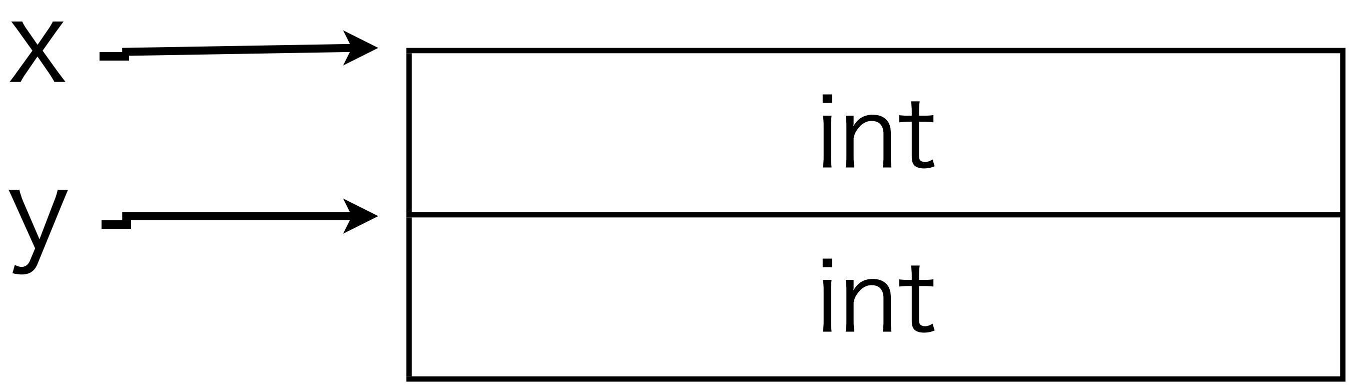
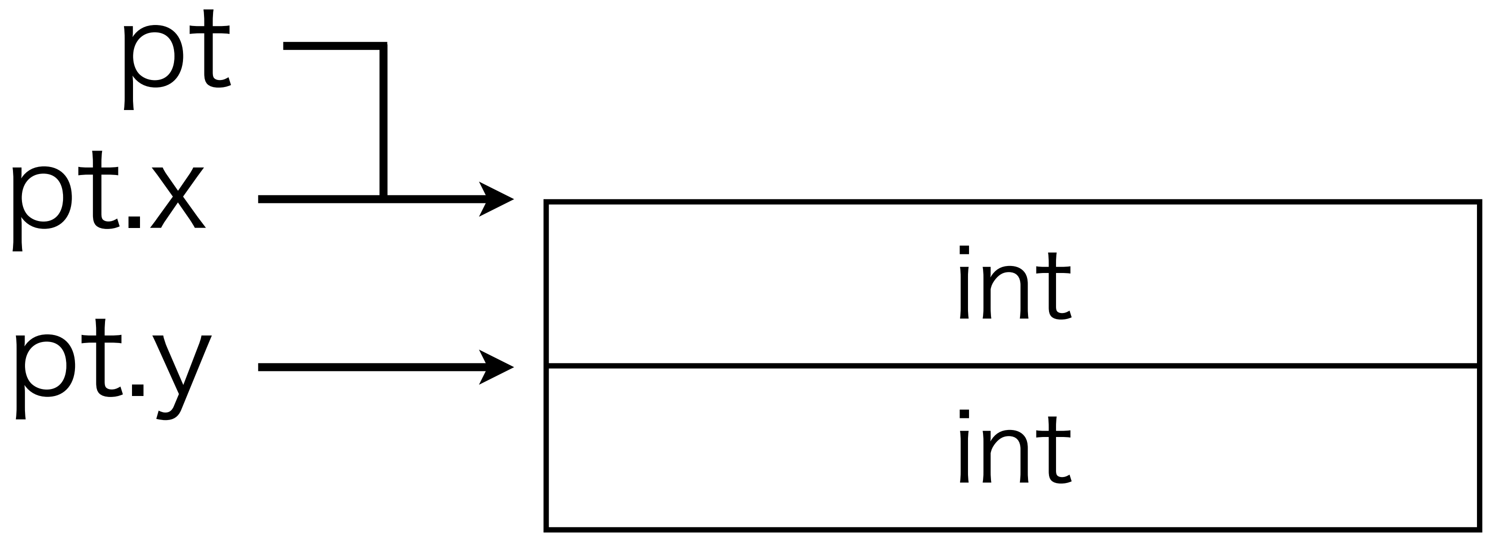
C++ Inheritance

