多态

主讲：陈笑沙

目录

9.1 多态性

多态的基本含义：

9.1 多态性

多态的基本含义：

同样的内容，在不同的上下文中，表达不同的含义。

9.1 多态性

多态的基本含义：

同样的内容，在不同的上下文中，表达不同的含义。

```
int b = 0;
Object b = Object();
// 等号处理不同类型数据
// 可以认为是一种多态
```

9.1 多态性

```
 1 int max(int a, int b) {
 2   return a > b ? a : b;
 3 }
 4
 5 int max(int a, int b, int c) {
 6   return max(max(a, b), c);
 7 }
 8
 9 int x = max(3, 6);
10 int y = max(3, 6, 9);
```

9.1 多态性

```
 1 int max(int a, int b) {
 2   return a > b ? a : b;
 3 }
 4
 5 int max(int a, int b, int c) {
 6   return max(max(a, b), c);
 7 }
 8
 9 int x = max(3, 6);
10 int y = max(3, 6, 9);
```

同样的max，实际执行内容不同，也可以看作是一种朴素的多态。

9.1 多态性

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

9.1 多态性

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

狭义的面向对象程序设计中的多态，相同的指针，不同的执行效果。

9.1 多态性

赋值兼容规则

- 在公有派生方式下，派生类对象可以作为基类对象来使用，具体方式如下：

9.1 多态性

赋值兼容规则

- 在公有派生方式下，派生类对象可以作为基类对象来使用，具体方式如下：
  - 派生类的对象可以直接赋值给基类的对象

9.1 多态性

赋值兼容规则

- 在公有派生方式下，派生类对象可以作为基类对象来使用，具体方式如下：
  - 派生类的对象可以直接赋值给基类的对象
  - 基类对象的引用可以引用一个派生类对象

9.1 多态性

赋值兼容规则

- 在公有派生方式下，派生类对象可以作为基类对象来使用，具体方式如下：
  - 派生类的对象可以直接赋值给基类的对象
  - 基类对象的引用可以引用一个派生类对象
  - 基类对象的指针可以指向一个派生类对象

9.1 多态性

赋值兼容规则

- 在公有派生方式下，派生类对象可以作为基类对象来使用，具体方式如下：
  - 派生类的对象可以直接赋值给基类的对象
  - 基类对象的引用可以引用一个派生类对象
  - 基类对象的指针可以指向一个派生类对象
- 这个规则可以简单的理解为：所有的狗都是动物。但不是所有的动物都是狗 ——所有的子类对象都是基类的对象。

9.1 多态性

```
 1 #include <iostream>
 2
 3 using std::cout;
 4 using std::endl;
 5
 6 class Animal {
 7 public:
 8   void bark() const {
 9     cout << "Some animal is barking!" << endl;
10   }
11 };
12
13 class Dog : public Animal {
14 public:
15   void bark() const {
```

9.1 多态性

```
 1 #include <iostream>
 2
 3 using std::cout;
 4 using std::endl;
 5
 6 class Animal {
 7 public:
 8   void bark() const {
 9     cout << "Some animal is barking!" << endl;
10   }
11 };
12
13 class Dog : public Animal {
14 public:
15   void bark() const {
16     cout << "Woof! Woof!" << endl;
```

9.1 多态性

```
 9      cout << "Some animal is barking!" << endl;
10    }
11  };
12
13  class Dog : public Animal {
14  public:
15    void bark() const {
16      cout << "Wang wang!" << endl;
17    }
18  };
19
20  int main(int argc, char** argv) {
21    Animal *animal = new Dog();
22    animal->bark();
```

9.1 多态性

```
10    }
11 };
12
13 class Dog : public Animal {
14 public:
15   void bark() const {
16     cout << "Wang wang!" << endl;
17   }
18 };
19
20 int main(int argc, char** argv) {
21   Animal *animal = new Dog();
22   animal->bark();
23   return 0;
24 }
```

9.1 多态性

9.1 多态性

- 通过基类引用或指针所能看到的是一个基类对象

9.1 多态性

- 通过基类引用或指针所能看到的是一个基类对象
- 派生类中的成员对于基类引用或指针来说是“不可见的”

9.1 多态性

- 通过基类引用或指针所能看到的是一个基类对象
- 派生类中的成员对于基类引用或指针来说是“不可见的”
- 我们可以利用 C++ 的虚函数机制，将基类的相关函数声明为虚函数形式

