RS Setup
Login: student.turningtechnologies.com
Session ID: 20220330<A|D>

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

Inheritance III

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

Agenda

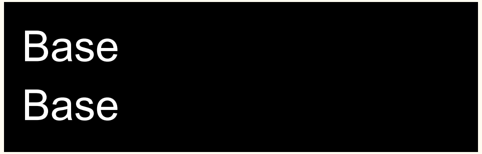
- Polymorphism (continued)
- Constructors
- Protected members
- Interfaces
- Overriding vs. overloading
- In-class Problem

Polymorphism

Polymorphism with non-members

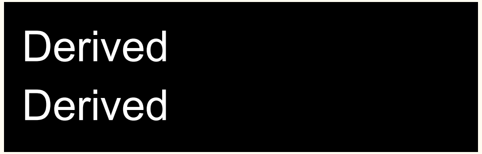
```
class Base {
public:

};

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

class Derived : public Base {
public:

};

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

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

observed



```
Base
Base
```

want

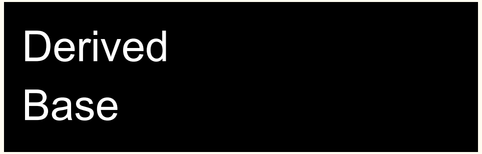


```
Derived
Derived
```

Polymorphism with non-members

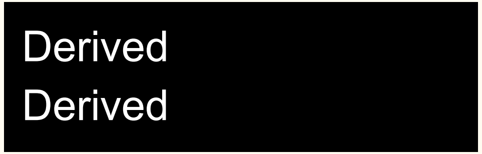
```
class Base {
public:

};

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

class Derived : public Base {
public:

};

ostream& operator<<(ostream& os, const Derived& rhs) {
    os << "Derived";
    return os;
}

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

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

observed



```
Derived
Base
```

want



```
Derived
Derived
```

Polymorphism with non-members

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

ostream& operator<<(ostream& os, const Base& rhs) {
    ~~os << "Base"~~;
    return os;
}

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

ostream& operator<<(ostream& os, const Derived& rhs) {
    os << "Derived";
    return os;
}

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

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

Polymorphism with non-members

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

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

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

ostream& operator<<(ostream& os, const Derived& rhs) {
    os << "Derived";
    return os;
}

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

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

Polymorphism with non-members

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

ostream& operator<<(ostream& os, const Base& rhs) {
    _1_
    return os;
}

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

ostream& operator<<(ostream& os, const Derived& rhs) {
    os << "Derived";
    return os;
}

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

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

TurningPoint

SRS Setup
Login: student.turningtechnologies.com
Session ID: 20220330<A|D>

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

Which expression replaces blank #1 to call the correct method for displaying `rhs` at **runtime**?

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

ostream& operator<<(ostream& os, const Base& rhs) {
    _1_
    return os;
}

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

ostream& operator<<(ostream& os, const Derived& rhs) {
    os << "Derived";
    return os;
}

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

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

Polymorphism with non-members

```cpp
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"; }
};

ostream& operator<<(ostream& os, const Derived& rhs) {
    os << "Derived";
    return os;
}

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

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

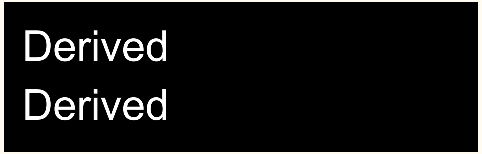
Polymorphism with non-members

```
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"; }
};

~~ostream& operator<<(ostream& os, const Derived& rhs) {~~
~~    os << "Derived";~~
~~    return os;~~
~~}~~

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

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

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

Constructors

Inheriting member variables

```cpp
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 {};
```

```cpp
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

```
class Animal {};

class Lion : public Animal {};

class Tiger : public Animal {};

class Bear : public Animal {};


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

invokes Animal() constructor

invokes Bear() constructor

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 base 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);

}
```

```
Base
```

Polymorphism in constructors

calls base class implementation
(`virtual` or not)

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

normal polymorphism
rules apply

```
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();
}
```

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

Protected members

`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) {}
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;
}
```

`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) {}
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!

