) {
    cout << "passed in a pet" << endl;
    a_pet.eat();
}
```

```
int main() {
    Cat felix;
    Pet peeve;

    poly(peeve);
    poly(felix);

    Slug sluggo;
    Roach archie;

    vector<Pet*> pets;
    pets.push_back(&felix); // add felix
    pets.push_back(_8_); // add peeve
    pets.push_back(___); // add sluggo
    pets.push_back(___); // add archie
    for (size_t i = 0; i < pets.size(); ++i) {
       // invoke pet's eat method
    }
}
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};

void poly(const Pet& a_pet) {
    cout << "passed in a pet" << endl;
    a_pet.eat();
}
```

```
int main() {
    Cat felix;
    Pet peeve;

    poly(peeve);
    poly(felix);

    Slug sluggo;
    Roach archie;

    vector<Pet*> pets;
    pets.push_back(&felix); // add felix
    pets.push_back(&peeve); // add peeve
    pets.push_back(___); // add sluggo
    pets.push_back(___); // add archie
    for (size_t i = 0; i < pets.size(); ++i) {
       // invoke pet's eat method
    }
}
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};

void poly(const Pet& a_pet) {
    cout << "passed in a pet" << endl;
    a_pet.eat();
}
```

```
int main() {
    Cat felix;
    Pet peeve;

    poly(peeve);
    poly(felix);

    Slug sluggo;
    Roach archie;

    vector<Pet*> pets;
    pets.push_back(&felix); // add felix
    pets.push_back(&peeve); // add peeve
    pets.push_back(&sluggo); // add sluggo
    pets.push_back(___); // add archie
    for (size_t i = 0; i < pets.size(); ++i) {
       // invoke pet's eat method
    }
}
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};

void poly(const Pet& a_pet) {
    cout << "passed in a pet" << endl;
    a_pet.eat();
}
```

```
int main() {
    Cat felix;
    Pet peeve;

    poly(peeve);
    poly(felix);

    Slug sluggo;
    Roach archie;

    vector<Pet*> pets;
    pets.push_back(&felix); // add felix
    pets.push_back(&peeve); // add peeve
    pets.push_back(&sluggo); // add sluggo
    pets.push_back(&archie); // add archie
    for (size_t i = 0; i < pets.size(); ++i) {
       // invoke pet's eat method
    }
}
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};

void poly(const Pet& a_pet) {
    cout << "passed in a pet" << endl;
    a_pet.eat();
}
```

```
int main() {
    Cat felix;
    Pet peeve;

    poly(peeve);
    poly(felix);

    Slug sluggo;
    Roach archie;

    vector<Pet*> pets;
    pets.push_back(&felix); // add felix
    pets.push_back(&peeve); // add peeve
    pets.push_back(&sluggo); // add sluggo
    pets.push_back(&archie); // add archie
    for (size_t i = 0; i < pets.size(); ++i) {
       // invoke pet's eat method
       ___
    }
}
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};

void poly(const Pet& a_pet) {
    cout << "passed in a pet" << endl;
    a_pet.eat();
}
```

```
int main() {
    Cat felix;
    Pet peeve;

    poly(peeve);
    poly(felix);

    Slug sluggo;
    Roach archie;

    vector<Pet*> pets;
    pets.push_back(&felix); // add felix
    pets.push_back(&peeve); // add peeve
    pets.push_back(&sluggo); // add sluggo
    pets.push_back(&archie); // add archie
    for (size_t i = 0; i < pets.size(); ++i) {
       // invoke pet's eat method
       _9_
    }
}
```

Which expression will replaces blank #9 to invoke the `eat()` method on each of the pets with a pointer in the `pets` vector?

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};

void poly(const Pet& a_pet) {
    cout << "passed in a pet" << endl;
    a_pet.eat();
}
```

```
int main() {
    Cat felix;
    Pet peeve;

    poly(peeve);
    poly(felix);

    Slug sluggo;
    Roach archie;

    vector<Pet*> pets;
    pets.push_back(&felix); // add felix
    pets.push_back(&peeve); // add peeve
    pets.push_back(&sluggo); // add sluggo
    pets.push_back(&archie); // add archie
    for (size_t i = 0; i < pets.size(); ++i) {
       // invoke pet's eat method
       _9_
    }
}
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};

void poly(const Pet& a_pet) {
    cout << "passed in a pet" << endl;
    a_pet.eat();
}
```

```
int main() {
    Cat felix;
    Pet peeve;

    poly(peeve);
    poly(felix);

    Slug sluggo;
    Roach archie;

    vector<Pet*> pets;
    pets.push_back(&felix); // add felix
    pets.push_back(&peeve); // add peeve
    pets.push_back(&sluggo); // add sluggo
    pets.push_back(&archie); // add archie
    for (size_t i = 0; i < pets.size(); ++i) {
       // invoke pet's eat method
       pets[i]->eat();
    }
}
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};

void poly(const Pet& a_pet) {
    cout << "passed in a pet" << endl;
    a_pet.eat();
}
```

```
int main() {
    Cat felix;
    Pet peeve;

    poly(peeve);
    poly(felix);

    Slug sluggo;
    Roach archie;

    vector<Pet*> pets;
    pets.push_back(&felix); // add felix
    pets.push_back(&peeve); // add peeve
    pets.push_back(&sluggo); // add sluggo
    pets.push_back(&archie); // add archie
    for (size_t i = 0; i < pets.size(); ++i) {
       // invoke pet's eat method
       pets[i]->eat();
    }
}
```

