POLYMORPHISM

Lecturer：陈笑沙

TABLE OF CONTENTS

9.1 POLYMORPHISM

The basic meaning of polymorphism:

9.1 POLYMORPHISM

The basic meaning of polymorphism:

The same content, in different contexts, expresses different meanings.

9.1 POLYMORPHISM

The basic meaning of polymorphism:

The same content, in different contexts, expresses different meanings.

```
int b = 0;
Object b = Object();
// Equal sign handles different types of data
// Can be considered a form of polymorphism
```

9.1 POLYMORPHISM

```
int max(int a, int b) {
  return a > b ? a : b;
}

int max(int a, int b, int c) {
  return max(max(a, b), c);
}

int x = max(3, 6);
int y = max(3, 6, 9);
```

9.1 POLYMORPHISM

```
int max(int a, int b) {
  return a > b ? a : b;
}

int max(int a, int b, int c) {
  return max(max(a, b), c);
}

int x = max(3, 6);
int y = max(3, 6, 9);
```

The same max, with different actual execution content, can also be considered a naive form of polymorphism.

9.1 POLYMORPHISM

```
Base *b1 = new Derived1();
Base *b2 = new Derived2();
b1->print();
b2->print();
```

9.1 POLYMORPHISM

```
Base *b1 = new Derived1();
Base *b2 = new Derived2();
b1->print();
b2->print();
```

Narrowly speaking, polymorphism in object-oriented programming design, the same pointer, different execution effects.

9.1 POLYMORPHISM

ASSIGNMENT COMPATIBILITY RULES

- In public inheritance, derived class objects can be used as base class objects in the following ways:

9.1 POLYMORPHISM

ASSIGNMENT COMPATIBILITY RULES

- In public inheritance, derived class objects can be used as base class objects in the following ways:
  - Derived class objects can be directly assigned to base class objects

9.1 POLYMORPHISM

ASSIGNMENT COMPATIBILITY RULES

- In public inheritance, derived class objects can be used as base class objects in the following ways:
  - Derived class objects can be directly assigned to base class objects
  - Base class object references can reference a derived class object

9.1 POLYMORPHISM

ASSIGNMENT COMPATIBILITY RULES

- In public inheritance, derived class objects can be used as base class objects in the following ways:
    - Derived class objects can be directly assigned to base class objects
    - Base class object references can reference a derived class object
    - Base class object pointers can point to a derived class object

9.1 POLYMORPHISM

ASSIGNMENT COMPATIBILITY RULES

- In public inheritance, derived class objects can be used as base class objects in the following ways:
  - Derived class objects can be directly assigned to base class objects
  - Base class object references can reference a derived class object
  - Base class object pointers can point to a derived class object
- This rule can be simply understood as: All dogs are animals. But not all animals are dogs — all child class objects are base class objects.

9.1 POLYMORPHISM

```cpp
#include <iostream>

using std::cout;
using std::endl;

class Animal {
public:
  void bark() const {
    cout << "Some animal is barking!" << endl;
  }
};

class Dog : public Animal {
public:
  void bark() const {
```

9.1 POLYMORPHISM

```cpp
#include <iostream>

using std::cout;
using std::endl;

class Animal {
public:
  void bark() const {
    cout << "Some animal is barking!" << endl;
  }
};

class Dog : public Animal {
public:
  void bark() const {
```

9.1 POLYMORPHISM

```cpp
  void bark() const {
    cout << "Some animal is barking!" << endl;
  }
};

class Dog : public Animal {
public:
  void bark() const {
    cout << "Wang wang!" << endl;
  }
};

int main(int argc, char** argv) {
  Animal *animal = new Dog();
  animal->bark();
  void bark() const {
```

9.1 POLYMORPHISM

```
  }
};

class Dog : public Animal {
public:
  void bark() const {
    cout << "Wang wang!" << endl;
  }
};

int main(int argc, char** argv) {
  Animal *animal = new Dog();
  animal->bark();
  return 0;
}
  void bark() const {
```