Bits & Bobs

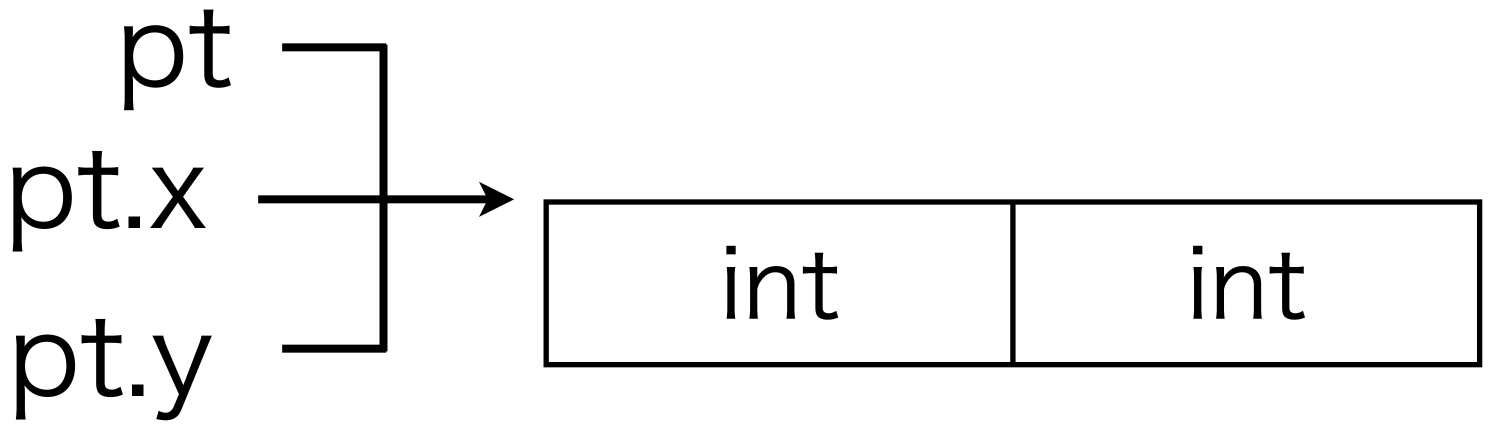
```
int x;
int y;
```



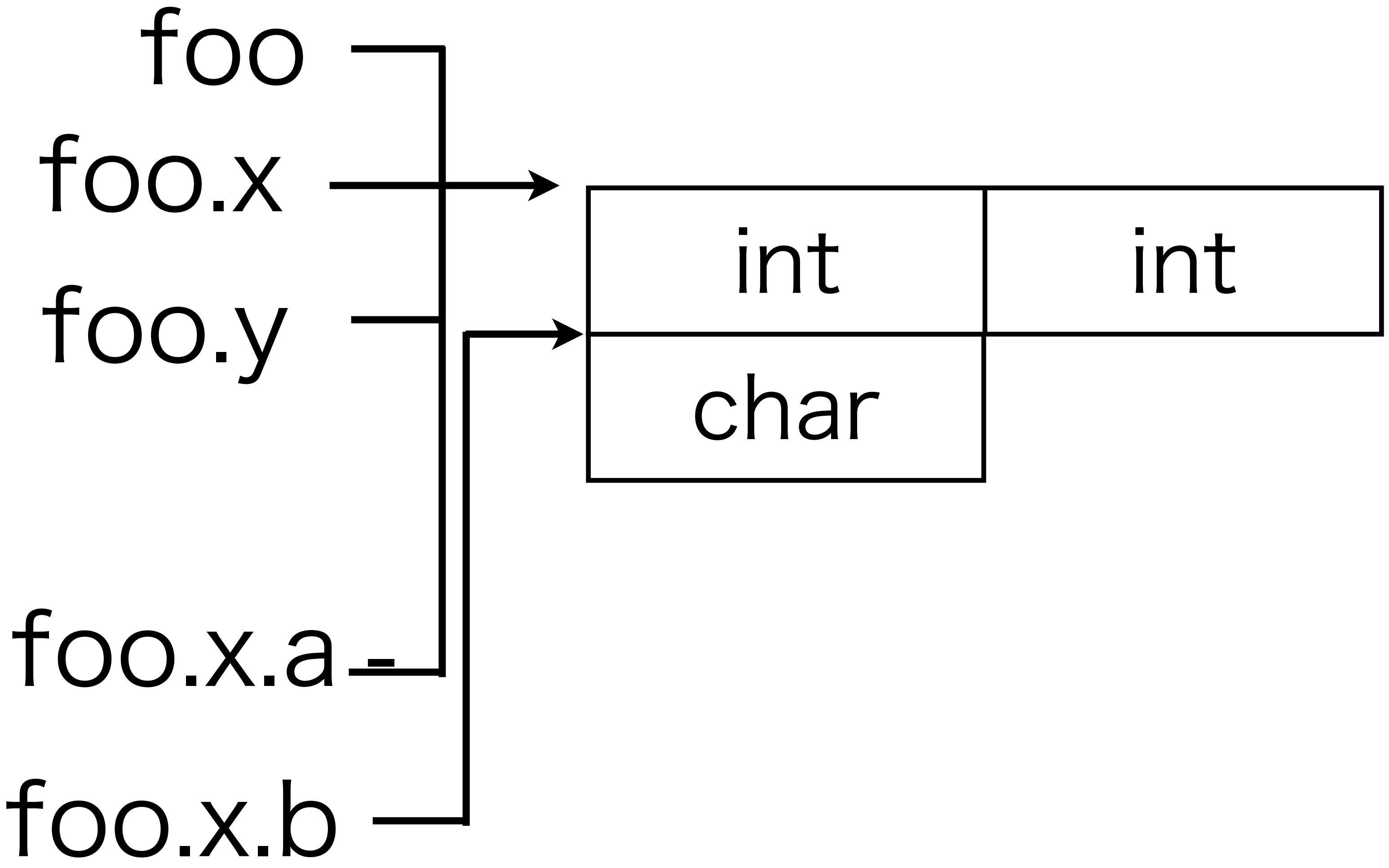
```
struct {
  int x;
  int y;
} pt;
```



```
union {
  int x;
  int y;
} pt;
```



```
union {
  struct {
    int a;
    char b;
  } x;
  int y;
} foo;
```


