Наследование

А. Г. Фенстер, http://info.fenster.name

23 ноября 2009 г.

Конспект семинара №5 по объектно-ориентированному программированию. Пожалуйста, не относитесь к этому как к руководству по C++ или ООП: здесь изложен только самый минимум информации по теме; тем не менее, этого минимума должно хватить для выполнения третьего задания.

О найденных ошибках и неточностях, пожалуйста, пишите мне по адресу `fenster@fenster.name`. Также буду рад получить любые комментарии по поводу этого текста.

В первой и второй частях было введено понятие класса и показано, как определяются конструктор, деструктор и основные операторы. В третьей части было введено понятие шаблона класса. В четвёртой части кратко обсуждались вопросы перегрузки операторов `*` (унарного) и `->`.

В этой части будет обсуждаться понятие наследования — одно из базовых понятий объектно-ориентированного программирования: возможность создавать классы, обладающие свойствами и методами некоторых других классов и расширяющие их.

Для краткости и упрощения изложения мы будем говорить только о наиболее простом случае наследования, при котором класс может иметь лишь один базовый класс, чьи свойства и методы он наследует. В C++ возможно множественное наследование, при котором класс наследует свойства и методы нескольких других классов; мы не будем рассматривать такое наследование и проблемы, возникающие при его использовании.

Наследование данных

Начнём с простого примера: объявим структуру, хранящую информацию об одном человеке (например, фамилию и имя). Будем использовать для хранения строк тип `std::string` (`#include <string>`), он весьма похож на тот класс строк, который вы реализовывали в первом задании.

Итак, имеем простую структуру (можно было бы назвать её классом, но она слишком проста для этого):

```
struct Person /* структура "человек" */
{
    std::string firstname, lastname;  /* фамилия, имя */
};
```

Представим, что нам нужно описать структуру для хранения информации о сотруднике организации: например, его имя, фамилию и отдел, в котором он работает. Мы можем описать структуру с тремя полями (имя, фамилия, отдел), но вместо этого мы *расширим* определение структуры `Person` следующим образом:

```
struct Employee: Person /* структура "сотрудник";
                           расширяет структуру "человек" */
{
    std::string department; /* отдел */
};
```

Теперь любой экземпляр структуры `Employee` будет содержать не только поле `department`, но также и поля `firstname` и `lastname`, *унаследованные* от структуры `Person`. Мы можем обратиться к ним так же, как если бы они были членами структуры `Employee`:

```
Employee e;
e.firstname = "Ivan";
e.lastname = "Petrov";
e.department = "sales";

std::cout << e.firstname << ", " << e.lastname << ", "  <<
                                    e.department << "\n";
```

Объект `e` обладает всеми свойствами типа `Person` и добавляет к ним новые свойства, указанные в описании типа `Employee`.

Структуру, от которой производится наследование (в нашем примере это `Person`), мы будем называть *базовой* (base), а наследующую структуру (`Employee`) — *производной* (derived). Аналогичная терминология будет использоваться и для классов: *базовый класс*, *производный класс*.

Теперь сделаем из базовой структуры эквивалентный ей класс:

```
class Person
{
public:
    std::string firstname, lastname;
};
```

В языке C++ существует правило, по которому при наследовании классов все поля и методы базового класса в производном классе «приобретают» уровень доступа `private`. Соответственно, находясь вне класса `Employee`, обратиться к полям базового класса (`firstname` и `lastname` будет нельзя. Эта проблема решается путём явного указания уровня доступа к базовому классу при наследовании:

```
class Employee: public Person
{
public:
    std::string department;
};
```

Ниже мы чуть подробнее поговорим про уровни доступа к базовому классу, пока же советую всегда при наследовании указывать уровень доступа `public`.

Наследование методов

Наши классы `Person` и `Employee` описаны не очень красиво: обычно не принято выносить данные (поля) в секцию `public` (это подробно обсуждалось в первой части). Место данных — в приватной части класса, а для доступа к ним должны быть описаны соответствующие методы. Перепишем класс `Person`, добавив к нему конструктор для инициализации членов `firstname` и `lastname` и методы для получения их значений:

```
class Person
{
    std::string firstname, lastname; /* теперь приватные */
public:
    Person(std::string f, std::string l): firstname(f),
                                          lastname(l)
                                          { }
    std::string getFirstname() { return firstname; }
    std::string getLastname() { return lastname; }
};
```

Напомним, что синтаксис

```
Person(std::string f, std::string l): firstname(f),
                                      lastname(l)
                                      { }
```

используется для инициализации полей класса при конструировании. Фактически, таким образом указываются аргументы для конструкторов `firstname` и `lastname`. Тело конструктора `Person` оставлено пустым (об этом говорят пустые фигурные скобки `{ }`).

Обратите внимание, что это **не эквивалентно** присваиванию полям значений в конструкторе:

```
Person(std::string f, std::string l) { firstname = f; lastname = l; }
```

В этом случае сначала `firstname` и `lastname` будут созданы при помощи конструктора по умолчанию, затем им будет присвоено значение при помощи оператора присваивания.

