

Полиморфизъм. Виртуални функции

Любомир Чорбаджиев
Технологическо училище "Електронни системи"
Технически университет, София
lchorbadjiev@elsys-bg.org
Revision : 1.2

19 декември 2004 г.

Заместване на функции

- Нека разгледаме следните два класа:

```
1 class Employee {
2     //...
3 public:
4     //...
5     void print(void) const;
6 };
1 class Manager: public Employee {
2     //...
3 public:
4     //...
5     void print(void) const;
6 };
```

- И в двата класа е дефинирана функцията `print()`. Идеята на тази функция е да извежда на стандартния изход данните, специфични за съответният клас.

Заместване на функции

- Нека сега разгледаме следният код, който използва класовете `Employee` и `Manager`.

```
1 Employee e1("ИванРаботников", 8101011);
2 Manager m1("ШефИванов", 8012121, 1);
3 //...
4 Employee* employee_list[10];
5 employee_list[0]=&e1;
6 employee_list[1]=&m1;
7 //...
8 employee_list[0]->print();
9 employee_list[1]->print();
```

- Кой метод се извиква в ред 9? `Employee::print()` или `Manager::print()`?

Заместване на функции

- Трябва да се реализира функция, която да разпечатва на стандартния изход данните за всички работници.

- Нека разгледаме следната примерна реализация:

```
1 void print_all(Employee* employees[]) {
2     for(int i=0; employees[i]!=NULL; i++)
3         employees[i]->print();
4 }
```

- В тази функция в ред 3 винаги се извиква функцията `Employee::print()`. Такава реализация е неудовлетворителна, тъй като се губи спецификата на различните видове работници.

Член-променлива за типа

- Вариант за решаването на проблема е в класа `Employee` да се добави член-променлива, в която се помни типа на обекта.

```
3 class Employee {
4 public:
5     enum EmployeeType {E,M};
6     EmployeeType employee_type;
7 private:
8     string name_;
9     long id_;
10 public:
11     Employee(string name, long id)
12     : employee_type(E),
13       name_(name),
14       id_(id)
15     {}
16     //..
17 };
```

4

```
18 class Manager:public Employee {
19     int level_;
20 public:
21     Manager(string name, long id, int level)
22     : Employee(name, id),
23       level_(level)
24     {
25         employee_type=Employee::M;
26     }
27 };
28 void print_all(Employee* employees[]){
29     for(int i=0;employees[i]!=NULL;i++){
30         if(employees[i]->employee_type==Employee::M){
31             // print manager
32         }
33         else {
34             // print employee
35         }
36     }
37 }
```

Член-променлива за типа

- Такова решение на проблема може да работи в малка програма, но когато йерархията от класове нараства, броят на проверките за типа на променливата също нараства;
- Когато се добавя нов клас в йерархията трябва да се променят всички функции, които зависят от проверки за типа;
- Когато се добавя нов клас в йерархията трябва да се промени и базовият клас;
- Използването на член-променлива за типа противоречи на идеята за капсулиране на данните;

5

Виртуални функции

- За решаването на такъв тип проблеми в езика C++ е реализирана идеята за виртуални функции.

```
4 class Employee {
5     string name_;
6     long id_;
7 public:
8     Employee(string name, long id)
9     : name_(name),
10       id_(id)
11     {}
12     virtual void print(void) const{
13         cout << name_ << "\n"
14              << id_ << endl;
15     }
16 };
```

6

```

17 class Manager:public Employee {
18     int level_;
19 public:
20     Manager(string name, long id, int level)
21     : Employee(name, id),
22       level_(level)
23     {}
24     void print(void) const {
25         Employee::print();
26         cout << "\tlevel:" << level_ << endl;;
27     }
28 };

```

Виртуални функции

- Функцията `Manager::print()` замества (override) функцията в базовия клас `Employee::print()`;
- Функцията в базовият клас е дефинирана като виртуална. Това означава, че изборът, коя функция да се извика се определя по време на изпълнение на програмата, в зависимост от действителния тип на променливата.

```

29 void print_all(Employee* employees[]) {
30     for(int i=0; employees[i]!=NULL; i++)
31         employees[i]->print();
32 }

```

Виртуални функции

```

33 int main(int argc, char* argv[]){
34     Employee e("Brown", 81010110L);
35     Manager m("Smith", 80121212L, 1);
36
37     Employee* employees[3];
38     employees[0]=&e;
39     employees[1]=&m;
40     employees[2]=NULL;
41
42     print_all(employees);
43     return 0;
44 }

```

Изход:

```

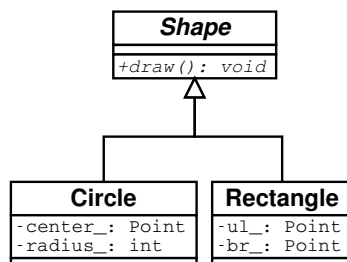
Brown 81010110
Smith 80121212
    level:1

```

Виртуални функции

- Когато функция от производния клас, има същата сигнатура като виртуална функция в базовия клас, се казва че тя **замества (override)** виртуалната функция от базовия клас;
- Когато се извиква виртуална функция, то автоматично се използва най-подходящият ѝ заместник, в зависимост от действителния тип на извикващия обект;
- Поведението, при което конкретната функция, която се извиква, зависи от динамичния тип на обект, чрез който е извикана, се нарича **полиморфизъм** или **динамично свързване**;
- За да бъде една член-функция полиморфна в C++ е необходимо тя да се декларира като виртуална с помощта на модификатора **virtual**;

Абстрактни класове



```
1 class Shape {
2 public:
3     virtual void draw(void) const=0;
4 };
```

В ред 3 е дефинирана **чисто виртуална** функция.

