

Семестр 1. Лекция 11. Полиморфизм.  
Статическое и динамическое связывание.

Евгений Линский

24 Ноября 2017

- ▶ Необходимо написать программу для бухгалтера в небольшом стартапе.
- ▶ Программа должна хранить информацию о сотрудниках и рассчитывать суммарный объем заработной платы (ЗП). На данный момент есть две должности:
  - Программист: имя, возраст, ставка, категория, премия за релиз.
  - Продавец: имя, возраст, город, цена продукта, процент с продажи, объем.
- ▶ Необходимо написать программу так, чтобы когда стартап вырастет и наймет сотрудников на другие должности (тестер, маркетолог и т.д.) со своими алгоритмами расчета ЗП, то программу не пришлось бы сильно менять.

На основе требований попробуем продумать main.

```
main() {  
    WorkerDB db; // stores info about workers  
    Delevloper d1("Masha", 20, 2000, 1, false);  
    Delevloper d2("Kolya", 21, 2000, 1, false);  
    Sale s("Marina", 25, "Spb", 500, 20, 100);  
    db.add(d1);  
    db.add(d2);  
    db.add(s);  
    int total = db.getTotalSalary();  
}
```

- ▶ Необходимо защитить WorkerDB от изменений, т.е. когда появятся Marketing и Tester этот класс не пришлось бы переделывать!
- ▶ Полиморфный код может работать с разными типами данных (в C++ с наследниками некоторого базового класса).

- ▶ Вспомним лекцию про *LinkedList* и его наследника *DoubleList*.
- ▶ В функцию *fill(LinkedList &l, int value, int count)* можно передать объекты обоих классов, потому что *DoubleList* “умеет” все (поддерживает те же публичные методы), что умеет *LinkedList*.
- ▶ Воспользуемся этой идеей. Зададим базовый класс *Worker*, который будет описывать “контракт” (набор публичных методов), который должен выполнить любой класс, чтобы *WorkerDB* смог с ним работать.

```
class Worker {
protected:
    char *name;
    int age;
public:
    Worker(const char *n, ...) {
        name = new char [strlen(n)+1]; strcpy(name, n);
    }
    ~Worker() { delete [] name; }
    // dummy implementation
    virtual int getSalary() const { return 0; }
};
```

```
class Developer: public Worker {
protected:
    int base;
    int level;
    bool isBonus;
public:
    Developer(const char* n, ...) : Worker(n, ...) { ... }
    //destructor is not required
    virtual int getSalary() {
        int res = isBonus ? base * level + 500 : base * level;
        return res;
    }
};
```

```
class Sale: public Worker {
protected:
    char *city;
    int price;
    int num;
    float percent;
public:
    Sale(const char* n, const char *c ...) : Worker(n, ...) {
        city = new char [strlen(c) + 1];
        strcpy(city, c);
    }
    ~Sale() { delete [] city; }
    ...
    virtual int getSalary() const {
        int res = price * num * percent;
        return res;
    }
};
```

```
class WorkerDB {
private:
    // don't forget to add default constructor to Worker
    Worker workers[100];
    int size;
public:
    void addWorker(const Worker& w) {
        workers[size++] = w;
    }

    int getTotalSalary() const {
        int sum = 0;
        for(int i = 0; i < size; i++) {
            sum += workers[i].getSalary();
        }
    }
};
```

main: *db.add(d1); db.add(d2); db.add(s);*

Для *worker[0]* вызовется *getSalary()* из *Developer*.

Для *worker[2]* вызовется *getSalary()* из *Sale*.

Можно забыть в производном классе (например, в *Sale*) перекрыть метод *getSalary()*, тогда будет использован метод заглушка (dummy из *Worker*) и зарплата будет 0.

Решение:

```
class Worker {  
    ...  
    virtual int getSalary() = 0 const;  
};
```

- ▶ *getSalary()* — чисто (pure) виртуальная функция, класс *Worker* — абстрактный, т.е. объект такого класса создать нельзя (не скомпилируется), но можно использовать этот класс при наследовании.
- ▶ Если наследник (например, *Sale*) не перекроет *getSalary()*, то тоже станет абстрактным.