foo

foo.x

foo.y

foo.x.a

foo.x.b

int

int

char

Why Inheritance?

```
struct Circle {
  int x, y;
  int radius;
  void draw();
};
```

```
struct Square {
  int x, y;
  int width;
  void draw();
};
```

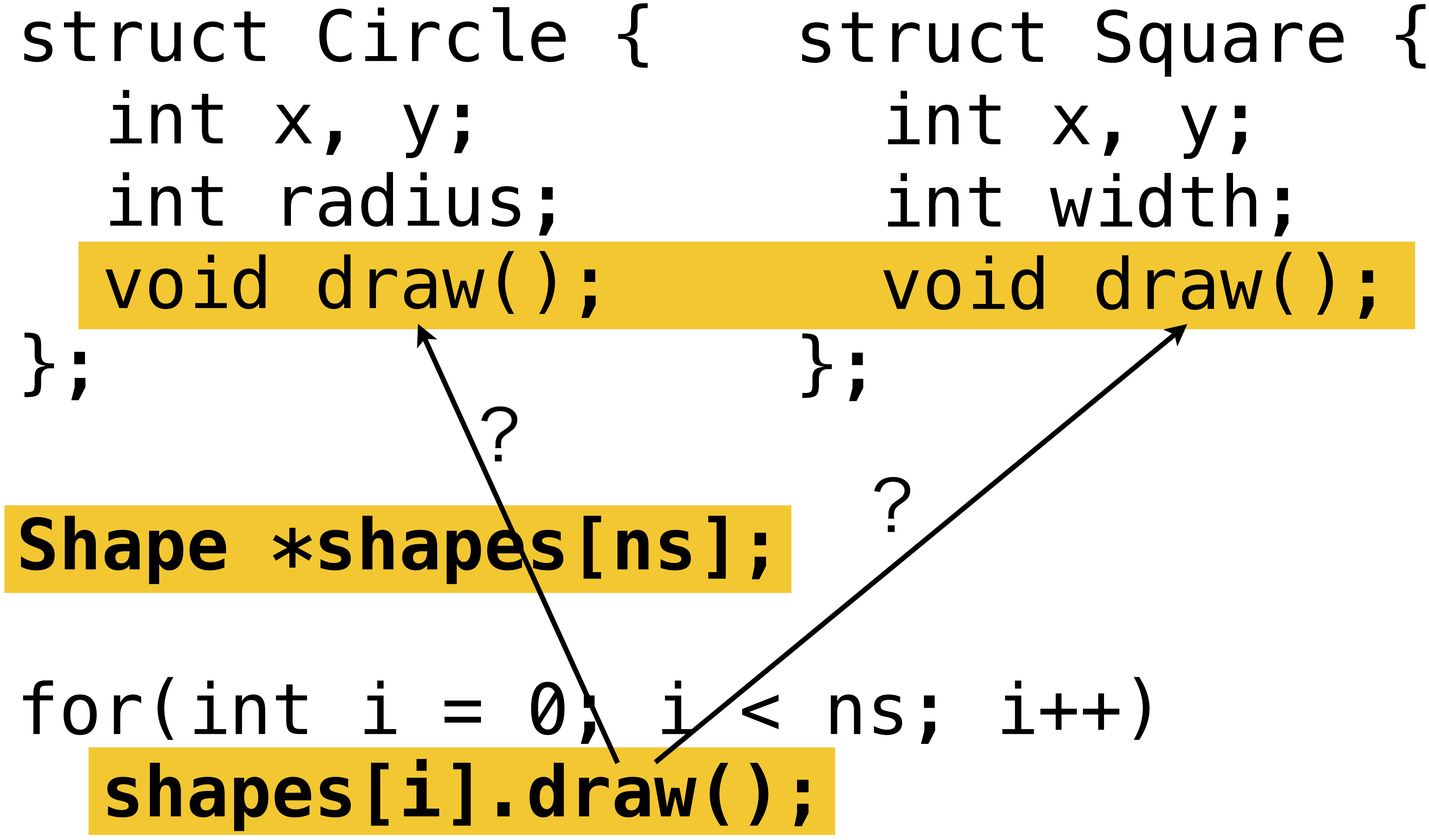
```
struct Circle {
  int x, y;
  int radius;
  void draw();
};
```

```
struct Square {
  int x, y;
  int width;
  void draw();
};
```

```
Circle *circles[nc];
Square *squares[ns];
```

```
struct Circle {   struct Square {
  int x, y;         int x, y;
  int radius;       int width;
  void draw();      void draw();
};                };

Circle *circles[nc];
Square *squares[ns];

for(int i = 0; i < nc; i++)
  circles[i].draw();
for(int i = 0; i < ns; i++)
  squares[i].draw();
```

```
Circle *circles[nc];
Square *squares[ns];

for(int i = 0; i < nc; i++)
  circles[i].draw();
for(int i = 0; i < ns; i++)
  squares[i].draw();

for(int i = 0; i < nc; i++)
  delete circles[i];
for(int i = 0; i < ns; i++)
  delete squares[i];

for(int i = 0; i < nc; i++)
  printf("%d\n", circles[i].width);
```



```
struct Circle {        struct Square {
  int x, y;              int x, y;
  int radius;            int width;
  void draw();           void draw();
};                     };

Shape *shapes[ns];

for(int i = 0; i < ns; i++)
  shapes[i].draw();

for(int i = 0; i < ns; i++)
  delete shapes[i];
```