`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;
};
```

private even for derived classes

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

```
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;
}
```

compilation error!

`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
    void set_x(int val) { ___ }
private:
    int x_mem; } private even for derived classes
};

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

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

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
    void set_x(int val) { x_mem = val; }
private:
    int x_mem; } private even for derived classes
};

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

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

`protected` mode

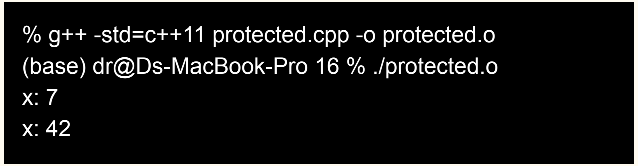
```
class Base {
    friend ostream& operator<<(ostream& os, const Base& base) {
        return os <<  "x: " << base.x_mem;
    }
public:
    Base(int x_val) : x_mem(x_val) {}
protected:
    // define a mutator method for modifying x_mem
    void set_x(int val) { x_mem = val; }
private:
    int x_mem;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

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

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

private even for derived classes

~~compilation error!~~

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:
    void set_x(int val) { x_mem = val; }
private:
    int x_mem;
};

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

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



```
% g++ -std=c++11 protected.cpp -o protected.o
(base) dr@Ds-MacBook-Pro 16 % ./protected.o
x: 7
x: 42
```

`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) {}
};
```

```
int main() {
    Cat felix("Felix");
    cout << felix.get_name();   ✘
}
```

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; }
};
```

```
int main() {
    Cat felix("Felix");
    ~~cout << felix.get_name();~~  ✘
    felix.display(); ✔
}
```

```
 % g++ -std=c++11 protected2.cpp -o protected2.o
% ./protected2.o
Felix
```

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;
    }
};
```

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

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

`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; ✘
    }
};
```

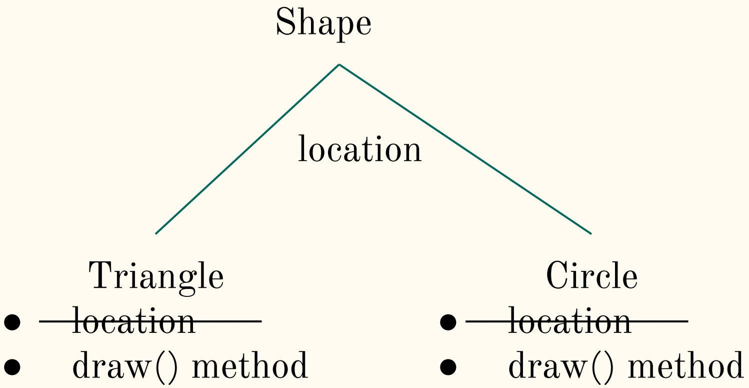
get_name() method only accessible for current object or object of same type