Representing pets

```
class Pet{
public:
    void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};

void poly(const Pet& a_pet) {
    cout << "passed in a pet" << endl;
    a_pet.eat();
}
```

```
int main() {
    Cat felix;
    Pet peeve;

    poly(peeve);
    poly(felix);

    Slug sluggo;
    Roach archie;

    vector<Pet*> pets;
    pets.push_back(&felix); // add felix
    pets.push_back(&peeve); // add peeve
    pets.push_back(&sluggo); // add sluggo
    pets.push_back(&archie); // add archie
    for (size_t i = 0; i < pets.size(); ++i) {
       pets[i]->eat();
    }
}
```

```
% g++ -std=c++11 inheritance1.cpp -o inheritance1.o
% inheritance1.o
…
eating
eating
eating
eating
```

Pet version of eat() always invoked

Representing pets

```
class Pet{
public:
    ___ void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};

void poly(const Pet& a_pet) {
    cout << "passed in a pet" << endl;
    a_pet.eat();
}
```

```
int main() {
    Cat felix;
    Pet peeve;

    poly(peeve);
    poly(felix);

    Slug sluggo;
    Roach archie;

    vector<Pet*> pets;
    pets.push_back(&felix); // add felix
    pets.push_back(&peeve); // add peeve
    pets.push_back(&sluggo); // add sluggo
    pets.push_back(&archie); // add archie
    for (size_t i = 0; i < pets.size(); ++i) {
       pets[i]->eat();
    }
}
```

Pet version of eat() always invoked

Representing pets

```
class Pet{
public:
    _10_ void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};

void poly(const Pet& a_pet) {
    cout << "passed in a pet" << endl;
    a_pet.eat();
}
```

```
int main() {
    Cat felix;
    Pet peeve;

    poly(peeve);
    poly(felix);

    Slug sluggo;
    Roach archie;

    vector<Pet*> pets;
    pets.push_back(&felix); // add felix
    pets.push_back(&peeve); // add peeve
    pets.push_back(&sluggo); // add sluggo
    pets.push_back(&archie); // add archie
    for (size_t i = 0; i < pets.size(); ++i) {
       pets[i]->eat();
    }
}
```

Pet version of eat() always invoked

Which keyword replaces blank #10 to enable the version of `eat()` that is invoked to be determined dynamically based on the most specific method definition?

```
class Pet{
public:
    _10_ void eat() { cout << "eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};

void poly(const Pet& a_pet) {
    cout << "passed in a pet" << endl;
    a_pet.eat();
}
```

```
int main() {
    Cat felix;
    Pet peeve;

    poly(peeve);
    poly(felix);

    Slug sluggo;
    Roach archie;

    vector<Pet*> pets;
    pets.push_back(&felix); // add felix
    pets.push_back(&peeve); // add peeve
    pets.push_back(&sluggo); // add sluggo
    pets.push_back(&archie); // add archie
    for (size_t i = 0; i < pets.size(); ++i) {
       pets[i]->eat();
    }
}
```

Pet version of eat() always invoked

Representing pets

```
class Pet{
public:
    virtual void eat() const { cout <<
"eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};

void poly(const Pet& a_pet) {
    cout << "passed in a pet" << endl;
    a_pet.eat();
}
```

```
int main() {
    Cat felix;
    Pet peeve;

    poly(peeve);
    poly(felix);

    Slug sluggo;
    Roach archie;

    vector<Pet*> pets;
    pets.push_back(&felix); // add felix
    pets.push_back(&peeve); // add peeve
    pets.push_back(&sluggo); // add sluggo
    pets.push_back(&archie); // add archie
    for (size_t i = 0; i < pets.size(); ++i) {
       pets[i]->eat();
    }
}
```

```
% g++ -std=c++11 inheritance1.cpp -o inheritance1.o
% inheritance1.o
…
Cat eating
eating
eating
eating
```

~~Pet version of eat() always invoked~~

Representing pets (so far)

```
class Pet{
public:
    virtual void eat() const { cout <<
"eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};
```

In-class problem II

Constructing pets

```
class Pet{
public:
    virtual void eat() const { cout <<
"eating\n"; }
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};
```

Constructing pets

```
class Pet{
public:
    virtual void eat() const { cout <<
"eating\n"; }
private:
    string name;
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};
```

Constructing pets

```
class Pet{
public:
    Pet(const string& name) : name(name) {}
    virtual void eat() const { cout <<
"eating\n"; }
private:
    string name;
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};
```

```
int main() {
    Pet peeve;
    Cat felix;

    Slug sluggo;
    Roach archie;
}
```

compilation errors!

