Лекция 7. Наследование, перегрузка, переопределение

Александр Смаль

CS центр

10 октября 2017

Санкт-Петербург

Наследование

Наследование — это механизм, позволяющий создавать производные классы, расширяя уже существующие.

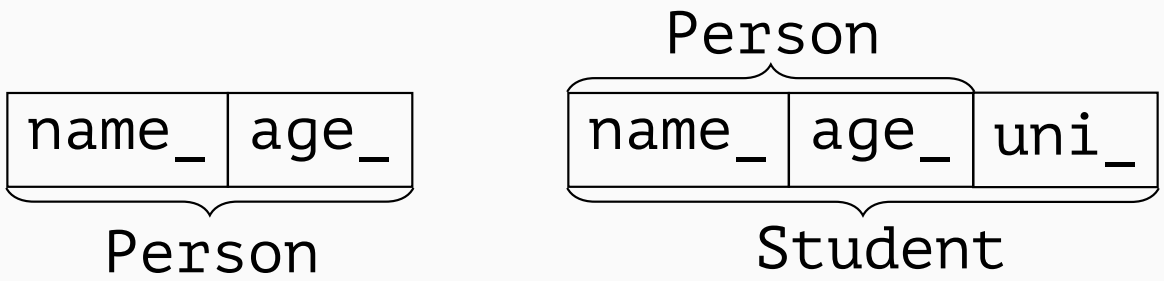
```
struct Person {
    string name() const { return name_; }
    int     age() const { return age_; }
private:
    string name_;
    int age_;
};

struct Student : Person {
    string university() const { return uni_; }
private:
    string uni_;
};
```

Класс-наследник

У объектов класса-наследника можно вызывать публичные методы родительского класса.

```
Student s;
cout << s.name() << endl
     << s.age()  << endl
     << s.university() << endl;
```

Внутри объекта класса-наследника хранится экземпляр родительского класса.



Создание/удаление объекта класса-наследника

При создании объекта производного класса сначала вызывается конструктор родительского класса.

```
struct Person {
    Person(string name, int age)
        : name_(name), age_(age)
    {}
};
struct Student : Person {
    Student(string name, int age, string uni)
        : Person(name, age), uni_(uni)
    {}
};
```

После деструктора `Student` вызывается деструктор `Person`.

Приведения

Для производных классов определены следующие приведения:

```
Student s("Alex", 21, "Oxford");
Person & l = s;  // Student & -> Person &
Person * r = &s; // Student * -> Person *
```

Поэтому объекты класса-наследника могут присваиваться объектам родительского класса:

```
Student s("Alex", 21, "Oxford");
Person p = s; // Person("Alex", 21);
```

При этом копируются только поля класса-родителя (срезка).
(Т.е. в данном случае вызывается конструктор копирования `Person(Person const& p)`, который не знает про `uni_`.)

Модификатор доступа `protected`

- Класс-наследник не имеет доступа к `private`-членам родительского класса.
- Для определения закрытых членов класса доступных наследникам используется модификатор `protected`.

```
struct Person {
    ...
protected:
    string name_;
    int age_;
};

struct Student : Person {
    ... // можно менять поля name_ и age_
};
```

Перегрузка функций

В отличие от C в C++ можно определить несколько функций с одинаковым именем, но разными параметрами.

```
double square(double d) { return d * d; }

int    square(int    i) { return i * i; }
```

При вызове функции по имени будет произведен поиск наиболее подходящей функции:

```
int      a = square(4);       // square(int)
double   b = square(3.14);    // square(double)
double   c = square(5);       // square(int)
int      d = square(b);       // square(double)
float    e = square(2.71f);   // square(double)
```

Перегрузка методов

```
struct Vector2D {
    Vector2D(double x, double y) : x(x), y(y) {}

    Vector2D mult(double d) const
        { return Vector2D(x * d, y * d); }

    double  mult(Vector2D const& p) const
        { return x * p.x + y * p.y;   }

    double x, y;
};
```

```
    Vector2D p(1, 2);
    Vector2D q = p.mult(10); // (10, 20)
    double  r  = p.mult(q);  // 50
```

Перегрузка при наследовании

```
struct File {
    void write(char const * s);
    ...
};

struct FormattedFile : File {
    void write(int i);
    void write(double d);
    using File::write;
    ...
};
```

```
    FormattedFile f;
    f.write(4);
    f.write("Hello");
```

Правила перегрузки

1. Если есть точное совпадение, то используется оно.
2. Если нет функции, которая могла бы подойти с учётом преобразований, выдаётся ошибка.
3. Есть функции, подходящие с учётом преобразований:
   - 3.1 Расширение типов.
     `char, signed char, short` → `int`
     `unsigned char, unsigned short` → `int`/`unsigned int`
     `float` → `double`
   - 3.2 Стандартные преобразования (числа, указатели).
   - 3.3 Пользовательские преобразования.

   В случае нескольких параметров нужно, чтобы выбранная функция была *строго лучше* остальных.