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

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

Interfaces

Implementing an interface

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

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



Shape

location

Triangle

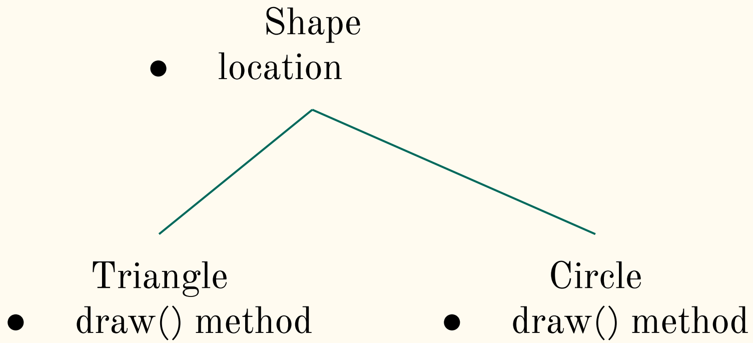
- ~~location~~
- draw() method

Circle

- ~~location~~
- draw() method

Implementing an interface

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

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



Shape

- location

Triangle

- draw() method

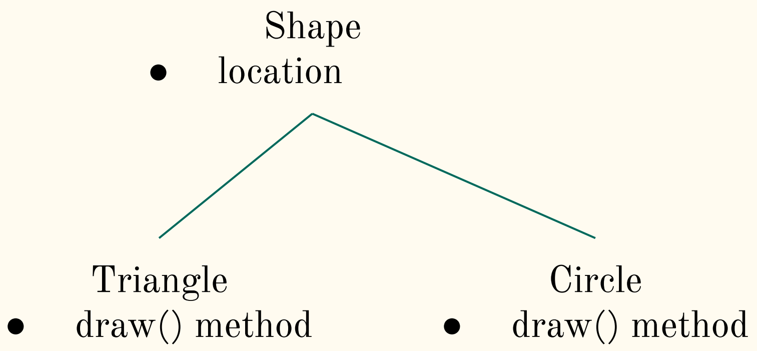
Circle

- draw() method

draw() methods have different behavior

Implementing an interface

```
class Shape {
public:
    Shape(int x, int y) : x(x), y(y) {}
private:
    int x, y;
};


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

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



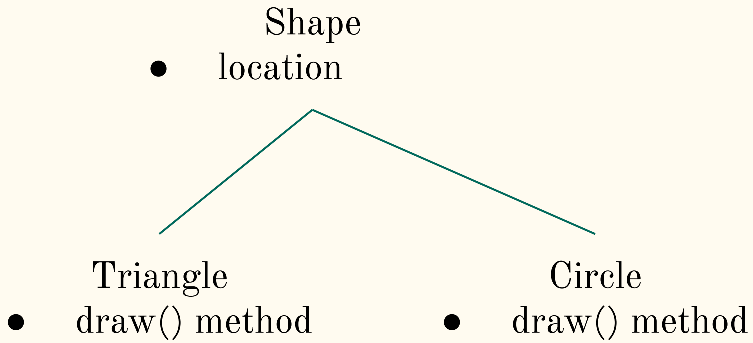
Shape
- location

Triangle
- draw() method

Circle
- draw() method

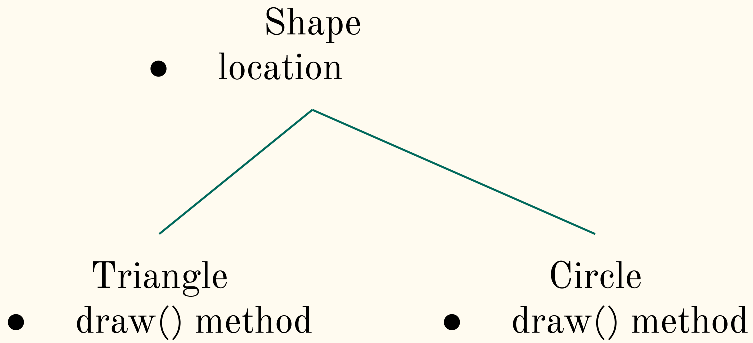
Implementing an interface

```
class Shape {
public:
    Shape(int x, int y) : x(x), y(y) {}
private:
    int x, y;
};


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

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



Implementing an interface

```
class Shape {
public:
    Shape(int x, int y) : x(x), y(y) {}
private:
    int x, y;
};


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

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



Shape

- location

Triangle

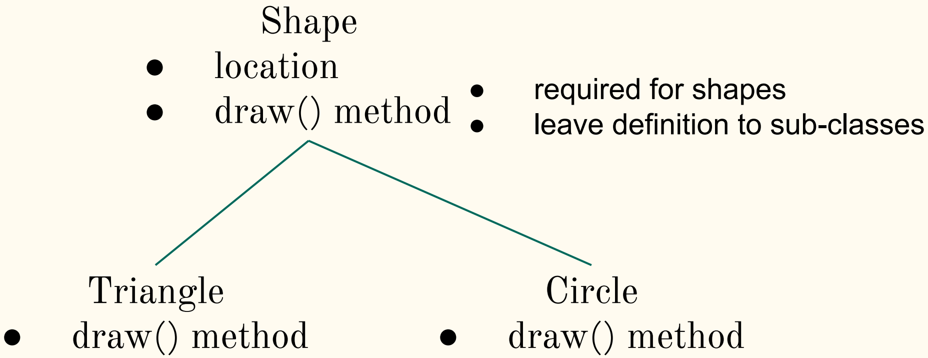
- draw() method

Circle

- draw() method

Implementing an interface