9.1 POLYMORPHISM

9.1 POLYMORPHISM

- Through base class references or pointers, what you can see is a base class object

9.1 POLYMORPHISM

- Through base class references or pointers, what you can see is a base class object
- Members of the derived class are "invisible" to base class references or pointers

9.1 POLYMORPHISM

- Through base class references or pointers, what you can see is a base class object
- Members of the derived class are "invisible" to base class references or pointers
- We can use the virtual function mechanism of C++ to declare the relevant functions of the base class as virtual functions

9.1 POLYMORPHISM

- Through base class references or pointers, what you can see is a base class object
- Members of the derived class are "invisible" to base class references or pointers
- We can use the virtual function mechanism of C++ to declare the relevant functions of the base class as virtual functions
- This allows you to access the relevant functions in the derived class through base class references or pointers.

9.1 POLYMORPHISM

- Different objects in the class hierarchy can exhibit different behaviors for the same operation
- Once the class hierarchy is included in unified processing, the polymorphic behavior in the class hierarchy cannot be demonstrated

9.1 POLYMORPHISM

- Overloading ordinary member functions in two ways:
  - Overloading within the same class: Overloaded functions are distinguished by parameter characteristics.
  - Derived classes overload member functions of the base class.
- Since overloaded functions are in different classes, their prototypes can be completely identical. When calling, use the "ClassName::FunctionName" method to distinguish.
- The matching of the above two overloads is statically completed at compile time.

9.1 POLYMORPHISM

- Overloading is a simple form of polymorphism.
- C++ provides another more flexible polymorphic mechanism: virtual functions.
- Virtual functions allow the matching of function calls and function bodies to be determined at runtime.
- Virtual functions provide a dynamic binding mechanism.

9.2 HOW POLYMORPHISM WORKS

VIRTUAL FUNCTIONS

- Member functions declared with the virtual keyword in the base class are called virtual functions.
- Virtual functions can be redefined in one or more derived classes, but the prototype of the virtual function (including return value type, function name, and parameter list) must be exactly the same when redefined. Otherwise, the function will lose its virtual nature.

9.2 HOW POLYMORPHISM WORKS

VIRTUAL FUNCTIONS

- Declare functions as virtual functions using virtual in the base class.
- Overload the virtual function consistently in the public derived class.
- Define a base class reference or pointer to refer to or point to a derived class object.
- When calling a virtual function through this reference or pointer, the function will exhibit its virtual nature.

9.2 HOW POLYMORPHISM WORKS

VIRTUAL FUNCTIONS

In C++, the base class must indicate which functions it expects derived classes to redefine. Functions declared as virtual are those that the base class expects derived classes to redefine, while functions that the base class expects derived classes to inherit should not be declared as virtual.

9.2 HOW POLYMORPHISM WORKS

Example project:

/example/lec09/particle

9.3 IMPROPER VIRTUAL FUNCTIONS

Improper Virtual Functions

```
#include <iostream>
using namespace std;
class Base {
public:
  virtual void fn(int x) {
    cout << "In Base class, int x = " << x << endl;
  } // Base class virtual function
};
class SubClass : public Base {
public:
  // Subclass virtual function,
  // not a virtual function passed to the subclass
  virtual void fn(float x) {
    cout << "In SubClass, float x = " << x << endl;
  }
```

9.3 IMPROPER VIRTUAL FUNCTIONS

Improper Virtual Functions

```
#include <iostream>
using namespace std;
class Base {
public:
  virtual void fn(int x) {
    cout << "In Base class, int x = " << x << endl;
  } // Base class virtual function
};
class SubClass : public Base {
public:
  // Subclass virtual function,
  // not a virtual function passed to the subclass
  virtual void fn(float x) {
    cout << "In SubClass, float x = " << x << endl;
  }
```