NB: перегрузка выполняется на этапе компиляции.

Перегрузка методов (overloading)

```
struct Person {
    string name() const { return name_; }
    ...
};
struct Professor : Person {
    string name() const {
        return "Prof. " + Person::name();
    }
    ...
};
```

```
    Professor pr("Stroustrup");
    cout << pr.name() << endl; // Prof. Stroustrup
    Person * p = &pr;
    cout << p->name() << endl; // Stroustrup
```

Переопределение методов (overriding)

```
struct Person {
    virtual string name() const { return name_; }
    ...
};
struct Professor : Person {
    string name() const {
        return "Prof. " + Person::name();
    }
    ...
};
```

```
    Professor pr("Stroustrup");
    cout << pr.name() << endl; // Prof. Stroustrup
    Person * p = &pr;
    cout << p->name() << endl; // Prof. Stroustrup
```

Чистые виртуальные (абстрактные) методы

```
struct Person {
  virtual string occupation() const = 0;
  ...
};
struct Student : Person {
  string occupation() const {return "student";}
  ...
};
struct Professor : Person {
  string occupation() const {return "professor";}
  ...
};
```

```
  Person * p = next_person();
  cout << p->occupation();
```

Виртуальный деструктор

К чему приведёт такой код?

```
struct Person {
    ...
};
struct Student : Person {
    ...
private:
    string uni_;
};

int main() {
    Person * p = new Student("Alex",21,"Oxford");
    ...
    delete p;
}
```

Виртуальный деструктор

Правильная реализация:

```
struct Person {
    ...
    virtual ~Person() {}
};
struct Student : Person {
    ...
private:
    string uni_;
};

int main() {
    Person * p = new Student("Alex",21,"Oxford");
    ...
    delete p;
}
```

Полиморфизм

Полиморфизм

Возможность единообразно обрабатывать разные типы данных.

Перегрузка функций

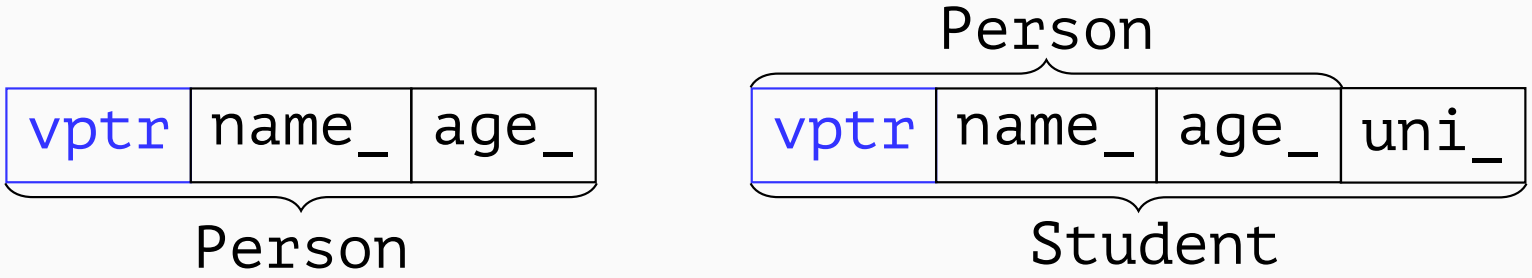
Выбор функции происходит в момент компиляции на основе типов аргументов функции, *статический полиморфизм*.

Виртуальные методы

Выбор метода происходит в момент выполнения на основе типа объекта, у которого вызывается виртуальный метод, *динамический полиморфизм*.

Таблица виртуальных методов

- Динамический полиморфизм реализуется при помощи таблиц виртуальных методов.
- Таблица заводится для каждого *полиморфного* класса.
- Объекты полиморфных классов содержат указатель на таблицу виртуальных методов соответствующего класса.



- Вызов виртуального метода — это вызов метода по адресу из таблицы (в коде сохраняется номер метода в таблице).

```
p->occupation(); // p->vptr[1]();
```

Таблица виртуальных методов

```
struct Person {
    virtual ~Person() {}
    string name() const {return name_;}
    virtual string occupation() const = 0;
    ...
};
struct Student : Person {
    string  occupation() const {return "student";}
    virtual int group() const {return group_;}
    ...
};
```

Person

0	~Person	0xab22
1	occupation	0x0000

Student

0	~Student	0xab46
1	occupation	0xab68
2	group	0xab8a

Построение таблицы виртуальных методов

```
struct Person {
    virtual ~Person() {}
    virtual string occupation() = 0;
    ...
};

struct Teacher : Person {
    string occupation() {...}
    virtual string course() {...}
    ...
};

struct Professor : Teacher {
    string occupation() {...}
    virtual string thesis() {...}
    ...
};
```

Person

0	~Person	0xab20
1	occupation	0x0000

Teacher

0	~Teacher	0xab48
1	occupation	0xab60
2	course	0xab84

Professor

0	~Professor	0xaba8
1	occupation	0xabb4
2	course	0xab84
3	thesis	0xabc8