```
class Shape {
public:
    Shape(int x, int y) : x(x), y(y) {}
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";
    }
};
```



Shape

- location
- draw() method
  - required for shapes
  - leave definition to sub-classes

Triangle

- draw() method

Circle

- draw() method

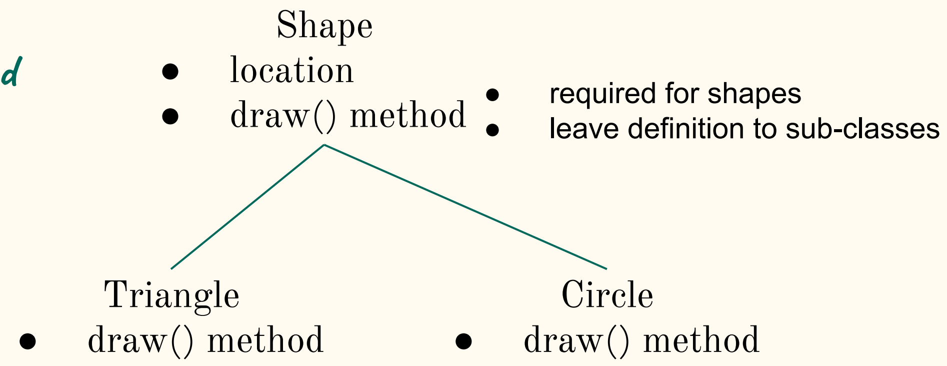
Implementing an interface

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

abstract class

abstract/pure virtual method

```
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";
    }
};
```



Shape

- location
- draw() method
  - required for shapes
  - leave definition to sub-classes

Triangle

- draw() method

Circle

- draw() method

Implementing an interface

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

abstract class

prevents class from
being instantiated

```
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";
    }
};
```

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

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

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

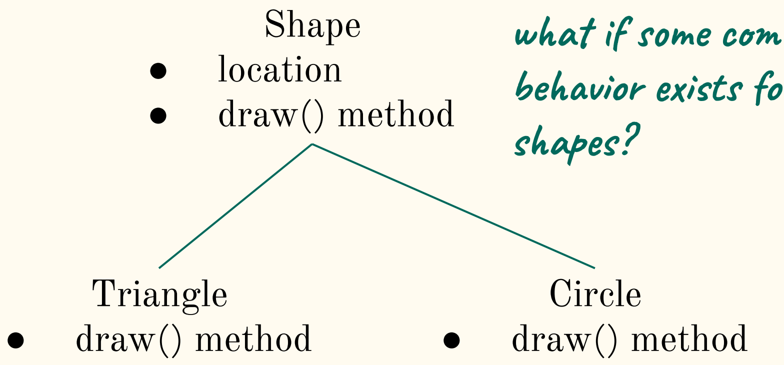
compilation error!

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

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

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


Shape
- location
- draw() method

Triangle
- draw() method

Circle
- draw() method

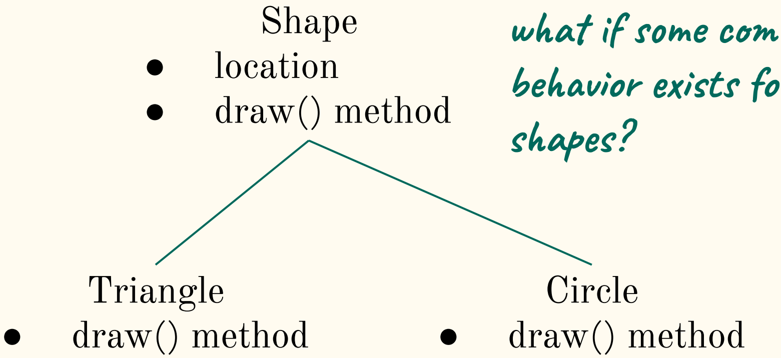
what if some common behavior exists for drawing shapes?

Implementing an interface

abstract class

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

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

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



Shape
- location
- draw() method

Triangle
- draw() method

Circle
- draw() method

what if some common behavior exists for drawing shapes?

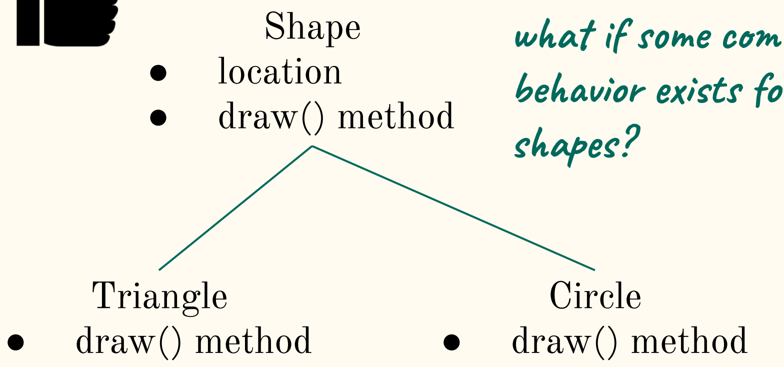
Implementing an interface

~~abstract class~~

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

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";
    }
};
```

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

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

code reuse



Shape
- location
- draw() method

Triangle
- draw() method

Circle
- draw() method

what if some common behavior exists for drawing shapes?