9.3 IMPROPER VIRTUAL FUNCTIONS

Improper Virtual Functions

```cpp
  virtual void fn(int x) {
    cout << "In Base class, int x = " << x << endl;
  } // Base class virtual function
};
class SubClass : public Base {
public:
  // Subclass virtual function,
  // not a virtual function passed to the subclass
  virtual void fn(float x) {
    cout << "In SubClass, float x = " << x << endl;
  }
};

// When b refers to a base class object,
// fn is a base class function,
// and since there is no virtual function passed
```

9.3 IMPROPER VIRTUAL FUNCTIONS

Improper Virtual Functions

```cpp
};

// When b refers to a base class object,
// fn is a base class function,
// and since there is no virtual function passed
// to the subclass, polymorphism is not displayed
void test(Base &b) {
  // When b refers to a subclass object,
  // all visible functions are base class functions
  b.fn(2);
  // When b refers to a base class object,
  // all visible functions are base class functions
  b.fn(2.2f);
}
int main() {
```

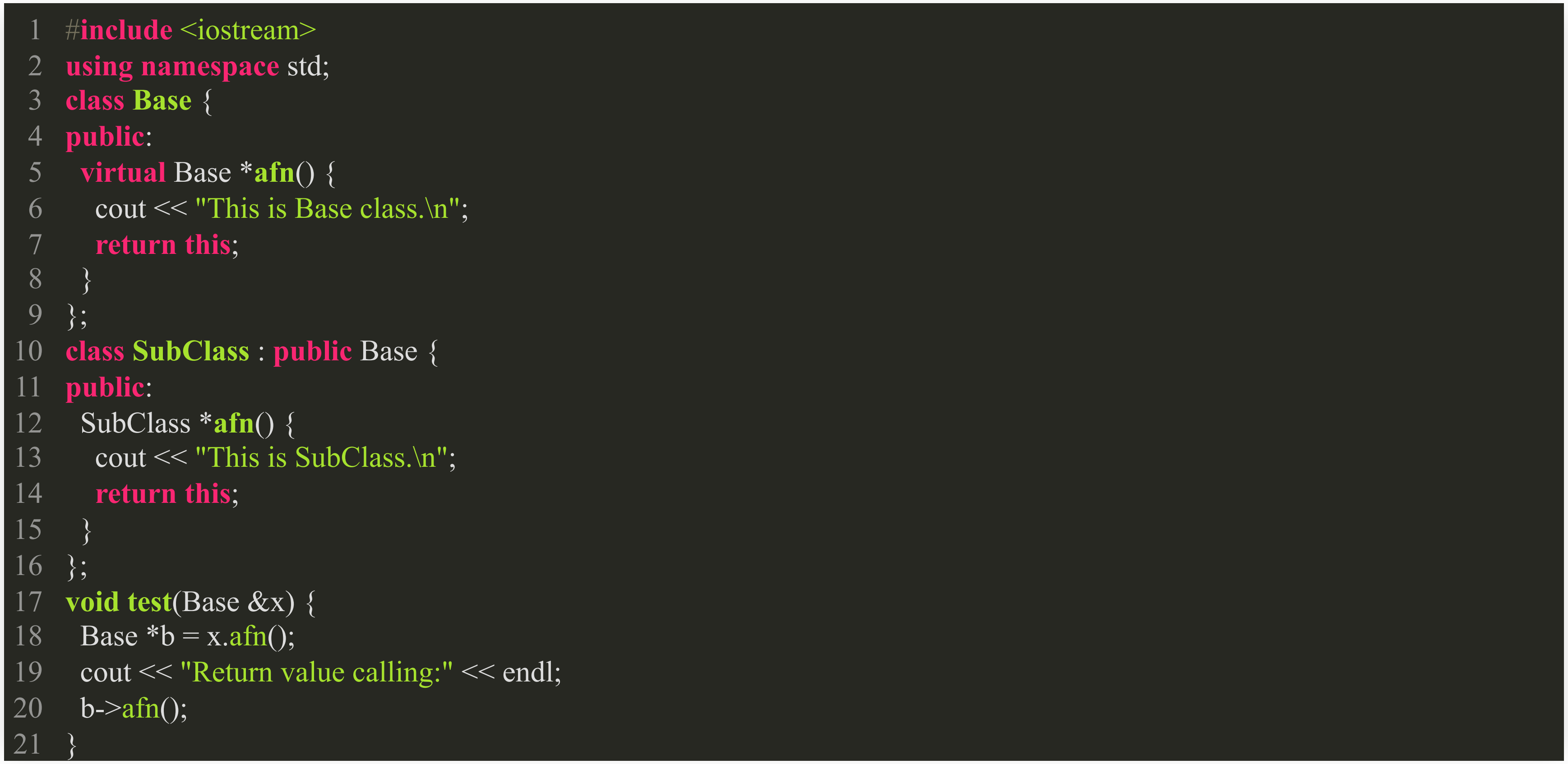
9.3 IMPROPER VIRTUAL FUNCTIONS

Improper Virtual Functions

```
  // When b refers to a subclass object,
  // all visible functions are base class functions
  b.fn(2);
  // When b refers to a base class object,
  // all visible functions are base class functions
  b.fn(2.2f);
}
int main() {
  cout << "Calling test(bc)\n";
  Base b;
  test(b); // Passing a base class object
