Семестр 1. Лекция 14. Наследование: детали.

Евгений Линский

15 Декабря 2017

План

- Сортировка в стиле C vs в стиле ООП
- Структуры данных в стиле C vs в стиле ООП
- Множественное наследование
- public/private/protected наследование

Сортировка в стиле C

```
struct point_s {
  int x, y;
};

void qsort (void* base, size_t num, size_t size,
            int (*compar)(const void*,const void*)){}

int cmp_point1(const void* v1,const void* v2) {
  const struct point_s *p1 = (const struct point_s*)v1;
  const struct point_s *p2 = (const struct point_s*)v2;
  ...
}

struct point_s points[10];
qsort(points, 10, sizeof(points[0]), cmp_point1);
```

*void** – нет никакой проверки типов со стороны компилятора!

Сортировка в стиле C

Или так:

```
int cmp_point2(const void* v1,const void* v2) {
  const struct point_s *p1 = (const struct point_s*)*v1;
  const struct point_s *p2 = (const struct point_s*)*v2;
  ...
}
struct point_s **points;
//allocation:  points = new ...  and for() { points[i] = new }
qsort(points, N, sizeof(points[0]), cmp_point2);
```

Можно использовать и для структур (классов) и для примитивных типов.

```
int cmp_int(const void* v1,const void* v2) {
  const int *p1 = (const int*)v1;
  const int *p2 = (const int*)v2;
  ...
}
int numbers[10];
qsort(numbers, 10, sizeof(numbers[0]), cmp_int);
```

Сортировка в стиле ООП

ООП:

```
class Comparable{
    //or virtual bool operator<(const Comparable *v) = 0 const;
    virtual int compare(const Comparable* v) = 0 const;
};
void nsort(Comparable** m, size_t size){
  ...
  m[i] =
}

class Point: public Comparable {
private:
  int x, y;
public:
  virtual int compare(const Comparable* v) const {...};
};

Point** points;
//allocation:  points = new ...  and for() { points[i] = new }
nsort(points, N);
```

Сортировка в стиле ООП

А можно ли было сделать так?

```
void nsort(Comparable* m, size_t size){}

Point p[100];
nsort(p, 100);
```

Для примитивных типов надо написать “обертку” (наследник *Comparable*).

```
class Integer : public Comparable {
private:
  int value;
public:
  virtual int compare(const Comparable* v) const { ... };
};
```

Связный список в стиле C: неинтрузивный

```
struct node_s {
  void* user_data;
  struct node_s *next;
};
struct list_s {
  struct node_s *head;
};

void push_back(struct list_s *l, struct node_s *n);
```

```
struct node_s *n = malloc(sizeof(struct node_s));
n->user_data = malloc(sizeof(struct point_s));
push_back(&l, n);
```

- Два вызова malloc!
- *void** – нет никакой проверки типов со стороны компилятора!

Связный список в стиле C: интрузивный

```
struct node_s {
  struct node_s *next;
};
struct list_s {
  struct node_s *head;
}

void push_back(struct list_s *l, struct node_s *n);
```

```
struct point_s {
  int x, y;
  struct node_s node;
};
list_t l;
struct point_s *pn = malloc(sizeof(*pn));
push_back(&l, &pn->node);
```

- Один malloc!
- Но нужен трюк, чтобы, имея указатель на node, добраться до полей x и y (см. лабу про интрузивные списки)

Связный список в стиле ООП

Требуется сделать базовый класс, от которого должны быть отнаследованы все классы в языке C++.

```
class Object {
public:
  // for red-black tree/priority queue/etc
  virtual bool operator<(const Object* o)
    { return this < o; }
  virtual bool operator==(const Object* o)
    { return this == o; }
  // for hash table
  virtual int hash(const Object* o)
    { return (int)this; }
};
```

Связный список в стиле ООП

```
class Node {
private:
  Object *o;
  Node *next;
  ...
};

class List {
  Node *head;
public:
  void push_back(Node *n);
};
```

Связный список в стиле ООП

```
class Point: public Object {
  int x, y;
  virtual bool operator<(const Object* o) {
    Point *p = (Point*)o;
    ...
  }
};

List l;
Node *n = new Node;
n->setData(new Point(3,3));
l.push_back(n);
```

- Два вызова new!
- Необходимо сделать обертки для примитивных типов *class Integer: public Object*.
- Иногда придется использовать множественное наследование *class Developer: public Object, public Person*.
- Vector придется делать так: *Object** array*.