Скопируем в массив только часть объекта. Ничего работать не будет!

```
// sizeof(Developer) > sizeof(Worker)
workers[size++] = w;
```

Решение:

```
class WorkerDB {
    Worker* workers[100];
    void addWorker(const Worker* w) { workers[size++] = w; }
    ~WorkerDB() {
        for(int i = 0; i < size; i++) {
            delete workers[i];
        }
    }
};

main() {
    WorkerDb db;    Developer *d1 = new Developer(...);
    db.addPerson(d1)
}
```

Деструктор Sale из деструктора WorkerDB не вызовется.

```
class Sale {
    ~Sale() { ... }
};

class WorkerDB {
    ~WorkerDB() {
        for(int i = 0; i < size; i++) {
            delete workers[i];
        }
    }
};
```

```
Developer *d = new Developer(...);  
//destructor Worker called, destructor Developer called  
delete d;  
  
Worker *w = new Developer(...);  
//destructor Worker called, destructor Developer not called  
delete w;
```

Решение:

```
class Sale {  
    virtual ~Sale() { ... }  
};
```

Подробности в рассказе про связывание (static/dynamic binding).

# Статическое и динамическое связывание

В данном контексте “связывание” — сопоставление имени функции и ее адреса в памяти.

Статическое связывание — имя можно заменить на адрес на этапе построения программы.

```
Sale s(...);  
int r = s.getSalary();
```

Динамическое связывание — имя можно заменить на адрес только на этапе выполнения программы.

```
Worker *w;  
int d;  
scanf("%d", &d);  
if(d > 0) {  
    w = new Developer(...);  
}  
else {  
    w = new Sale(...);  
}  
w->getSalary();
```

- ▶ По умолчанию в C++ используется статическое связывание.
- ▶ Если у метода задано ключевое слово `virtual`, то для него используется динамическое связывание.

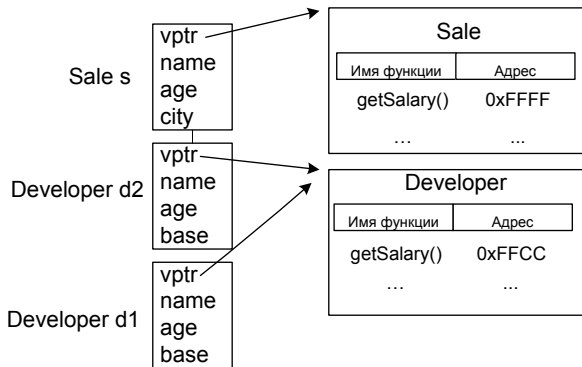
```
class Sale {  
    ...  
    virtual ~Sale() { ... }  
};  
class Worker {  
    ...  
    virtual int getSalary() = 0 const;  
};
```

# Таблица виртуальных функций

- ▶ Как при динамическом связывании происходит выбор нужной функции? С помощью таблицы виртуальных функций!
- ▶ При старте программы для каждого класса с виртуальными методами создается таблица, в которой задано соответствие между именем функции и ее адресом в памяти (указатель на функцию).
- ▶ Каждый объект класса с виртуальными функциями хранит указатель (скрытое поле *vptr*) на начало таблицы виртуальных функций своего класса.
- ▶ В месте вызова виртуальной функции компилятор генерирует следующий код:
  - 1 Пойди по адресу *vptr*.
  - 2 Найди в таблице функцию с нужным именем.
  - 3 Считай из таблицы адрес функции и перейди на нее.

Вызов какой функции происходит быстрее: обычной или виртуальной?

# Таблица виртуальных функций



На самом деле:

- ▶ Имя функции в виде строки ни в коде программы, ни в таблице не хранится.
- ▶ Просмотр все строк таблицы для поиска не производится.

Придумайте способ ускорения описанного алгоритма вызова виртуальной функции. Hint: относительные адреса в памяти.