Лекция 5. Наследование

Илья Макаров

CS центр

5 октября 2021

Санкт-Петербург

Наследование

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

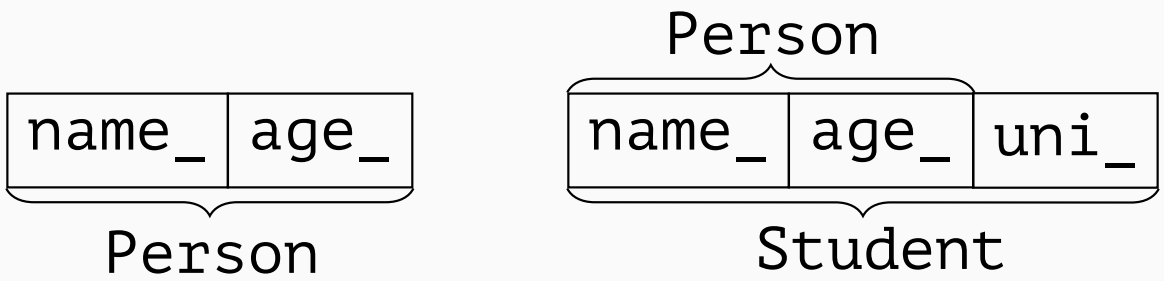
```
struct Person {
    string name() const { return name_; }
    int age() const { return age_; }
private:
    string name_;
    int age_;
};

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)
    {}
};
```

Деструкторы вызываются в обратном порядке.

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");
```

Перекрытие методов

```
struct A {
    void foo(int);
};

struct B : A {
    void foo(long long);
};

int main() {
    B b;
    b.foo(1); // calls B::foo
}
```

Перекрытие методов

```
struct A {
    void foo(int);
};

struct B : A {
    void foo(long long);
    using A::foo;
};

int main() {
    B b;
    b.foo(1); // calls A::foo
}
```

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` $\to$ `int`
     `unsigned char, unsigned short` $\to$ `int`/`unsigned int`
     `float` $\to$ `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 override {
        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 override {return "student";}
};
struct Professor : Person {
  string occupation() const override {
      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() = default;
};
struct Student : Person {
    ...
private:
    string uni_;
};

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

Ключевые слова final и override

- Ключевое слово `final` позволяет запретить дальнейшее наследование или переопределение методов.

```
struct A final {};
struct B : A {}; // error

struct A { virtual void foo() {}; };
struct B : A { void foo() final {}; };
struct C : B { void foo() {}; }; // error
```

- Ключевое слово `override` позволяет провести проверки времени компиляции.

```
struct A { virtual void foo(int) {}; };
struct B : A {
    void foo(double) override {}; // error
};
```

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() = default;
    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

Ещё раз об ООП

Объектно-ориентированное программирование — концпеция программирования, основанная на понятиях объектов и классов.

Основные принципы:

- инкапсуляция,
- наследование,
- полиморфизм,
- абстракция.

Агрегирование vs наследование

- *Агрегирование* — это включение объекта одного класса в качестве поля в другой.
- Наследование устанавливает более сильные связи между классами, нежели агрегирование:
  - приведение между объектами,
  - доступ к `protected` членам.
- Если наследование можно заменить легко на агрегирование, то это нужно сделать.

Примеры некорректного наследования

- Класс `Circle` унаследовать от класса `Point`.
- Класс `LinearSystem` унаследовать от класса `Matrix`.

Принцип подстановки Барбары Лисков

Liskov Substitution Principle (LSP)

> *Функции, работающие с базовым классом, должны иметь возможность работать с подклассами не зная об этом.*

Этот принцип является важнейшим критерием при построении иерархий наследования.

Другие формулировки

- Поведение наследуемых классов не должно противоречить поведению, заданному базовым классом.
- Более простыми словами можно сказать, что поведение наследующих классов не должно противоречить поведению, заданному базовым классом, то есть поведение наследующих классов должно быть ожидаемым для кода, использующего переменную базового типа.

Модификаторы при наследовании

При наследовании можно использовать модификаторы доступа:

```
struct A {};
struct B1 : public A {};
struct B2 : private A {};
struct B3 : protected A {};
```

Для классов, объявленных как `struct`, по-умолчанию используется `public`, для объявленных как `class` – `private`.

Важно: *отношение наследования* (в терминах ООП) задаётся только `public`-наследованием.

Использование `private`- и `protected`-наследований целесообразно, если необходимо не только агрегировать другой класс, но и переопределить его виртуальные методы.

Переопределение private виртуальных методов

```
struct NetworkDevice {
    void send(void * data, size_t size) {
        log("start sending");
        send_impl(data, size);
        log("stop sending");
    }
    ...
private:
    virtual void send_impl(void * data, size_t size)
    {...}
};

struct Router : NetworkDevice {
private:
    void send_impl(void * data, size_t size) {...}
};
```

Реализация чистых виртуальных методов

Чистые виртуальные методы могут иметь определения:

```
struct NetworkDevice {
    virtual void send(void * data, size_t size) = 0;
    ...
};

void NetworkDevice::send(void * data, size_t size) {
    ...
}

struct Router : NetworkDevice {
    void send(void * data, size_t size) {
        // невиртуальный вызов
        NetworkDevice::send(data, size);
    }
};
```

Интерфейсы

Интерфейс — это абстрактный класс, у которого отсутствуют поля, а все методы являются чистыми виртуальными.

```
struct IConvertibleToString {
    virtual ~IConvertibleToString() {}
    virtual string toString() const = 0;
};
```

```
struct IClonable {
    virtual ~IClonable() {}
    virtual IClonable * clone() const = 0;
};
```

```
struct Person : IClonable {
    Person * clone() {return new Person(*this);}
};
```

Множественное наследование

В C++ разрешено множественное наследование.

```
struct Person {};
struct Student : Person {};
struct Worker  : Person {};
struct WorkingStudent : Student, Worker {};
```

Стоит избегать *наследования реализаций* более чем от одного класса, вместо этого использовать интерфейсы.

```
struct IWorker {};
struct Worker : Person, IWorker {};
struct Student : Person {};
struct WorkingStudent : Student, IWorker {};
```

Множественное наследование — это отдельная большая тема.

Дружественные классы

```
struct String {
    ...
    friend struct StringBuffer;
private:
    char * data_;
    size_t len_;
};

struct StringBuffer {
    void append(String const& s) {
        append(s.data_);
    }
    void append(char const* s) {...}
    ...
};
```

Дружественные функции

Дружественные функции можно определять прямо внутри описания класса (они становятся `inline`).

```
struct String {
    ...

    friend void print(String const& s)
    {
        os << s.data_;
    }

private:
    char * data_;
    size_t len_;
};
```

Дружественные методы

```
struct String;
struct StringBuffer {
    void append(String const& s);
    void append(char const* s) {...}
    ...
};

struct String {
    ...
    friend
      void StringBuffer::append(String const& s);
};

void StringBuffer::append(String const& s) {
    append(s.data_);
}
```

Отношение дружбы

Отношение дружбы можно охарактеризовать следующими утверждениями:

- Отношение дружбы не симметрично.
- Отношение дружбы не транзитивно.
- Отношение наследования не задаёт отношение дружбы.
- Отношение дружбы сильнее, чем отношение наследования.

Вывод

Стоит избегать ключевого слова `friend`, так как оно нарушает инкапсуляцию.