Constructing pets

```
class Pet{
public:
    Pet(const string& name) : name(name) {}
    virtual void eat() const { cout <<
"eating\n"; }
private:
    string name;
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
};
class Slug : public Pet {};
class Roach : public Pet {};
```

```
int main() {
    Pet peeve("Peeve");
    Cat felix;

    Slug sluggo;
    Roach archie;
}
```

compilation errors!

Constructing pets

```
class Pet{
public:
    Pet(const string& name) : name(name) {}
    virtual void eat() const { cout <<
"eating\n"; }
private:
    string name;
};

class Cat : public Pet {
public:
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
private:
    string fur_color;
};

class Slug : public Pet {};
class Roach : public Pet {};
```

```
int main() {
    Pet peeve("Peeve");
    Cat felix;

    Slug sluggo;
    Roach archie;
}
```

compilation errors!

Constructing pets

```
class Pet{
public:
    Pet(const string& name) : name(name) {}
    virtual void eat() const { cout <<
"eating\n"; }
private:
    string name;
};

class Cat : public Pet {
public:
    Cat(const string& name, const string& color)
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
private:
    string fur_color;
};

class Slug : public Pet {};
class Roach : public Pet {};
```

```
int main() {
    Pet peeve("Peeve");
    Cat felix;

    Slug sluggo;
    Roach archie;
}
```

compilation errors!

Constructing pets

```
class Pet{
public:
    Pet(const string& name) : name(name) {}
    virtual void eat() const { cout <<
"eating\n"; }
private:
    string name;
};

class Cat : public Pet {
public:
    Cat(const string& name, const string& color)
        : ___, ___ {}
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
private:
    string fur_color;
};

class Slug : public Pet {};
class Roach : public Pet {};
```

```
int main() {
    Pet peeve("Peeve");
    Cat felix;

    Slug sluggo;
    Roach archie;
}
```

compilation errors!

Constructing pets

```
class Pet{
public:
    Pet(const string& name) : name(name) {}
    virtual void eat() const { cout <<
"eating\n"; }
private:
    string name;
};

class Cat : public Pet {
public:
    Cat(const string& name, const string& color)
        : _11_, ___ {}
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
private:
    string fur_color;
};

class Slug : public Pet {};
class Roach : public Pet {};
```

```
int main() {
    Pet peeve("Peeve");
    Cat felix;

    Slug sluggo;
    Roach archie;
}
```

compilation errors!

Which expression replaces blank #11 to initialize the `name` of the `Cat` object to the same value as the `name` parameter?

```
class Pet{
public:
    Pet(const string& name) : name(name) {}
    virtual void eat() const { cout <<
"eating\n"; }
private:
    string name;
};

class Cat : public Pet {
public:
    Cat(const string& name, const string& color)
        : _11_, ___ {}
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
private:
    string fur_color;
};

class Slug : public Pet {};
class Roach : public Pet {};
```

```
int main() {
    Pet peeve("Peeve");
    Cat felix;

    Slug sluggo;
    Roach archie;
}
```

compilation errors!

Constructing pets

```
class Pet{
public:
    Pet(const string& name) : name(name) {}
    virtual void eat() const { cout <<
"eating\n"; }
private:
    string name;
};

class Cat : public Pet {
public:
    Cat(const string& name, const string& color)
        : Pet(name), ___ {}
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
private:
    string fur_color;
};

class Slug : public Pet {};
class Roach : public Pet {};
```

```
int main() {
    Pet peeve("Peeve");
    Cat felix;

    Slug sluggo;
    Roach archie;
}
```

compilation errors!

Constructing pets

```
class Pet{
public:
    Pet(const string& name) : name(name) {}
    virtual void eat() const { cout <<
"eating\n"; }
private:
    string name;
};

class Cat : public Pet {
public:
    Cat(const string& name, const string& color)
        : Pet(name), _12_ {}
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
private:
    string fur_color;
};

class Slug : public Pet {};
class Roach : public Pet {};
```

```
int main() {
    Pet peeve("Peeve");
    Cat felix;

    Slug sluggo;
    Roach archie;
}
```

compilation errors!