Inheritance

```
class Shape {
public:
  virtual void draw() = 0;
};

class Circle : public Shape {
private:
  int x, y;
  int radius;
public:
  virtual void draw();
};

void Circle::draw() {
  ...
}
```

Best Practice

1) Subclassing

2) `virtual`

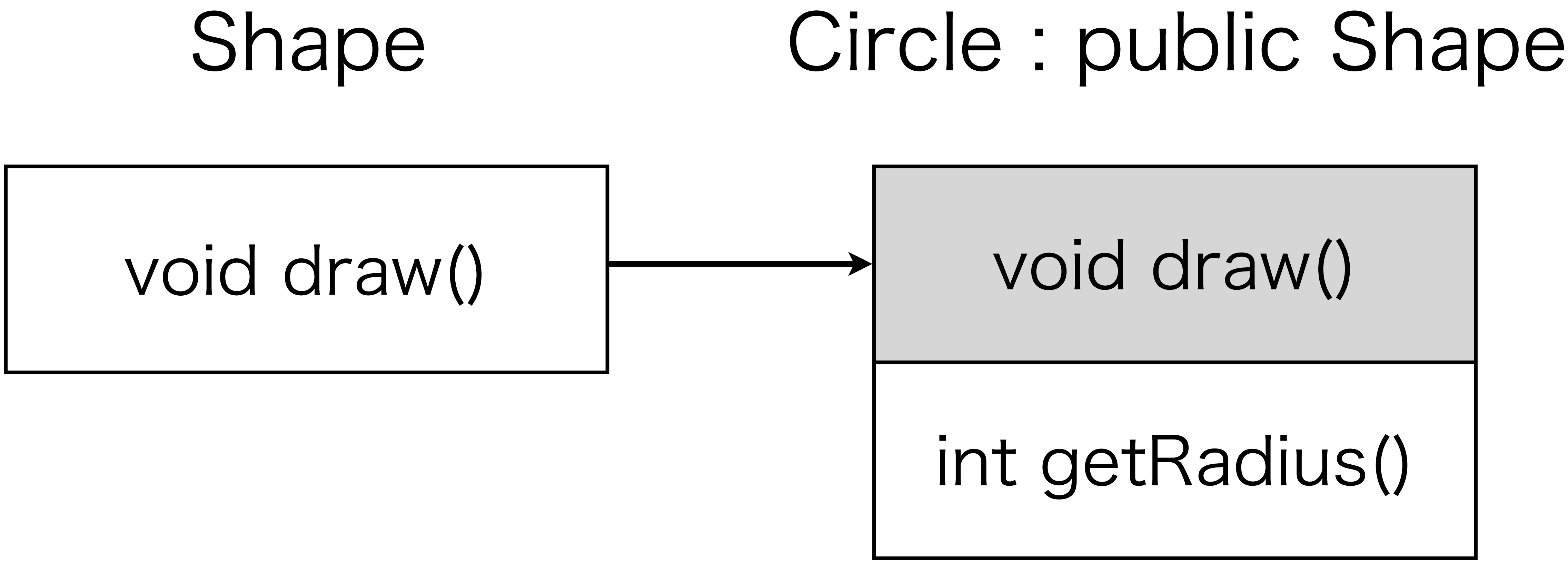
1) Subclassing

2) `virtual`

1) Subclassing

inherit behavior from the parent

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