9.1 多态性

- 通过基类引用或指针所能看到的是一个基类对象
- 派生类中的成员对于基类引用或指针来说是“不可见的”
- 我们可以利用 C++ 的虚函数机制，将基类的相关函数声明为虚函数形式
- 这样就可以通过基类引用或指针来访问派生类中的相关函数。

9.1 多态性

- 类系中不同类对象对同名操作可以表现各异
- 一旦将类系纳入统一处理,则类系中多态表现无法体现

9.1 多态性

- 重载普通的成员函数的两种方式：
  - 在同一个类中重载：重载函数是以参数特征区分的。
  - 派生类重载基类的成员函数。
- 由于重载函数处在不同的类中，因此它们的原型可以完全相同。调用时使用“类名::函数名”的方式加以区分。
- 以上两种重载的匹配都是在编译的时候静态完成的。

9.1 多态性

- 重载是一种简单形式的多态。
- C++提供另一种更加灵活的多态机制：虚函数。
- 虚函数允许函数调用与函数体的匹配在运行时才确定。
- 虚函数提供的是一种动态绑定的机制。

9.2 多态如何工作

虚函数

- 在基类中用virtual关键字声明的成员函数即为虚函数。
- 虚函数可以在一个或多个派生类中被重新定义，但要求在重定义时虚函数的原型（包括返回值类型、函数名、参数列表）必须完全相同。否则该函数将丢失虚特性。

9.2 多态如何工作

虚函数

- 在基类中用virtual将函数说明为虚函数。
- 在公有派生类中原型一致地重载该虚函数。
- 定义基类引用或指针，使其引用或指向派生类对象。
- 当通过该引用或指针调用虚函数时，该函数将体现出虚特性。

9.2 多态如何工作

虚函数

C++中，基类必须指出希望派生类重定义那些函数。
定义为virtual的函数是基类期待派生类重新定义的，
基类希望派生类继承的函数不能定义为虚函数。

9.2 多态如何工作

示例项目：

`/example/lec09/particle`

9.3 不恰当的虚函数

虚函数不是以重载形式往子类传递

```
#include <iostream>
using namespace std;
class Base {
public:
  virtual void fn(int x) {
    cout << "In Base class, int x = " << x << endl;
  } // 基类虚函数
};
class SubClass : public Base {
public:
  virtual void fn(float x) { // 子类虚函数,非基类传递之虚函数
    cout << "In SubClass, float x = " << x << endl;
  }
};

```

9.3 不恰当的虚函数

虚函数不是以重载形式往子类传递

```cpp
#include <iostream>
using namespace std;
class Base {
public:
  virtual void fn(int x) {
    cout << "In Base class, int x = " << x << endl;
  } // 基类虚函数
};
class SubClass : public Base {
public:
  virtual void fn(float x) { // 子类虚函数,非基类传递之虚函数
    cout << "In SubClass, float x = " << x << endl;
  }
};

```

9.3 不恰当的虚函数

虚函数不是以重载形式往子类传递

```
 5   virtual void fn(int x) {
 6     cout << "In Base class, int x = " << x << endl;
 7   } // 基类虚函数
 8 };
 9 class SubClass : public Base {
10 public:
11   virtual void fn(float x) { // 子类虚函数,非基类传递之虚函数
12     cout << "In SubClass, float x = " << x << endl;
13   }
14 };
15
16 // 当b引用的是基类对象,fn为基类函数,因无传递的虚函数,不显多态
17 void test(Base &b) {
18   // 当b引用子类对象时,能见均为基类函数
```

9.3 不恰当的虚函数

虚函数不是以重载形式往子类传递

```
12      cout << "In SubClass, float x = " << x << endl;
13    }
14 };
15
16 // 当b引用的是基类对象,fn为基类函数,因无传递的虚函数,不显多态
17 void test(Base &b) {
18    // 当b引用子类对象时,能见均为基类函数
19    b.fn(2);
20    // 当b引用基类对象时,能见均为基类函数
21    b.fn(2.2f);
22 }
23 int main() {
24    cout << "Calling test(bc)\n";
25    Base b;
26    test(b); // 传递一个基类对象
```

9.3 不恰当的虚函数

虚函数不是以重载形式往子类传递

```
16 // 当b引用的是基类对象,fn为基类函数,因无传递的虚函数,不显多态
17 void test(Base &b) {
18     // 当b引用子类对象时,能见均为基类函数
19     b.fn(2);
20     // 当b引用基类对象时,能见均为基类函数
21     b.fn(2.2f);
22 }
23 int main() {
24     cout << "Calling test(bc)\n";
25     Base b;
26     test(b); // 传递一个基类对象
27     cout << "Calling test(sc)\n";
28     SubClass sc;
29     test(sc); // 传递一个子类对象
30 }
```