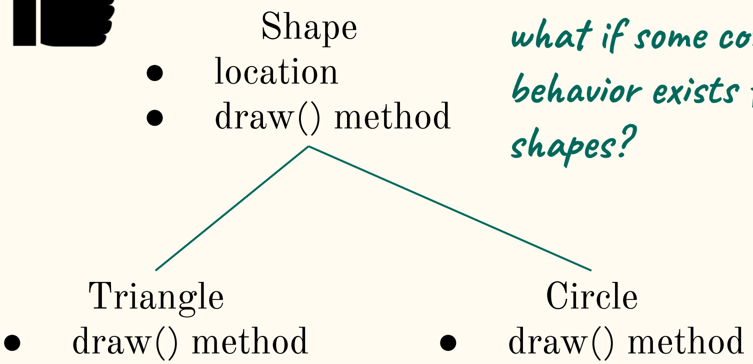
Implementing an interface

```
class Shape {   ~~abstract class~~
public:
    Shape(int x, int y) : x(x), y(y) {}
    //virtual void draw() = 0;
    ~~virtual void draw() { cout << "Default stuff... "; }~~
private:
    int x, y;
};

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";
    }
};
```

code reuse 👍

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

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

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

undesired behavior 👎



Shape
- location
- draw() method

Triangle
- draw() method

Circle
- draw() method

what if some common behavior exists for drawing shapes?

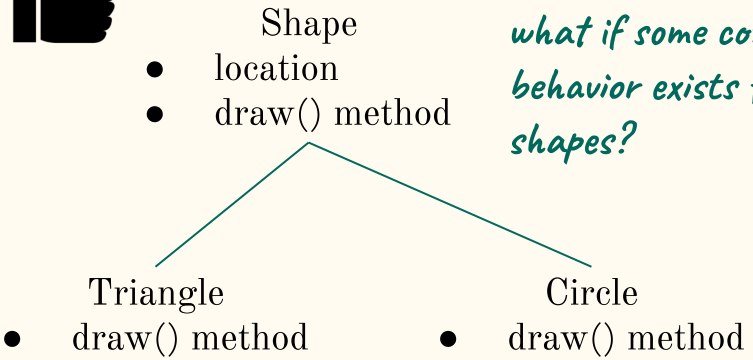
Implementing an interface

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

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);
}
```



Shape
- location
- draw() method

Triangle
- draw() method

Circle
- draw() method

what if some common behavior exists for drawing shapes?

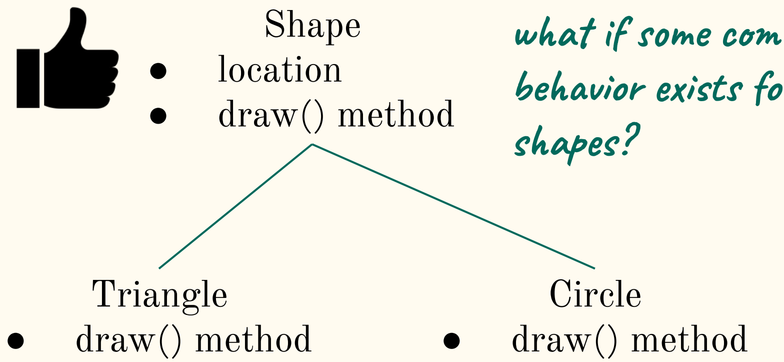
Implementing an interface

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



Shape
- location
- draw() method

Triangle
- draw() method

Circle
- draw() method

what if some common behavior exists for drawing shapes?

Implementing an interface

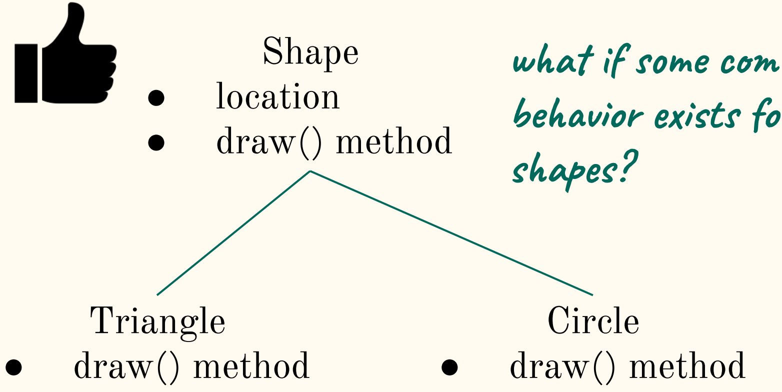
```
class Shape {
public:
    Shape(int x, int y) : x(x), y(y) {}
    virtual void draw() = 0;
private:
    int x, y;
};