class Circle : public Shape {
public:
  int getRadius();
};

int main() {
  Circle circle;
  circle.draw();
}
```



Shape

Circle : public Shape

void draw()

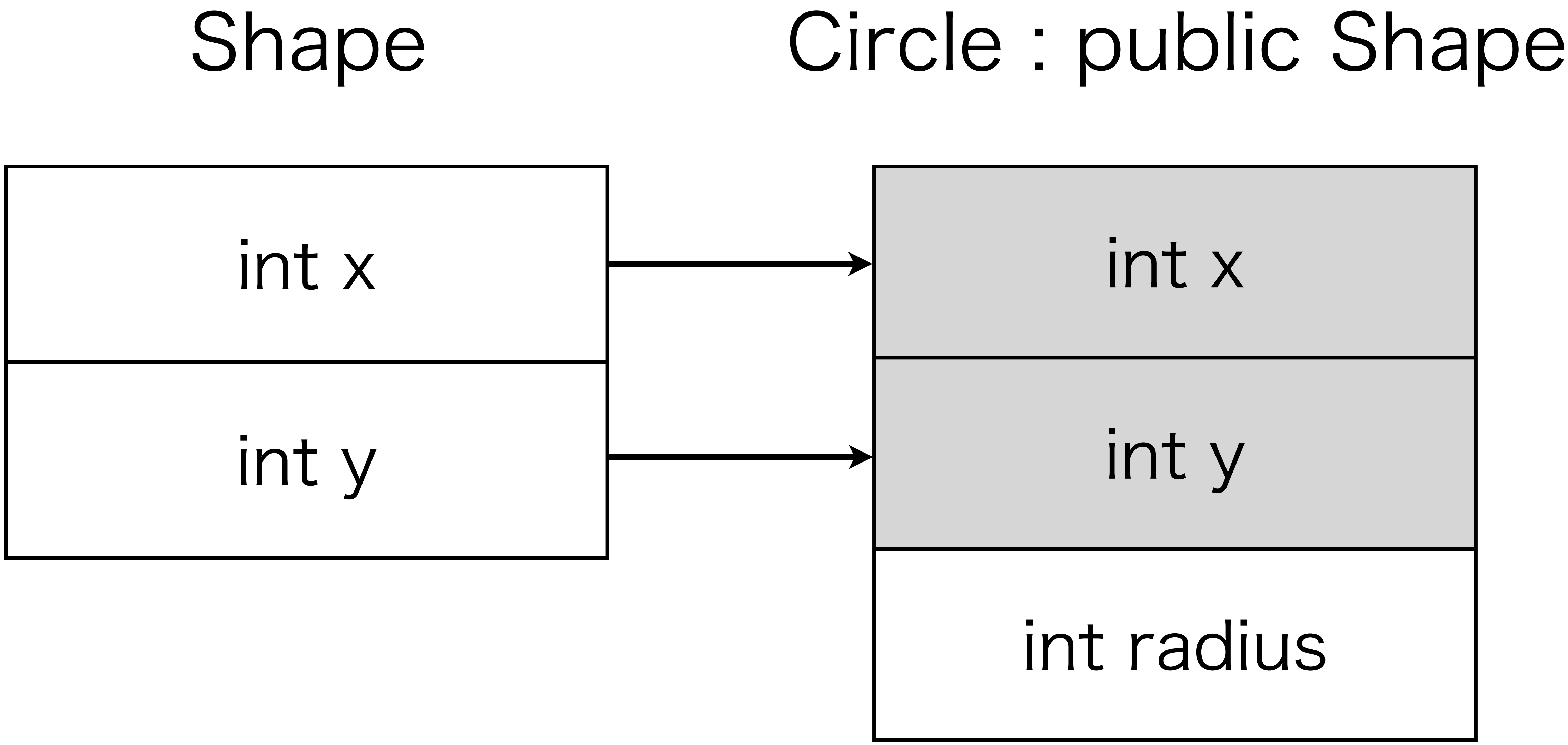
void draw()

int getRadius()

1) Subclassing

inherit fields from the parent

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

class Circle : public Shape {
public:
  int radius;
};

int main() {
  Circle circle;
  circle.x = 5;
}
```