Мы описали конструктор и два метода класса `Person`, теперь можно создать экземпляр этого класса и получить значения его полей следующим образом:

```
Person p("Ivan", "Petrov");
std::cout << p.getFirstname() << ", " <<
                p.getLastname() << "\n";
```

Что же происходит при наследовании? Вполне ожидаемо, что методы `getFirstname()` и `getLastname()` становятся доступными для класса `Employee`, т. к. они объявлены как `public` и при наследовании указан такой же уровень доступа к базовому классу. Но непонятно, как можно инициализировать поля `firstname` и `lastname` при конструировании

объекта класса `Employee`, ведь они не являются членами класса `Employee` и обычная инициализация типа `: firstname(f)` здесь не сработает.

Оказывается, при конструировании производного класса можно указывать *аргументы конструктора базового класса*, то есть явно указать, что для вызова конструктора базового класса `Person` необходимо использовать следующие параметры:

```
class Employee: public Person
{
    std::string department;
public:
    Employee(std::string f, std::string l, std::string d):
        Person(f, l), department(d)
        { }
    std::string getDepartment() { return department; }
};
```

Теперь ничто не мешает написать нам такой код:

```
Employee e("Ivan", "Petrov", "sales");

std::cout << e.getFirstname() << ", " <<
             e.getLastname() << ", " <<
             e.getDepartment() << "\n";
```

При этом сначала будет сконструирован объект базового класса `Person` (полям `firstname` и `lastname` будут присвоены значения `Ivan` и `Petrov` соответственно), а уже затем будет выполнен конструктор `Employee`. Деструкторы будут вызываться в обратном порядке: сначала деструктор производного, затем базового класса.

Уровень доступа protected

Наследование может использоваться для того, чтобы изменить поведение базового класса в производных классах. Зачастую при этом нужно иметь доступ к данным, хранящимся в базовом классе. Иными словами, хочется иметь возможность запретить доступ к полю или методу всем, кроме текущего класса и его производных классов.

Возможностей модификаторов `private` и `public` для этого недостаточно, поэтому в языке C++ введён третий уровень доступа, который называется `protected` (защищённый) и делает ровно то, что требуется: защищённый член класса доступен только методам этого класса и всех производных от него классов.[1]

В частности, в классе `Person`, возможно, логичным будет вынести поля `firstname` и `lastname` в секцию `protected:`, чтобы методы класса `Employee` имели к ним доступ «напрямую».

Уровни доступа к базовому классу

Те же ключевые слова — `private`, `public` и `protected` — используются для указания *уровня доступа к базовому классу*. До этого мы писали:

```
class Employee: public Person
```

и выше говорилось, что в этом случае у производного класса будет доступ к публичным полям и методам базового класса. Такой тип наследования можно назвать *открытым*. В случае *закрытого* (`private`) и *защищённого* (`protected`) наследования ситуация будет другой, что показано в таблице:

`class C` `{`	`class C1:` `public C {`	`class C2:` `protected C {`	`class C3:` `private C {`
`private:` `int a;`	недоступно	недоступно	недоступно
`protected:` `int b;`	доступно как `protected`	доступно как `protected`	доступно как `private`

`public:`
`int c;`
`};`
	доступно как
	`public`
	`};`
		доступно как
		`protected`
		`};`
			доступно как
			`private`
			`};`

Легко запомнить правило: приватные члены базового класса в производных классах недоступны; остальные члены доступны с минимальным

[1] А также, конечно, «друзьям» этих классов, но эту тему мы в нашем кратком курсе не затрагиваем.

из двух уровней доступа: их уровня доступа в базовом классе и уровня доступа к базовому классу при наследовании.

Если уровень доступа к базовому классу не указан, то для структур по умолчанию используется уровень `public`, а для классов — `private`.

Использование производных классов вместо базовых и виртуальные методы

Рассмотрим пример. Пусть оба наших класса, `Person` и `Employee`, определяют метод `print`, печатающий известную информацию о человеке. Заметьте, что все данные из области `private` перенесены в `protected`, чтобы стать доступными производным классам.

```
class Person
{
protected:
    std::string firstname, lastname;
public:
    Person(std::string f, std::string l):
                firstname(f), lastname(l) { }
    void print()
    {
        std::cout << "Person: " <<
               firstname << " " <<
                 lastname << "\n";
    }
};

class Employee: public Person
{
protected:
    std::string department;
public:
    Employee(std::string f, std::string l, std::string d):
              Person(f, l), department(d)  { }
    void print()
    {
```

```
        std::cout << "Employee: " <<
                firstname << " " <<
                lastname << ", " <<
                department << "\n";
    }
};
```

Запомните общее правило: в случае открытого наследования **указатель на производный класс может быть использован вместо указателя на базовый класс**. Экземпляр производного класса по сути является объектом базового класса с добавленными новыми полями и методами, поэтому подстановка его вместо базового не вызывает проблем. То же самое правило работает и для ссылок на базовый и производный класс. Обратное, естественно, неверно: **использовать базовый класс вместо производного нельзя**.