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! 👍



Shape
- location
- draw() method

Triangle
- draw() method

Circle
- draw() method

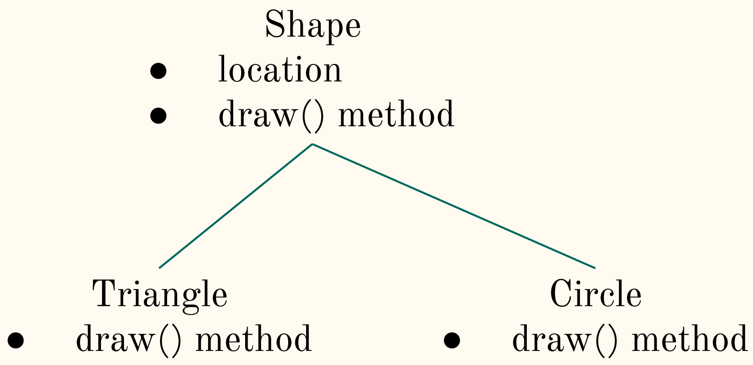
what if some common behavior exists for drawing shapes?

Implementing an interface

abstract class

```
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";
    }
};
```

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

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



Shape
- location
- draw() method

Triangle
- draw() method

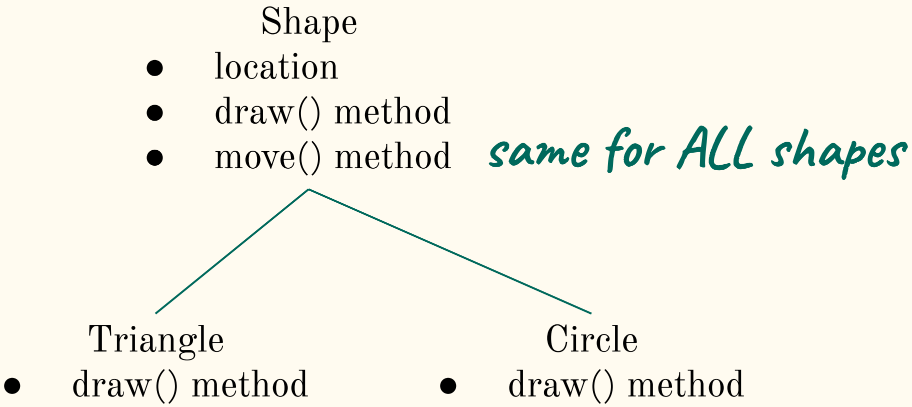
Circle
- draw() method

Implementing an interface

abstract class

```
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";
    }
};
```

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

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



Shape
- location
- draw() method
- move() method

same for ALL shapes

Triangle
- draw() method

Circle
- draw() method

Implementing an interface

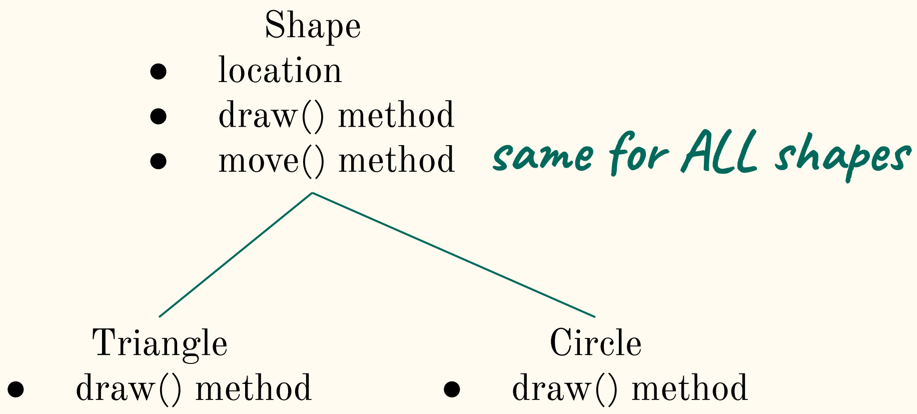
```
class Shape {
public:
    Shape(int x, int y) : x(x), y(y) {}
    virtual void draw() = 0;
    void move(int x, int y) {
        this->x = x;
        this->y = y;
    }
private:
    int x, y;
};
void Shape::draw() { cout << "Default stuff... "; }
```