Shape

int x

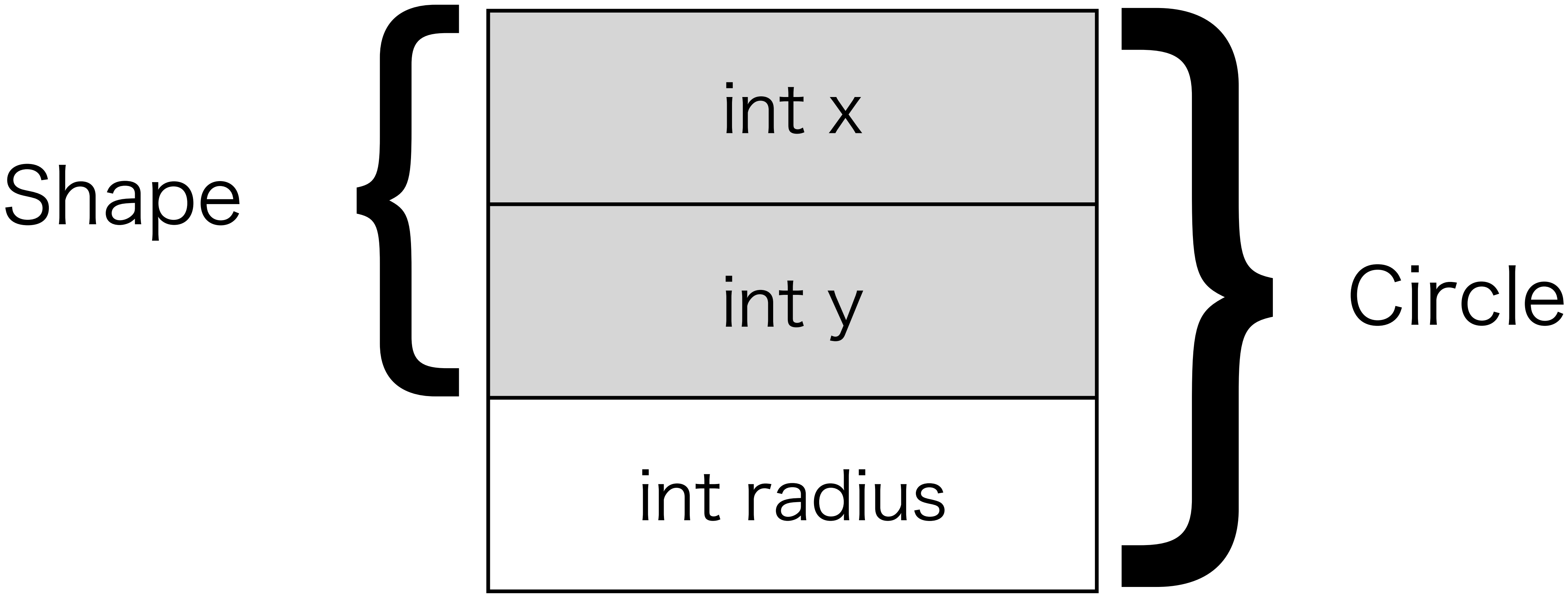
int y

Circle : public Shape

int x

int y

int radius



Shape

int x

int y

int radius

Circle

is-a or has-a?

```
class Circle
  : public Shape {
public:
  int radius;
};
```

```
class Circle {
public:
  Shape shape;
  int radius;
};
```

```
class Circle
  : public Shape {
public:
  int radius;
};

circle.x;
```

```
class Circle {
public:
  Shape shape;
  int radius;
};

circle.shape.x;
```

1) Subclassing

public/protected/private fields

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

only accessible in Shape class

```
void Circle::foo() {
  printf("%d", x);
  printf("%d", y); // compile error
}
```

```
class Shape {
public:
  int x;
protected:
  int y;
};
```

accessible in Shape class

and in subclasses

```
void Circle::foo() {
  printf("%d", x);
  printf("%d", y);
}

int main() {
  Circle circle;
  circle.x = 0; // compile error
}
```

1) Subclassing

public/protected/private inheritance

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

class Circle : public Shape {
};
```

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

class Circle : protected Shape {
};
```

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

class Circle : protected Shape {
protected:
  int getRadius();
};
```

The **inheritance** is protected.

If you can access **getRadius()**,

  you can access **draw()**

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

class Circle : private Shape {
private:
  int getRadius();
};
```

The **inheritance** is private.