9.4 返回自身的虚函数

若返回类型是类本身,则子类可返回子类实体,虚函数仍往子类传递

```
#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;
  }
```

9.4 返回自身的虚函数

若返回类型是类本身,则子类可返回子类实体,虚函数仍往子类传递

```
#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;
  }
```

9.4 返回自身的虚函数

若返回类型是类本身,则子类可返回子类实体,虚函数仍往子类传递

```
 6      cout << "This is Base class.\n";
 7      return this;
 8    }
 9  };
10  class SubClass : public Base {
11  public:
12    SubClass *afn() {
13      cout << "This is SubClass.\n";
14      return this;
15    }
16  };
17  void test(Base &x) {
18    Base *b = x.afn();
19    cout << "Return value calling:" << endl;
20    b->afn();
```

9.4 返回自身的虚函数

若返回类型是类本身,则子类可返回子类实体,虚函数仍往子类传递

```
12    SubClass *afn() {
13      cout << "This is SubClass.\n";
14      return this;
15    }
16  };
17  void test(Base &x) {
18    Base *b = x.afn();
19    cout << "Return value calling:" << endl;
20    b->afn();
21  }
22  int main() {
23    Base b;
24    test(b); // 传递基类对象
25    SubClass sc;
26    test(sc); // 传递子类对象
```

9.4 返回自身的虚函数

若返回类型是类本身,则子类可返回子类实体,虚函数仍往子类传递

```
13      cout << "This is SubClass.\n";
14      return this;
15    }
16  };
17  void test(Base &x) {
18    Base *b = x.afn();
19    cout << "Return value calling:" << endl;
20    b->afn();
21  }
22  int main() {
23    Base b;
24    test(b); // 传递基类对象
25    SubClass sc;
26    test(sc); // 传递子类对象
27  }
```

9.5 虚函数的限制

- 只有成员函数才能声明为虚函数
- 不能将虚函数说明为全局函数
- 不能将虚函数说明为友元函数
- 静态成员函数不能是虚函数
- 内联函数不能是虚函数
- 构造函数不能是虚函数（析构函数通常为虚函数）

9.6 抽象类与纯虚函数

9.6 抽象类与纯虚函数

- 基类表示抽象的概念，它提供一些公共的接口，表示这类对象拥有的共同操作。

9.6 抽象类与纯虚函数

- 基类表示抽象的概念，它提供一些公共的接口，表示这类对象拥有的共同操作。
- 基类中的这些公共接口只需要有说明而不需要有实现，即纯虚函数。

9.6 抽象类与纯虚函数

- 基类表示抽象的概念，它提供一些公共的接口，表示这类对象拥有的共同操作。
- 基类中的这些公共接口只需要有说明而不需要有实现，即纯虚函数。
- 纯虚函数刻画了派生类应该遵循的协议，这些协议的具体实现由派生类来决定。

9.6 抽象类与纯虚函数

9.6 抽象类与纯虚函数

- 若类中含纯虚函数(可能不止一个),则该类是抽象类

9.6 抽象类与纯虚函数

- 若类中含纯虚函数(可能不止一个),则该类是抽象类
- 若类为抽象类,则必含纯虚函数

9.6 抽象类与纯虚函数

- 若类中含纯虚函数(可能不止一个),则该类是抽象类
- 若类为抽象类,则必含纯虚函数
- 抽象类不能创建对象

9.6 抽象类与纯虚函数

- 若类中含纯虚函数(可能不止一个),则该类是抽象类
- 若类为抽象类,则必含纯虚函数
- 抽象类不能创建对象
- 抽象类的子类若其虚函数都有定义(即不再有纯虚函数)，则为具体类

9.6 抽象类与纯虚函数

示例：将 particle 项目中的 Shape 改为抽象类。

作业

基于 `particle` 项目，实现一个粒子系统，有以下功能：

- 随机生成三种形状：圆、长方形、正方形
- 其中，圆和长方形继承自 `Shape`，正方形继承自长方形
- 每个粒子的颜色均不相同

提示：构建任意颜色方法：`(Color){r, g, b, a}`

参考资料：https://www.raylib.com/cheatsheet/cheatsheet.html