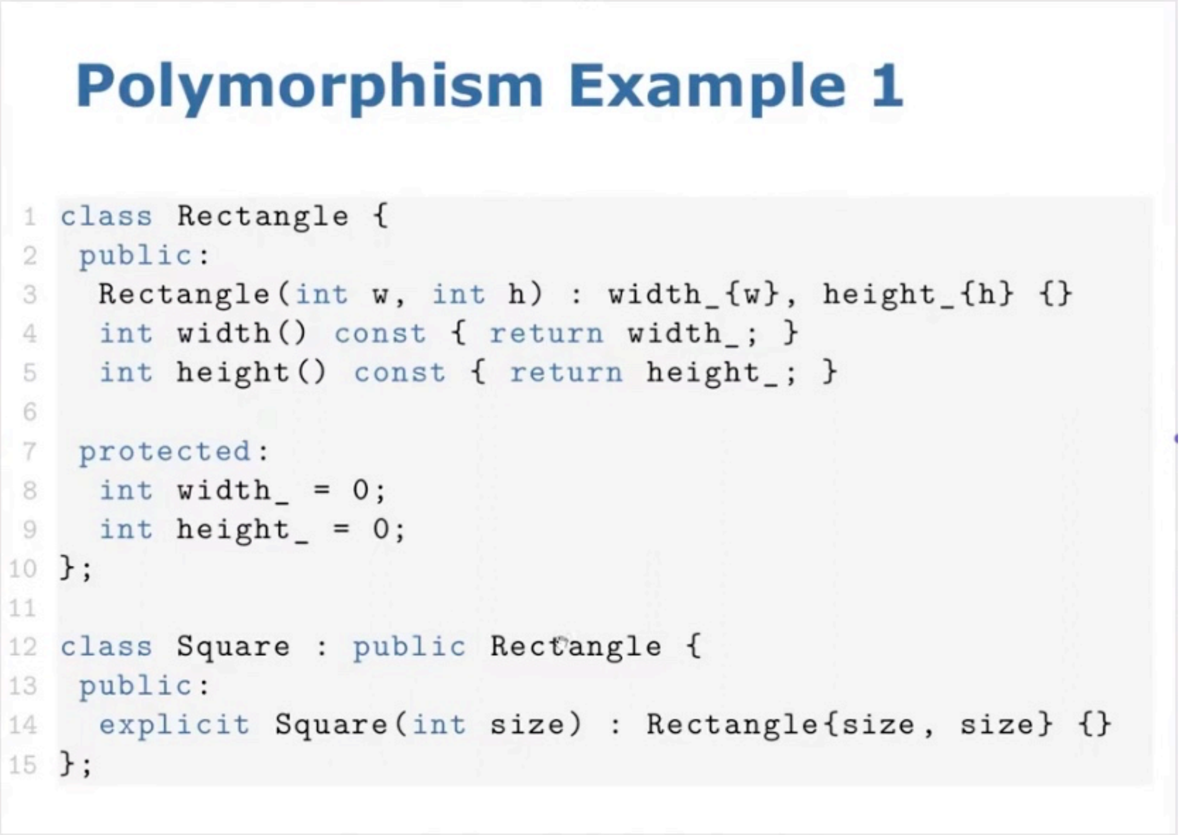
Polymorphism

- Polys means “many, much”
  morphe means “form, shape”
- Allow morphing derived classes into their base class type:
  const Base& base = Derived(…)