If you can access **getRadius()**,
you can access **draw()**

private inheritance:
is-a or has-a

1) Subclassing

multiple inheritance

```
class Color {
public: virtual void print();
};

class Mood {
public: virtual void print();
};

class Blue : public Color, public Mood {
public:
  virtual void print() {
    this->Color::print();
    this->Mood::print();
  }
};
```

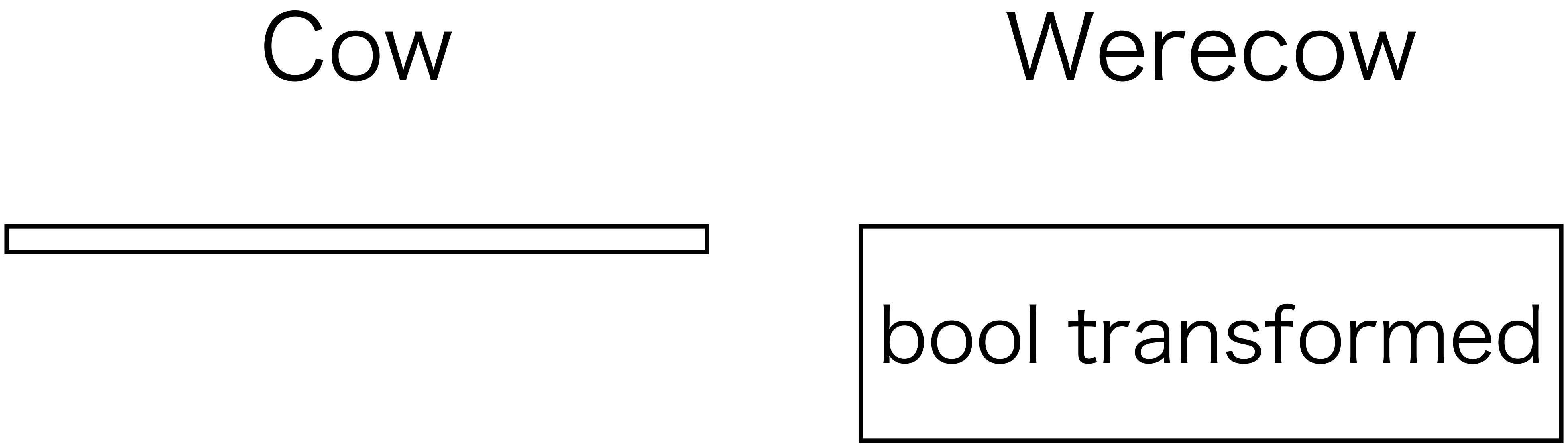
1) Subclassing

slicing

```
struct Cow {
  void speak() {
    printf("Moo.\n");
  }
};

struct Werecow : public Cow {
  bool transformed;
  void speak() {
    if (transformed)
      printf("Aaooooh!\n");
    else
      printf("Moo.\n");
  }
};
```

```
Werecow wcow;
wcow.transformed = true;

Cow cows[2];
cows[0] = Cow();
cows[1] = wcow;

for (int i = 0; i < 2; i++)
  cows[i].speak();
wcow.speak();

// Output:
//    Moo.
//    Moo.
//    Aaooooh!
```

```
void poke(Cow cow) {
  cow.speak();
}

Cow judy;
Werecow bev;
bev.transformed = true;

poke(judy);
poke(bev);
bev.speak();

// Output:
//    Moo.
//    Moo.
//    Aaooooh!
```



Cow

Werecow

bool transformed

Use pointers

Use `virtual`

1) Subclassing

2) `virtual`

```
class Shape {
public:
  void draw() { printf("shape\n"); }
};

class Circle : public Shape {
public:
  void draw() { printf("circle\n"); }
};
```

```
class Shape {
public:
  void draw() { printf("shape\n"); }
};

class Circle : public Shape {
public:
  void draw() { printf("circle\n"); }
};

Circle *circle = new Circle;
circle->draw(); // "circle"
```

```
class Shape {
public:
  void draw() { printf("shape\n"); }
};

class Circle : public Shape {
public:
  void draw() { printf("circle\n"); }
};

Shape *shape = new Circle;
shape->draw(); // "shape"
```

```
class Shape {
public:
  void draw() { printf("shape\n"); }
};

class Circle : public Shape {
public:
  void draw() { printf("circle\n"); }
};

Shape *shape = new Circle;
shape->draw(); // "shape"
```

draw() is **non-virtual**,

so it’s compiled like a C call

Non-virtual functions are -
determined at compile-time -

```
class Cat {
public:
  void yawn(int duration);
};

Cat cat, *pcat = new SuperCat;
cat.yawn(4);
pcat->yawn(4);
```