abstract class

<u>only 1</u> method needs to be pure virtual/abstract

```
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";
    }
};
```

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

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



Shape
- location
- draw() method
- move() method

same for ALL shapes

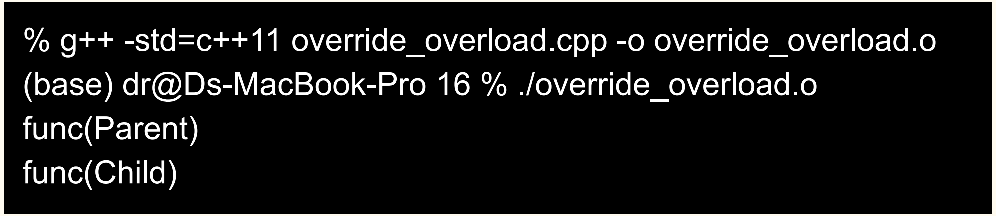
Triangle
- draw() method

Circle
- draw() method

Overriding vs overloading

Overloading

```
class Parent {};
class Child : public Parent {};

void func(const Parent& base) { cout << "func(Parent)\n"; }
void func(const Child& derived) { cout << "func(Child)\n"; }

int main() {
    Parent parent;
    func(parent);

    Child child;
    func(child);
}
```



```
% g++ -std=c++11 override_overload.cpp -o override_overload.o
(base) dr@Ds-MacBook-Pro 16 % ./override_overload.o
func(Parent)
func(Child)
```

Overloading

```
class Parent {};
class Child : public Parent {};
class Grandchild : public Child {};

void func(const Parent& base) { cout << "func(Parent)\n"; }
void func(const Child& derived) { cout << "func(Child)\n"; }

int main() {
    Parent parent;
    func(parent);

    Child child;
    func(child);

    Grandchild gc;
    func(gc);

}
```

Which version of `func()` gets called when `gc` is the argument?

```
class Parent {};
class Child : public Parent {};
class Grandchild : public Child {};

void func(const Parent& base) { cout << "func(Parent)\n"; }
void func(const Child& derived) { cout << "func(Child)\n"; }

int main() {
    Parent parent;
    func(parent);

    Child child;
    func(child);

    Grandchild gc;
    func(gc);

}
```

A. `func(const Parent& base)`

B. `func(const Child& derived)`

Overloading

```
class Parent {};
class Child : public Parent {};
class Grandchild : public Child {};

void func(const Parent& base) { cout << "func(Parent)\n"; }
void func(const Child& derived) { cout << "func(Child)\n"; }

int main() {
    Parent parent;
    func(parent);

    Child child;
    func(child);

    Grandchild gc;
    func(gc);

}
```

```
% g++ -std=c++11 override_overload.cpp -o override_overload.o
% ./override_overload.o
func(Parent)
func(Child)
func(Child)
```

Overloading

```
class Parent {};
class Child : public Parent {};
class Grandchild : public Child {};

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);
}
int main() {
    Parent parent;
    func(parent);

    Child child;
    func(child);

    Grandchild gc;
    func(gc);

    other_func(child);
}
```

Which version of `func()` will be called when child is the argument passed to `other_func()`?

```
class Parent {};
class Child : public Parent {};
class Grandchild : public Child {};

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);
}
int main() {
    Parent parent;
    func(parent);

    Child child;
    func(child);

    Grandchild gc;
    func(gc);

    other_func(child);
}
```

A. `func(const Parent& base)`

B. `func(const Child& derived)`

Overloading

```
class Parent {};
class Child : public Parent {};
class Grandchild : public Child {};

void func(const Parent& base) { cout << "func(Parent)\n"; }
```