Напишем функцию `print_info`, которая принимает ссылку на `Person` и вызывает для неё метод `print`:

```
void print_info(Person &p)
{
    p.print();
}
```

Используем её для печати информации об объекте типа `Person` и объекте типа `Employee`:

```
Person p("Vasily", "Sidorov");
Employee e("Ivan", "Petrov", "sales");

print_info(p); // напечатано Person: Vasily Sidorov

print_info(e); // e используется как Person;
               // напечатано Person: Ivan Petrov
```

Сравните последний вызов с вызовом метода `print` из `e` «напрямую»:

```
e.print(); // e используется как Employee;
           // напечатано Employee: Ivan Petrov, sales
```

Причины такого поведения понятны: функция `print_info` считает, что её аргумент — `Person` и потому вызывает метод `print`, определённый в классе `Person`.

Однако, при описании метода существует возможность указать, что при подстановке производного класса вместо базового следует использовать метод, определённый именно в производном классе. Для этого нужно при объявлении метода в базовом классе указать слово `virtual`.

Класс `Person` будет выглядеть так:

```
class Person
{
    ...
public:
    virtual void print()
    {
        std::cout << "Person: " <<
        firstname << " " <<
        lastname << "\n";
    }
};
```

С этим изменением тот же самый фрагмент кода будет работать по-другому:

```
Person p("Vasily", "Sidorov");
Employee e("Ivan", "Petrov", "sales");

print_info(p); // напечатано Person: Vasily Sidorov

print_info(e); // e используется как Person,
               // но метод print в Person виртуальный:
               // напечатано Employee: Ivan Petrov, sales
```

Обратите внимание, что речь здесь идёт только об указателях и ссылках, то есть о тех случаях, когда не происходит физического копирования объекта. В других случаях эта «магия» не работает:

```
void print_info_copy(Person p)
// передача по значению, т. е. копирование
{
```

```
    p.print();
}

print_info_copy(e);
    // часть объекта e, соответствующая типу Person,
    // копируется в параметр p, всё остальное теряется;
    // напечатано Person: Ivan Petrov
```

Чисто виртуальные функции и абстрактные классы

Бывают ситуации, когда заранее известно, что некоторый виртуальный метод базового класса никогда не будет вызываться для объекта базового класса, потому что во всех случаях нужны будут только методы производных классов. Классическим примером здесь является набор классов, задающих геометрические фигуры.

Пусть определён базовый класс `Figure` (фигура) и два производных от него класса: `Square` (квадрат) и `Circle` (круг). Мы хотим написать функцию, печатающую площадь данной фигуры. Опишем сначала классы:

```
class Figure
{
public:
    virtual double area() { throw "error!"; }
        // не знаем, как считать площадь произвольной фигуры
};

class Square: public Figure
{
    double length;
public:
    Square(double l): length(l) { }
    double area() { return length * length; }
};

class Circle: public Figure
```

```
{
    double radius;
public:
    Circle(double r): radius(r) { }
    double area() { return 3.14159 * radius * radius; }
};
```

и нашу функцию:

```
void print_area(Figure &p) { std::cout << p.area() << "\n"; }
```

Метод `area` виртуальный и функция `print_area` принимает параметр по ссылке, поэтому функции `print_area` можно передавать и круг, и квадрат — площадь будет напечатана верно:

```
Square s(1.0);
Circle c(1.0);

print_area(s); // вызывается Square::area(), печатается 1.0
print_area(c); // вызывается Circle::area(), печатается 3.14159
```

Вызывать функцию `area` для экземпляра класса `Figure` бессмысленно: мы не знаем, что это за фигура, и не можем вычислить её площадь. Поэтому эта функция вызывает исключение (`throw "error!"`). Вообще говоря, бессмысленно даже создавать экземпляр класса `Figure`: он не содержит никаких данных и ничего полезного с ним сделать нельзя. Но он нужен для того, чтобы функция `print_area` была определена в наиболее общем виде.

Проблема «лишнего» метода `area` класса `Figure` решается очень просто: он объявляется *чисто виртуальным*, т. е. виртуальным методом, требующим обязательного определения в производном классе. Это делается путём добавления `= 0;` вместо кода метода:

```
class Figure
{
public:
    virtual double area() = 0;
};
```

Если класс содержит хотя бы один чисто виртуальный метод, то создать экземпляр такого класса нельзя: можно только определить производный класс, реализующий этот метод, и создать экземпляр производного класса.

Класс, не содержащий никаких данных и имеющий только виртуальные методы, называется *абстрактным*. Зачастую при программировании в объектно-ориентированной парадигме пользовательские функции работают именно с абстрактными классами (так же, как функция `print_area`), а при вызове этих функций в параметры подставляются соответствующие производные классы.