Both use Cat::yawn -

because both have type Cat -

Virtual functions are determined at run-time

```
class Cat {
public:
  virtual void yawn(int duration);
};

Cat cat, *pcat = new SuperCat;
cat.yawn(4);
pcat->yawn(4);
```

Use Cat::yawn and SuperCat::yawn

(pcat’s type is checked every time it’s called) non-virtual: compile-time

virtual: run-time

2) `virtual`

pure virtual methods

```
class Shape {
public:
  virtual void draw() = 0;
};

int main() {
  Shape shape;
}
```

```
pure.cpp: In function ‘int main()’:
pure.cpp:7: error: cannot declare variable ‘shape’ to be of abstract type ‘Shape’
pure.cpp:1: note:   because the following virtual functions are pure within ‘Shape’:
pure.cpp:3: note:   virtual void Shape::draw()
```

```cpp
class Drawable {
public:
  virtual void draw() = 0;
};

class Fish : public Drawable {
public:
  virtual void draw();
};

int main() {
  Drawable *drawables[3];
  drawables[0] = new Fish;
  drawables[1] = new Salami;
  drawables[2] = new JackSparrow;
}
```

2) `virtual`

destructors

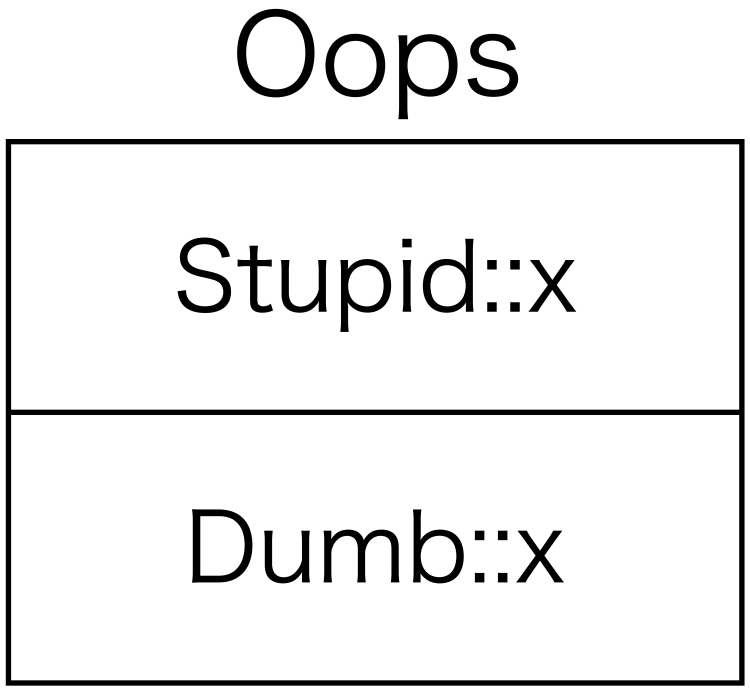
Make virtual destructors -

```
class Fish {
public:
  Fish() {
    gills[0] = new Gill;
    gills[1] = new Gill;
  }
  virtual ~Fish() {
    delete gills[0];
    delete gills[1];
  }
private:
  Gill *gills[2];
};
```

2) `virtual`

virtual inheritance

```
class Goofball {
  int x;
};

class Stupid : public Goofball { };
class Dumb : public Goofball { };

class Oops : public Stupid, public Dumb {
};
```



Oops

Stupid::x

Dumb::x

```
class Goofball {
  int x;
};

class Stupid : public Goofball { };
class Dumb : public Goofball { };

class Oops : public Stupid, public Dumb {
  int fail();
};

int Oops::fail() {
  Stupid::x = 1; Dumb::x = 2;
  return Stupid::x + Dumb::x; // 3
}
```

```
class Goofball {
  int x;
};

class Stupid : virtual public Goofball { };
class Dumb : virtual public Goofball { };

class Oops : public Stupid, public Dumb {
  int fail();
};

int Oops::fail() {
  Stupid::x = 1; Dumb::x = 2;
  return Stupid::x + Dumb::x; // 4
}
```

Conclusion

```
class Shape {
public:
  virtual void draw() = 0;
};

class Circle : public Shape {
private:
  int x, y;
  int radius;
public:
  virtual void draw();
};

void Circle::draw() {
  ...
}
```

Best Practice

6.S096 Introduction to C and C++
IAP 2013