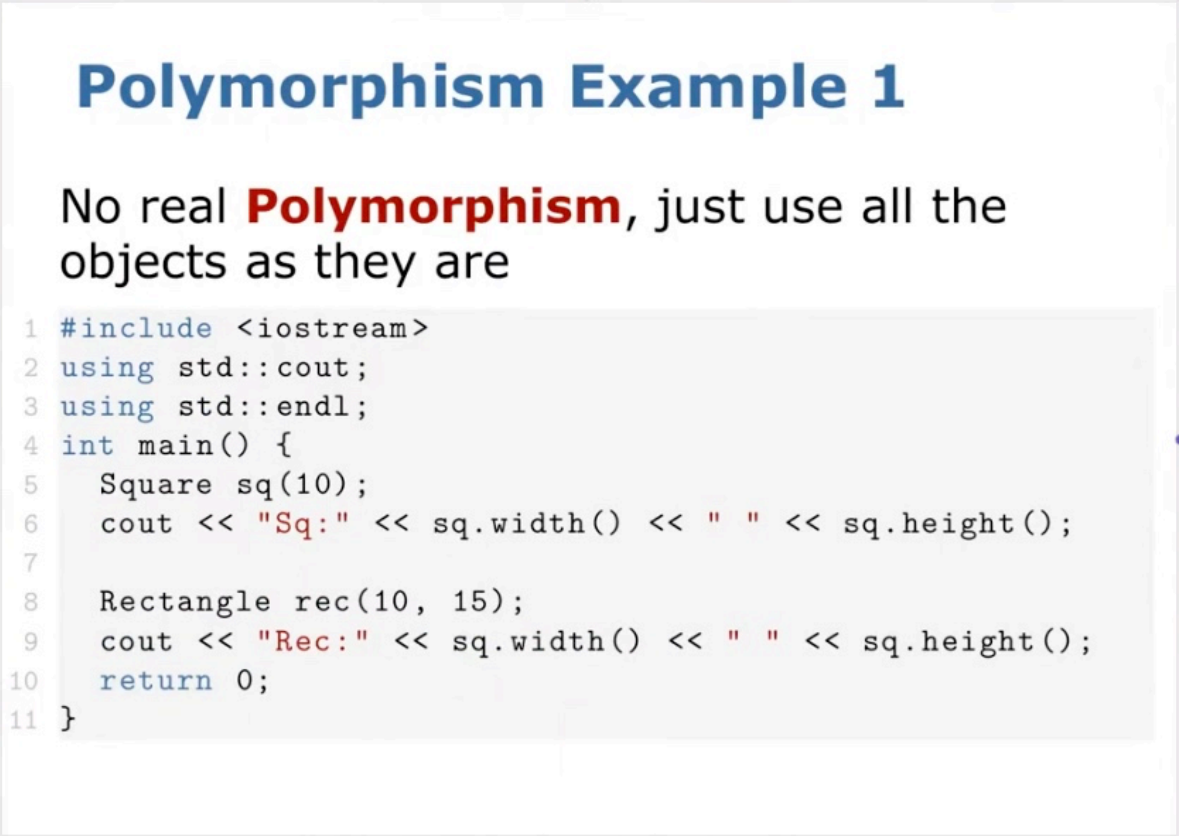


Polymorphism Example 1

```
class Rectangle {
 public:
  Rectangle(int w, int h) : width_{w}, height_{h} {}
  int width() const { return width_; }
  int height() const { return height_; }

 protected:
  int width_ = 0;
  int height_ = 0;
};

class Square : public Rectangle {
 public:
  explicit Square(int size) : Rectangle{size, size} {}
};
```