  cout << "Calling test(sc)\n";
  SubClass sc;
  test(sc); // Passing a subclass object
}
```

9.4 RESTRICTIONS ON VIRTUAL FUNCTIONS

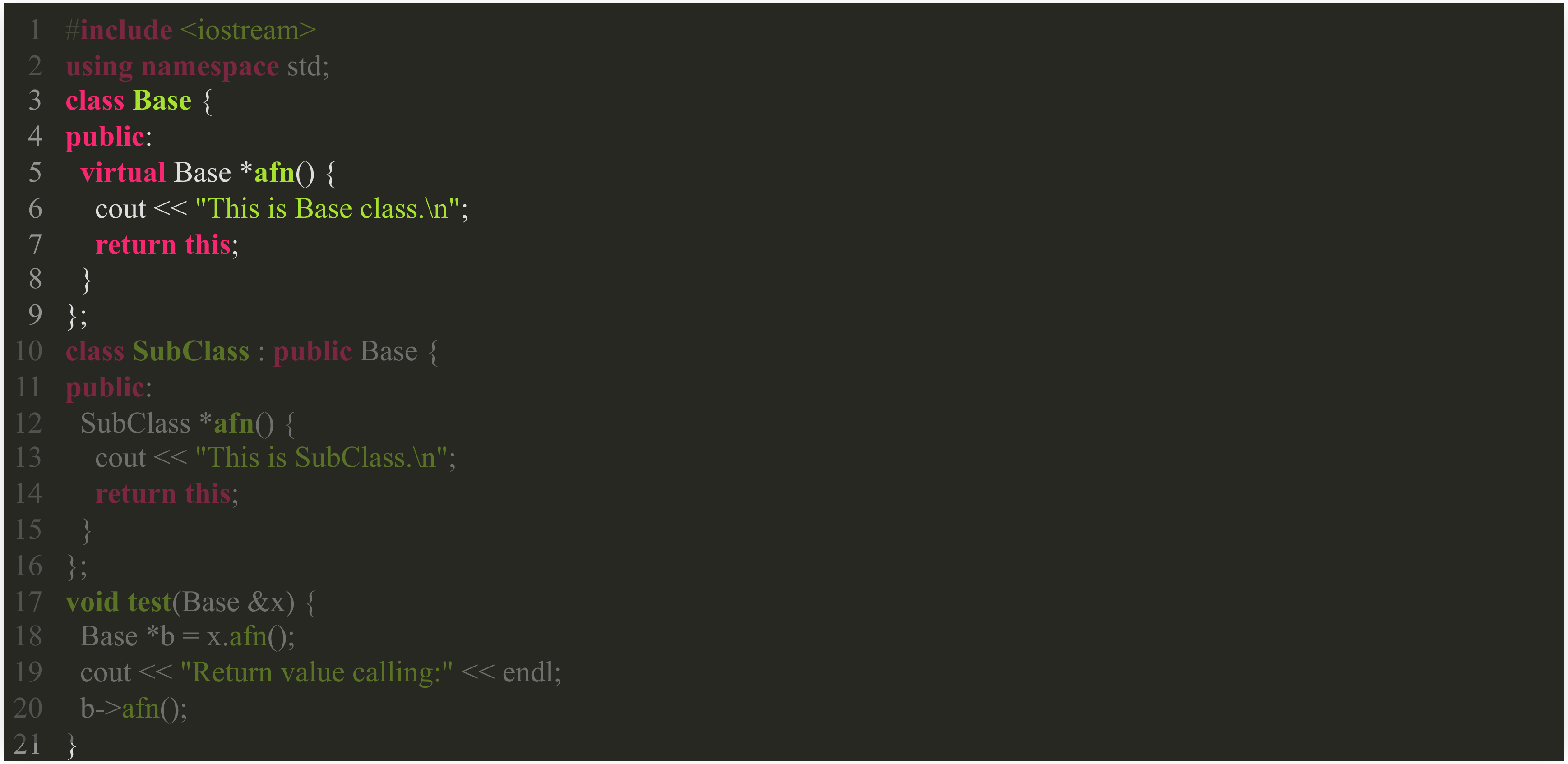
If the return type is the class itself, the subclass can return a subclass entity, and the virtual function continues to pass to the subclass



```cpp
#include <iostream>
using namespace std;
class Base {
public:
  virtual Base *afn() {
    cout << "This is Base class.\n";
    return this;
  }
};
class SubClass : public Base {
public:
  SubClass *afn() {
    cout << "This is SubClass.\n";
    return this;
  }
};
void test(Base &x) {
  Base *b = x.afn();
  cout << "Return value calling:" << endl;
  b->afn();
}
```