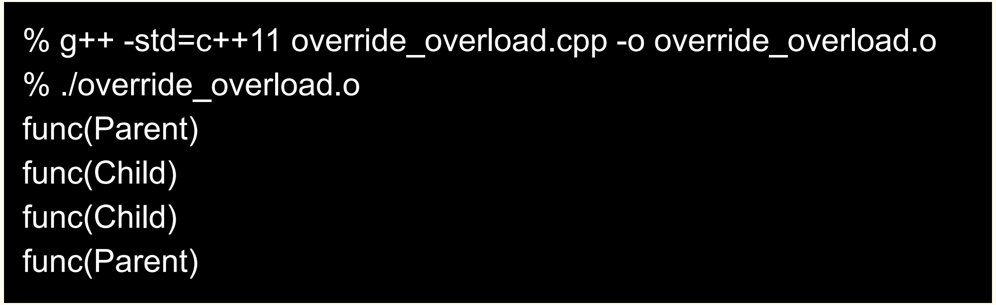
compiler calls this version

```
void func(const Child& derived) { cout << "func(Child)\n"; }

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

compiler sees this type

```
int main() {
    Parent parent;
    func(parent);

    Child child;
    func(child);

    Grandchild gc;
    func(gc);

    other_func(child);
}
```

even when argument is of a descendant class



```
% g++ -std=c++11 override_overload.cpp -o override_overload.o
% ./override_overload.o
func(Parent)
func(Child)
func(Child)
func(Parent)
```

Overriding

```
class Parent {
public:
    virtual void whereami() const {
        cout << "Parent" << endl;
    }
};
```

virtual function

```
class Child : public Parent {
public:
    void whereami() const {
        cout << "Child!!!" << endl;
    }
};
```

overridden

```
class Grandchild : public Child {
public:
    void whereami() const {
        cout << "Grandchild!!!" << endl;
    }
};
```

overridden

```
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();
}
```

```
int main() {
    Parent parent;
    other_func(parent);
}
```

Which class's `whereami()` implementation will be invoked when `other_func(parent)` is called?

```
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();
}
```

```
int main() {
    Parent parent;
    other_func(parent);
}
```

A. Parent
B. Child
C. Grandchild

Overriding

```
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();
}
```

```
int main() {
    Parent parent;
    other_func(parent);
}
```

```
% g++ -std=c++11 overrideload2.cpp -o overrideload2.o
% ./override_overload2.o
func(Parent)
Parent
```

Overriding

```
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();
}
```

```
int main() {
    Parent parent;
    other_func(parent);

    Child child;
    other_func(child);

}
```

Which class's `whereami()` implementation will be invoked when `other_func(child)` is called?

```
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();
}
```

```
int main() {
    Parent parent;
    other_func(parent);

    Child child;
    other_func(child);
}
```

A. Parent
B. Child
C. Grandchild

Overriding

```
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();
}
```

```
int main() {
    Parent parent;
    other_func(parent);

    Child child;
    other_func(child);
}
```

```
% g++ -std=c++11 overrideload2.cpp -o overrideload2.o
% ./overrideload2.o
func(Parent)
Parent
func(Parent)
Child!!!
```

Overriding

```
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();
}
```

```
int main() {
    Parent parent;
    other_func(parent);

    Child child;
    other_func(child);

    Grandchild gc;
    other_func(gc);
}
```

Which class's `whereami()` implementation will be invoked when `other_func(gc)` is called?

```
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();
}
```

```
int main() {
    Parent parent;
    other_func(parent);

    Child child;
    other_func(child);

    Grandchild gc;
    other_func(gc);

}
```

A. Parent
B. Child
C. Grandchild

Overriding

```
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();
}
```

```
int main() {
    Parent parent;
    other_func(parent);

    Child child;
    other_func(child);

    Grandchild gc;
    other_func(gc);
}
```

```
% g++ -std=c++11 overrideload2.cpp -o overrideload2.o
% ./overrideload2.o
func(Parent)
Parent
func(Parent)
Child!!!
func(Parent)
Grandchild!!!
```

Overriding

```
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;
    }
};
```

overriding --

choice made at runtime

```
void func(const Parent& base) { cout << "func(Parent)\n"; }
void func(const Child& derived) { cout << "func(Child)\n"; }
```

overloading --

choice made at compile-time

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