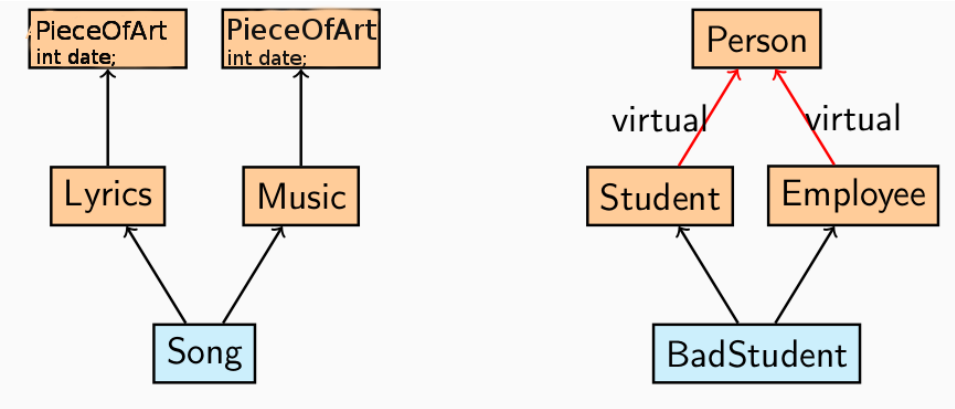
Множественное наследование

Множественное наследование (multiple inheritance) — возможность наследовать сразу несколько классов.

```
struct Student {
    string name()       const { return name_; }
    string university() const { return university_; }
private:
    string name_, university_;
};
struct Employee {
    string name()    const { return name_; }
    string company() const { return company_; }
private:
    string name_, company_;
};
struct BadStudent: Student, Employee {
    string name() const { return Student::name(); }
};
```

- **Два метода с одной сигнатурой (неоднозначность надо разрешить указанием полного имени Student::name()).**
- **Два экземпляра name_!**

Виртуальное наследование



Иногда два экземпляра базового класса это нормально. Дата создания есть и мелодии и у текста песни (и они могут быть разные).

```
class Song: public  Music, public Lyriscs {
  show() { std::cout << Music::date << Lyrics::date; }
}
```

Иногда два экзмепляра это странно (имя у человека одно, вне зависимости от того студент он или сотрудник), тогда можно использовать виртуальное наследование — будет один экземпляр.

```
struct Person {string name_};
struct Student  : virtual public Person {};
struct Employee : virtual public Person {};
struct BadStudent : public Student, public Employee {};
```

Множественное наследование

Размещение в памяти Song:

PieceOfArt	Music	PieceOfArt	Lyrics	поля Song

Размещение в памяти BandStudent:

Person	Student	Employee	поля BadStudent

Виртуальное наследование: конструктор

Кто вызывает конструктор базового класса?

```
struct Person {
    explicit Person(string const& name)
        : name_(name) {}
    string name_;
};
struct Student  : virtual Person {
    explicit Student(string const& name) : Person(name) {}
};
struct Employee : virtual Person {
    explicit Employee(string const& name) : Person(name) {}
};
struct BadStudent : Student, Employee {
    explicit BadStudent(string const& name)
        : Person(name), Student(name), Employee(name)
    {}
};
```

Конструктор "дедушки" вызовется из BadStudent.

Интерфейсы

- Но лучше (если возможно), все перепроектировать с использованием интерфейсов (все методы абстрактные, нет полей)
- Лучше наследовать интерфейсы, а не реализацию. Тогда нет описанных ранее проблем с дублированием переменных и методов.
- В некоторых языках разрешено только такое множественное наследование.

Интерфейсы

```
struct Person {
    string name() const { return name_; }
    string name_;
};
struct IStudent {
    virtual string name()       const = 0;
    virtual string university() const = 0;
    virtual ~IStudent() {}
};
struct IFullTimeEmployee {
    virtual string name()    const = 0;
    virtual string company() const = 0;
    virtual ~IFullTimeEmployee() {}
};
struct BadStudent: Person, IStudent, IFullTimeEmployee {
    string name()       const { return Person::name(); }
    string university() const { return university_; }
    string company()    const { return company_; }
    string university_, company_;
};
```

private/protected/public наследование

```
class A {
public:
    int x;
protected:
    int y;
private:
    int z;
};
```

private/protected/public наследование

```
class B : public A {
    // x is public
    // y is protected
    // z is not accessible from B
};

class C : protected A {
    // x is protected
    // y is protected
    // z is not accessible from C
};

class D : private A   { // 'private' is default for classes
    // x is private
    // y is private
    // z is not accessible from D
};
```

private/protected/public наследование

- Обычно при дизайне иерархии классов стараются сохранить между классами отношения из “реального мира”.
- Два типа отношений:
  - “has a”: у машины есть двигатель, у машины есть тормоза.

```
class Car {
  Engine e;
  Brakes b;
};
```

  - “is a”: грузовик это машина, автобус это машина

```
class Track : public Car { };
class Bus : public Car { };
```

private/protected/public наследование

Иногда в реализации надо сделать 'хак': у поля класса вызвать protected метод или перекрыть виртуальную функции и т.п. В этом случае можно применить private/protected наследование.

```
class Engine {
protected:
  void maintenanceCheck() { ... };
...
};

public class Car : private Engine {
  void reset() {
    maintenanceCheck();
  }
};
```

Хотим подчеркнуть, что все-таки машина не является двигателем и это просто хак.