9.4 RESTRICTIONS ON VIRTUAL FUNCTIONS

If the return type is the class itself, the subclass can return a subclass entity, and the virtual function continues to pass to the subclass



```cpp
#include <iostream>
using namespace std;
class Base {
public:
  virtual Base *afn() {
    cout << "This is Base class.\n";
    return this;
  }
};
class SubClass : public Base {
public:
  SubClass *afn() {
    cout << "This is SubClass.\n";
    return this;
  }
};
void test(Base &x) {
  Base *b = x.afn();
  cout << "Return value calling:" << endl;
  b->afn();
}
```

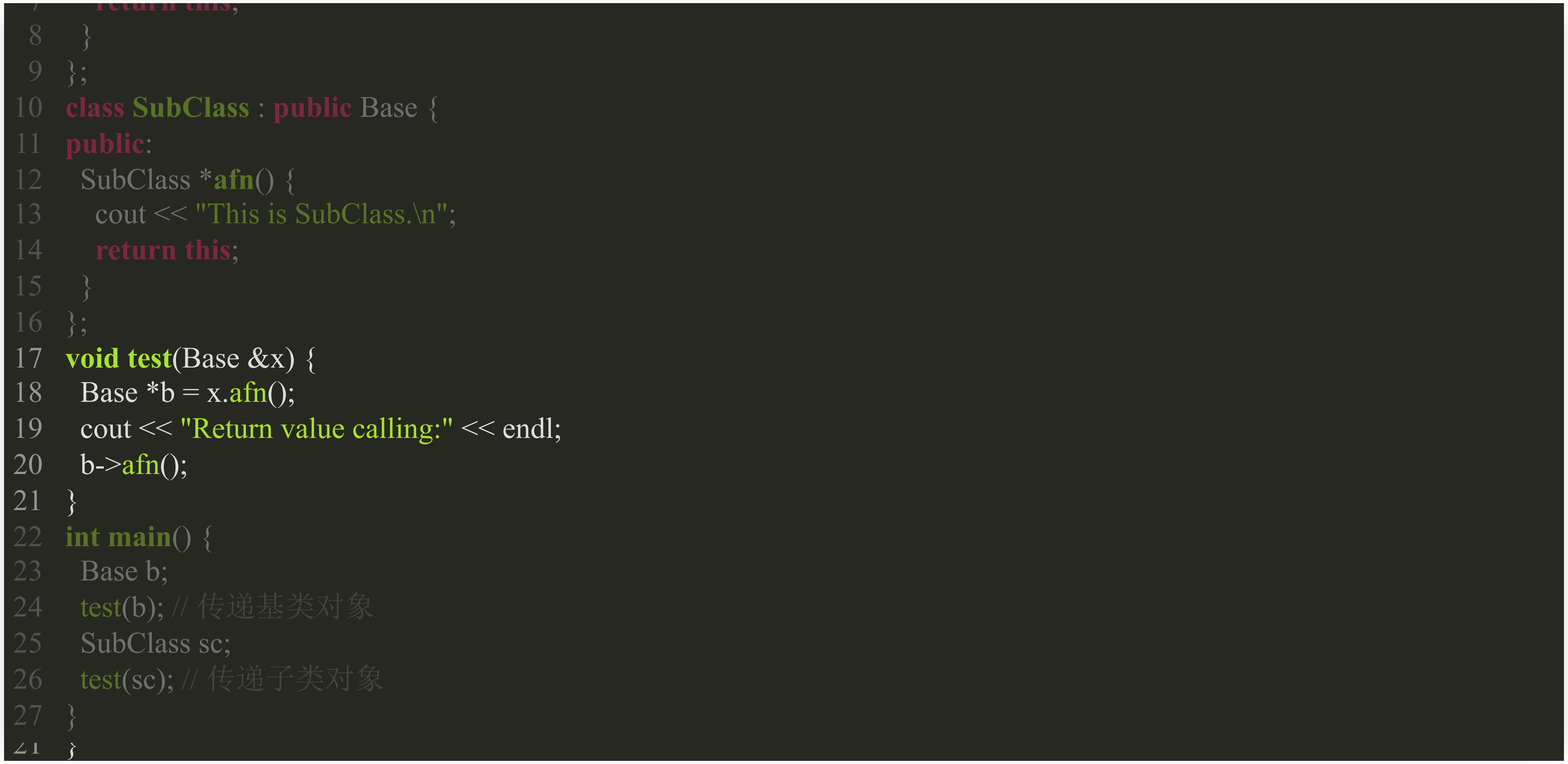
9.4 RESTRICTIONS ON VIRTUAL FUNCTIONS

If the return type is the class itself, the subclass can return a subclass entity, and the virtual function continues to pass to the subclass

```cpp
class Base {
public:
  virtual Base *afn() {
    cout << "This is Base class.\n";
    return this;
  }
};
class SubClass : public Base {
public:
  SubClass *afn() {
    cout << "This is SubClass.\n";
    return this;
  }
};
void test(Base &x) {
  Base *b = x.afn();
  cout << "Return value calling:" << endl;
  b->afn();
}
int main() {
  Base b;
}
```