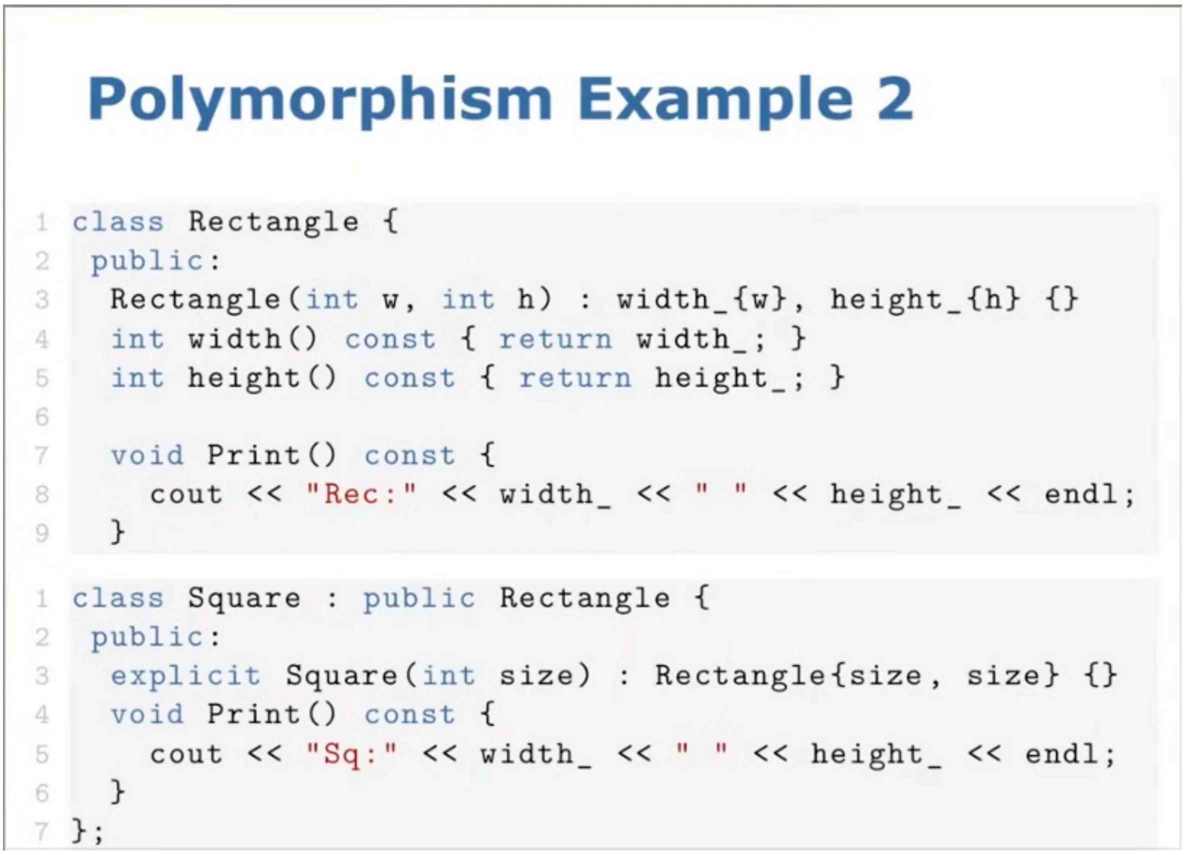


Polymorphism Example 1

No real **Polymorphism**, just use all the objects as they are

```cpp
#include <iostream>
using std::cout;
using std::endl;
int main() {
  Square sq(10);
  cout << "Sq:" << sq.width() << " " << sq.height();

  Rectangle rec(10, 15);
  cout << "Rec:" << sq.width() << " " << sq.height();
  return 0;
}
```

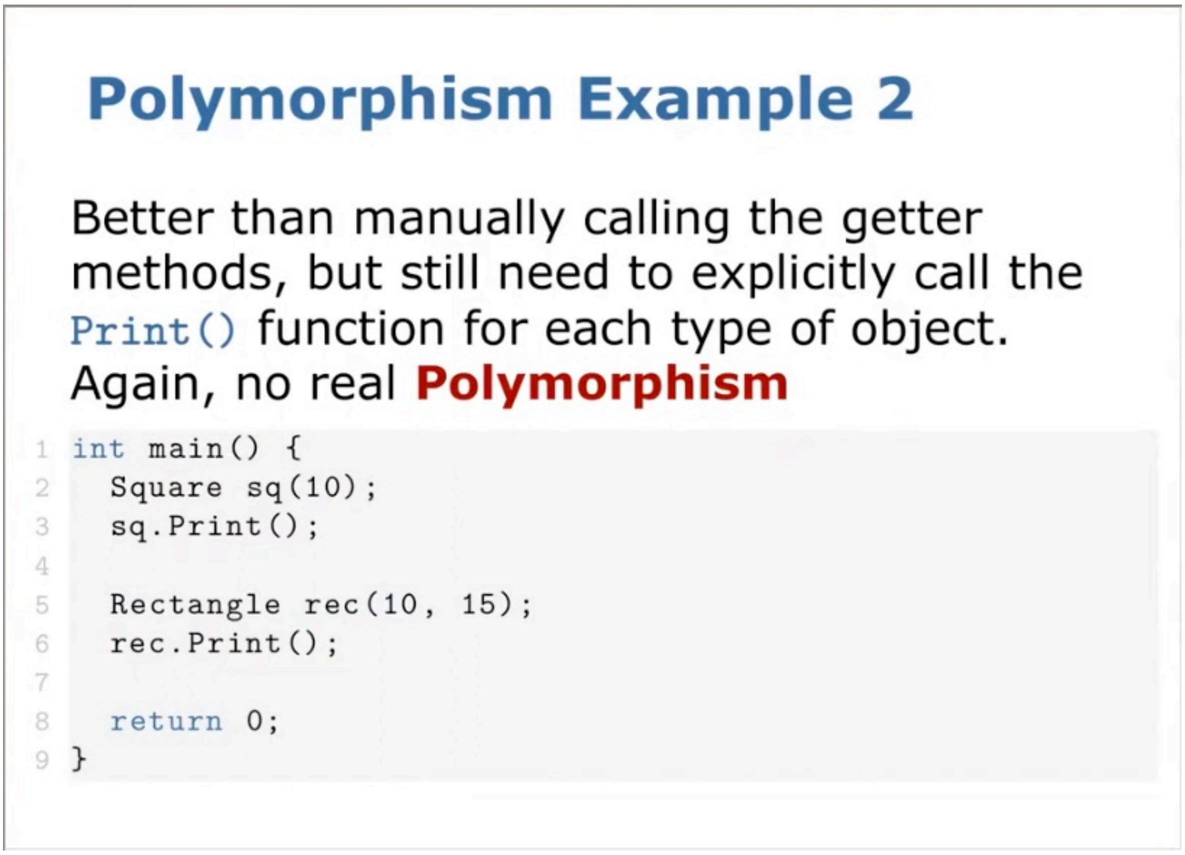


Polymorphism Example 2

```cpp
class Rectangle {
 public:
  Rectangle(int w, int h) : width_{w}, height_{h} {}
  int width() const { return width_; }
  int height() const { return height_; }

  void Print() const {
    cout << "Rec:" << width_ << " " << height_ << endl;
  }
```

```cpp
class Square : public Rectangle {
 public:
  explicit Square(int size) : Rectangle{size, size} {}
  void Print() const {
    cout << "Sq:" << width_ << " " << height_ << endl;
  }
};
```