Which expression replaces blank #12 to initialize the `fur_color` variable to the same `string` value as the `color` parameter?

```
class Pet{
public:
    Pet(const string& name) : name(name) {}
    virtual void eat() const { cout << "eating\n"; }
private:
    string name;
};

class Cat : public Pet {
public:
    Cat(const string& name, const string& color)
        : Pet(name), _12_ {}
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
private:
    string fur_color;
};

class Slug : public Pet {};
class Roach : public Pet {};
```

```
int main() {
    Pet peeve("Peeve");
    Cat felix;

    Slug sluggo;
    Roach archie;
}
```

compilation errors!

Constructing pets

```
class Pet{
public:
    Pet(const string& name) : name(name) {}
    virtual void eat() const { cout << "eating\n"; }
private:
    string name;
};

class Cat : public Pet {
public:
    Cat(const string& name, const string& color)
        : Pet(name), fur_color(color) {}
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
private:
    string fur_color;
};

class Slug : public Pet {};
class Roach : public Pet {};
```

```
int main() {
    Pet peeve("Peeve");
    Cat felix;

    Slug sluggo;
    Roach archie;
}
```

compilation errors!

Constructing pets

```
class Pet{
public:
    Pet(const string& name) : name(name) {}
    virtual void eat() const { cout << "eating\n"; }
private:
    string name;
};

class Cat : public Pet {
public:
    Cat(const string& name, const string& color)
        : Pet(name), fur_color(color) {}
    void eat() const {
        cout << "Cat ";
        Pet::eat();
    }
private:
    string fur_color;
};

class Slug : public Pet {};
class Roach : public Pet {};
```

constructors needed

```
int main() {
    Pet peeve("Peeve");
    Cat felix("Felix", "grey");

    Slug sluggo;
    Roach archie;
}
```

compilation errors!

The `final` keyword

Preventing overriding

```
class Pet{
public:
    virtual void communicate() = 0;
};

class Dog : public Pet {
public:
    void communicate() { cout << "Woof!"; }
};


class Poodle : public Dog {};
class Bulldog : public Dog {};


int main() {
    Poodle pete;
    Bulldog billy;

    pete.communicate();
    billy.communicate();
}
```

```
% g++ --std=c++11 final_kw.cpp -o final_kw.o
% ./final_kw.o
Woof!
Woof!
```

Preventing overriding

```
class Pet{
public:
    virtual void communicate() = 0;
};
class Dog : public Pet {
public:
    void communicate() { cout << "Woof!" << endl; }
};

class Poodle : public Dog {
public:
    void communicate() {
        cout << "woof..." << endl;
    }
};
class Bulldog : public Dog {
public:
    void communicate() {
        cout << "WOOF!!!" << endl;
    }
};
```

```
int main() {
    Poodle pete;
    Bulldog billy;

    pete.communicate();
    billy.communicate();
}
```

```
% g++ --std=c++11 final_kw.cpp -o final_kw.o
% ./final_kw.o
woof…
WOOF!!!
```

Preventing overriding

```
class Pet{
public:
    virtual void communicate() = 0;
};
class Dog : public Pet {
public:
    void communicate() { cout << "Woof!" << endl; }
};

class Poodle : public Dog {
public:
    void communicate() {
        cout << "woof..." << endl;
    }
};
class Bulldog : public Dog {
public:
    void communicate() {
        cout << "WOOF!!!" << endl;
    }
};
```

```
int main() {
    Poodle pete;
    Bulldog billy;

    pete.communicate();
    billy.communicate();
}
```

What if we want all Dogs to communicate in the same way with no exceptions?

Preventing overriding

```
class Pet{
public:
    virtual void communicate() = 0;
};
class Dog : public Pet {
public:
    void communicate() final { cout << "Woof!" << endl; }
};

class Poodle : public Dog {
public:
    void communicate() {
        cout << "woof..." << endl;
    }
};
class Bulldog : public Dog {
public:
    void communicate() {
        cout << "WOOF!!!" << endl;
    }
};
```

prevents any further overriding in descendant classes

```
int main() {
    Poodle pete;
    Bulldog billy;

    pete.communicate();
    billy.communicate();
}
```

```
% g++ --std=c++11 final_kw.cpp -o final_kw.o
final_kw.cpp:16:8: error: declaration of 'communicate' overrides a 'final' function
  void communicate() { cout << "woof..." << endl; }
       ^
final_kw.cpp:11:8: note: overridden virtual function is here
  void communicate() final { cout << "Woof!" << endl; }
       ^
final_kw.cpp:20:8: error: declaration of 'communicate' overrides a 'final' function
  void communicate() { cout << "WOOF!!!" << endl; }
       ^
final_kw.cpp:11:8: note: overridden virtual function is here
  void communicate() final { cout << "Woof!" << endl; }
       ^
2 errors generated.
```