10

Абстрактни класове

- Класове, в които са дефинирани една или повече чисто виртуални функции се наричат **абстрактни**.
- Не е възможно да се създаде обект от абстрактен клас:

```
Shape s; //error!!!
```
- Абстрактните класове се използват като **интерфейсни** класове и като базови за други класове;
- Чисто виртуалните функции, които не са определени в производните класове остават чисто виртуални. Това означава, че е възможно е и базовия и производния клас да бъдат абстрактни.

11

Абстрактни класове

```
1 #include <iostream>
2 using namespace std;
3 class Point {
4     double x_;
5     double y_;
6 public:
7     Point(double x, double y)
8         : x_(x), y_(y)
9     {}
10    double get_x() const {return x_;}
11    double get_y() const {return y_;}
12    void dump() const {
13        cout << "(" << x_ << ", " << y_ << ")";
14    }
15 };
```

12

```
16 class Shape {
17 public:
18     virtual void draw(void) const=0;
19 };
20
21 class Circle: public Shape {
22     Point center_;
23     int radius_;
24 public:
25     Circle(const Point& center, int radius)
26         : center_(center), radius_(radius)
27     {}
28     void draw(void) const {
29         cout << "Circle::draw(";
30         center_.dump();
31         cout << ", " << radius_ << ")" << endl;
32     }
33 };
```

```

34 class Rectangle: public Shape {
35     Point ul_;
36     Point br_;
37 public:
38     Rectangle(const Point& ul, const Point& br)
39         : ul_(ul), br_(br)
40     {}
41     void draw(void) const {
42         cout << "Rectangle::draw(";
43         ul_.dump();
44         cout << ", ";
45         br_.dump();
46         cout << ")" << endl;
47     };
48 };

```

Абстрактни класове

```

1 class Drawing {
2     Shape** shapes_;
3 public:
4     //...
5     void draw(void) const {
6         for(int i=0; shapes_[i]!=NULL; i++)
7             shapes_[i]->draw();
8     };
9 };

```

Виртуален деструктор

- Нека разгледаме следният клас:

```

1 class MyDrawing {
2     Shape** shapes_;
3 public:
4     //...
5     ~MyDrawing(void) {
6         for(int i=0; shapes_[i]!=NULL; i++)
7             delete shapes_[i];
8     };
9 };

```

- В ред 7 се извиква деструкторът на Shape;

Виртуален деструктор

- В ред 7 се извиква деструкторът на Shape;
- В класа Drawing се държат обекти, които са наследници на абстрактния клас Shape. При унищожаването на тези обекти трябва да се извика не деструктора на базовият клас, а деструктора на съответния клас наследник;
- Това означава че деструкторите на йерархията от класове наследници на Shape трябва да бъдат **полиморфни**;
- За да се обезпечи такова поведение, в класа Shape трябва да се дефинира виртуален деструктор:

```

1 class Shape {
2 public:
3     virtual ~Shape(void) {}
4     virtual void draw(void) const=0;
5 };

```

Копиране. Срез

- Нека разгледаме следният пример:

```
1 void fun(Employee ee){
2     ee.print();
3 };
4 int main() {
5     Employee e(/*...*/);
6     Manager m(/*...*/);
7     fun(e);
8     fun(m);
9     return 0;
10 }
```

- При предаване на параметър на функция по стойност се извиква копиращ конструктор (вж. ред 1).
- Копиращият конструктор “срязва” действително предадения параметър (вж. ред 8).
- В тялото на функцията се работи с копието на предадения параметър.

16

Динамично преобразуване

```
1 void fun(Shape* sh){
2     Circle* pc=dynamic_cast<Circle*>(sh);
3     if(pc!=NULL){
4         // it's a circle
5         return;
6     }
7     Rectangle* pr=dynamic_cast<Rectangle*>(sh);
8     if(pr!=NULL){
9         // it's a rectangle
10    }
11 }
```

18

Динамично преобразуване

- C++ поддържа RTTI (run-time type identification, идентификация на типа по време на изпълнение);
- RTTI ни дава възможност да идентифицираме истинския тип, към който сочи даден указател;
- RTTI може да се използва по няколко начина — единият е динамичното преобразуване на типовете **dynamic_cast<...>(...)**:

```
1 void fun(Shape* sh){
2     Circle* pc=dynamic_cast<Circle*>(sh);
3     //...
4 }
```

- Когато преобразуването е успешно операторът за динамично преобразуване на типа **dynamic_cast<...>(...)** връща ненулев указател;

17