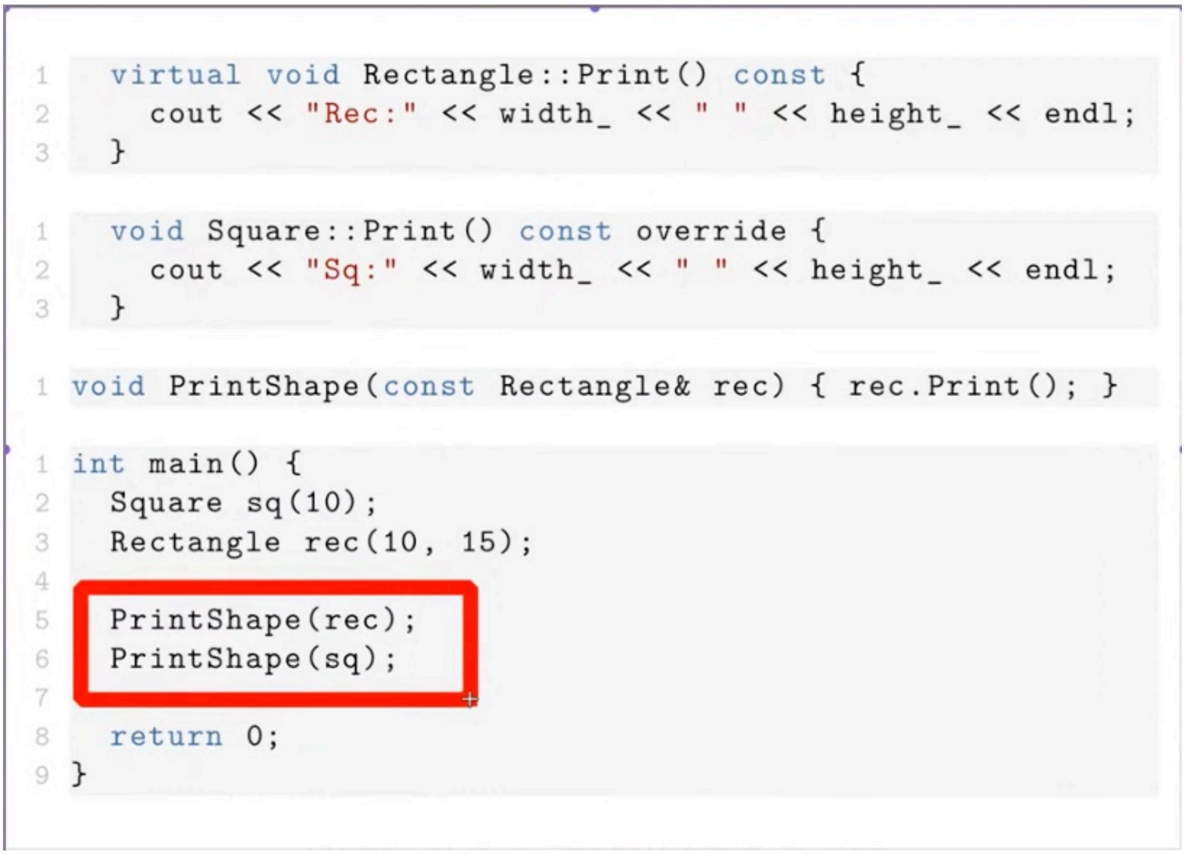
Polymorphism Example 2

Better than manually calling the getter methods, but still need to explicitly call the `Print()` function for each type of object. Again, no real **Polymorphism**

```cpp
int main() {
  Square sq(10);
  sq.Print();

  Rectangle rec(10, 15);
  rec.Print();

  return 0;
}
```



```cpp
  virtual void Rectangle::Print() const {
    cout << "Rec:" << width_ << " " << height_ << endl;
  }
```

```cpp
  void Square::Print() const override {
    cout << "Sq:" << width_ << " " << height_ << endl;
  }
```

```cpp
void PrintShape(const Rectangle& rec) { rec.Print(); }
```

```cpp
int main() {
  Square sq(10);
  Rectangle rec(10, 15);

  PrintShape(rec);
  PrintShape(sq);

  return 0;
}
```

Now we are using Runtime Polymorphism, we are printing shapes to the std::cost and deciding at runtime with type of shape it is.

When is it useful?

- Allow encapsulating the implementation inside a class only asking it to conform to a common interface.
- Often used for:
  - Working with all children of some Base class in unified manner.
  - Enforcing an interface in multiple classes to implement some functionality.
  - In strategy pattern, where some complex functionality is outsourced into separate classes and is passed to the object in a modular fashion.