9.4 RESTRICTIONS ON VIRTUAL FUNCTIONS

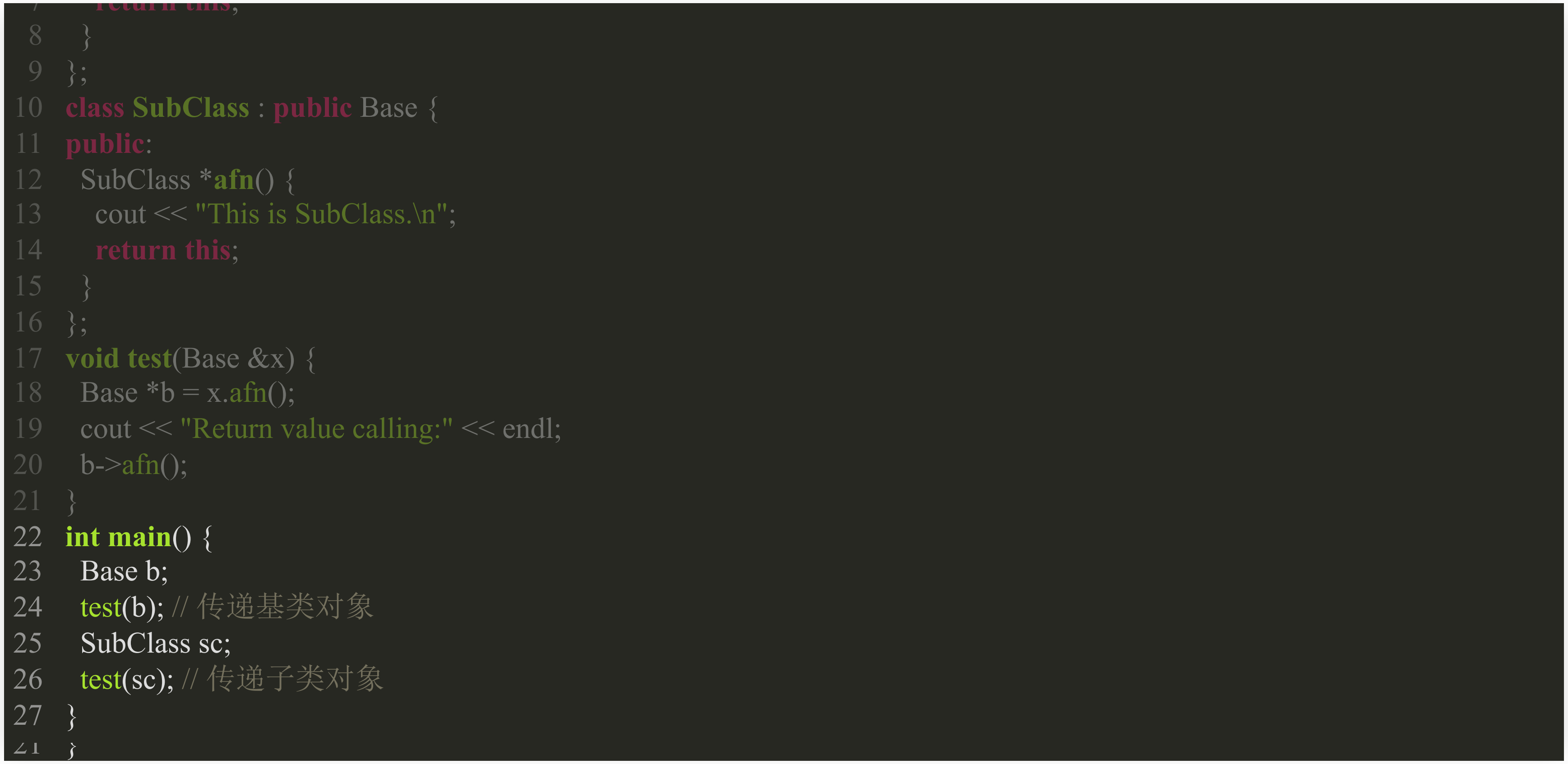
If the return type is the class itself, the subclass can return a subclass entity, and the virtual function continues to pass to the subclass



```
    return this;
  }
};
class SubClass : public Base {
public:
  SubClass *afn() {
    cout << "This is SubClass.\n";
    return this;
  }
};
void test(Base &x) {
  Base *b = x.afn();
  cout << "Return value calling:" << endl;
  b->afn();
}
int main() {
  Base b;
  test(b); // 传递基类对象
  SubClass sc;
  test(sc); // 传递子类对象
}
}
```

9.4 RESTRICTIONS ON VIRTUAL FUNCTIONS

If the return type is the class itself, the subclass can return a subclass entity, and the virtual function continues to pass to the subclass



```
  }
};
class SubClass : public Base {
public:
  SubClass *afn() {
    cout << "This is SubClass.\n";
    return this;
  }
};
void test(Base &x) {
  Base *b = x.afn();
  cout << "Return value calling:" << endl;
  b->afn();
}
int main() {
  Base b;
  test(b); // 传递基类对象
  SubClass sc;
  test(sc); // 传递子类对象
}
```

9.5 ABSTRACT CLASSES AND PURE VIRTUAL FUNCTIONS

- Only member functions can be declared as virtual functions
- Virtual functions cannot be declared as global functions
- Virtual functions cannot be declared as friend functions
- Static member functions cannot be virtual functions
- Inline functions cannot be virtual functions
- Constructors cannot be virtual functions (destructors are usually virtual functions)

9.5 ABSTRACT CLASSES AND PURE VIRTUAL FUNCTIONS

9.5 ABSTRACT CLASSES AND PURE VIRTUAL FUNCTIONS

- The base class represents an abstract concept, providing some common interfaces representing the common operations that objects of this class have.

9.5 ABSTRACT CLASSES AND PURE VIRTUAL FUNCTIONS

- The base class represents an abstract concept, providing some common interfaces representing the common operations that objects of this class have.
- These common interfaces in the base class only need to be declared without implementation, i.e., pure virtual functions.

9.5 ABSTRACT CLASSES AND PURE VIRTUAL FUNCTIONS

- The base class represents an abstract concept, providing some common interfaces representing the common operations that objects of this class have.
- These common interfaces in the base class only need to be declared without implementation, i.e., pure virtual functions.
- Pure virtual functions outline the protocols that derived classes should follow, with the specific implementations determined by the derived classes.

9.5 ABSTRACT CLASSES AND PURE VIRTUAL FUNCTIONS

9.5 ABSTRACT CLASSES AND PURE VIRTUAL FUNCTIONS

- If a class contains pure virtual functions (possibly more than one), then the class is an abstract class

9.5 ABSTRACT CLASSES AND PURE VIRTUAL FUNCTIONS

- If a class contains pure virtual functions (possibly more than one), then the class is an abstract class
- If a class is an abstract class, it must contain pure virtual functions

9.5 ABSTRACT CLASSES AND PURE VIRTUAL FUNCTIONS

- If a class contains pure virtual functions (possibly more than one), then the class is an abstract class
- If a class is an abstract class, it must contain pure virtual functions
- Abstract classes cannot create objects

9.5 ABSTRACT CLASSES AND PURE VIRTUAL FUNCTIONS

- If a class contains pure virtual functions (possibly more than one), then the class is an abstract class
- If a class is an abstract class, it must contain pure virtual functions
- Abstract classes cannot create objects
- If the subclass of an abstract class has all its virtual functions defined (i.e., no longer has pure virtual functions), it becomes a concrete class

9.5 ABSTRACT CLASSES AND PURE VIRTUAL FUNCTIONS

Example: Change Shape in the particle project to an abstract class.

HOMEWORK

- Create a Shape class, which only includes double getArea() and double getPerimeter() pure virtual functions.
- Create three sub-classes of Shape: Rectangle, Circle and Triangle.
- Implement getArea and getPerimeter for these sub classes.
- Create a std::vector that contains Shape objects.
- Put some subclasses into that vector (should include all shape types).
- Print the area and perimeter of each shape in that vector.

1. If a triangle's side lengths are a, b, c, then its area is: $\sqrt{p(p-a)(p-b)(p-c)}$, where $p = (a+b+c)/2$
2. Submit your homework in website: https://pdcxs.github.io/homework