Основы ООП

Инкапсуляция

Наследование

Полиморфизм

Лекция #9

Пустовалова О.Г.
доцент. каф. мат.мод.
ИММиКН ЮФУ

Содержание

Основы ООП

Основы ООП

- **Классы и объекты** в C++ являются основными концепциями объектно-ориентированного программирования — ООП.
- **Объектно-ориентированное программирование** — расширение структурного программирования.
- **Классы в C++** — это абстракция описывающая методы, свойства, ещё не существующих объектов.
- **Объекты** — конкретное представление абстракции, имеющее свои свойства и методы.
- Созданные объекты на основе одного класса называются **экземплярами этого класса**.
- Эти объекты могут иметь различное поведение, свойства, но все равно будут являться объектами одного класса.

Основы ООП

Инженер знает, как разработать продукт

Финансовый директор знает, где взять деньги

Коммерческий директор знает, кому продать

не знает, кому продать и где взять на это деньги

не знает, что с ними делать

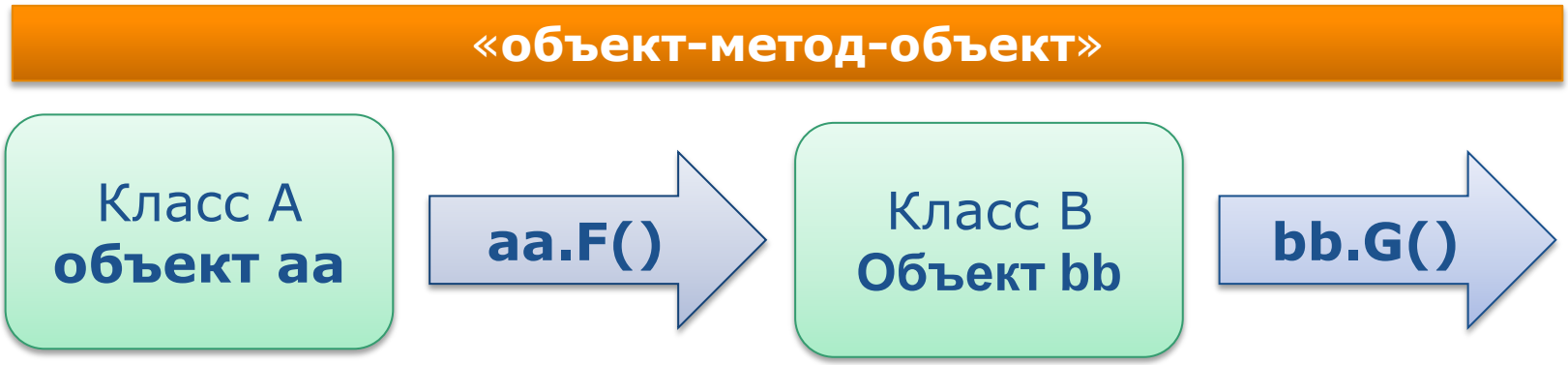
не знает, как сделать продукт

генеральный директор не знает ничего перечисленного, но может заставить работать всех вместе

основной принцип ООП: каждый класс моделирует отдельную **сущность** и работает в пределах своей компетенции

Основы ООП

- Объектно-ориентированный подход заключается в **первичности данных** (объектов) по отношению к алгоритмам (методам) их обработки.
- Причем любая сущность, с которой программист сталкивается при написании программы, должна являться объектом



Основы ООП

- В ООП существует три основных принципа построения классов:

Инкапсуляция — это свойство, позволяющее объединить в классе и данные, и методы, работающие с ними и **скрыть детали реализации от пользователя**.

Наследование — это свойство, позволяющее **создать новый класс-потомок на основе уже существующего**, при этом все характеристики класса родителя присваиваются классу-потомку.

Полиморфизм — свойство классов, позволяющее использовать объекты классов с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Инкапсуляция

Инкапсуляция

В общем случае в разных языках программирования термин «инкапсуляция» относится к одной или обеим одновременно следующим нотациям:

- механизм языка, позволяющий ограничить доступ одних компонентов программы к другим;
- языковая конструкция, позволяющая связать данные с методами, предназначенными для обработки этих данных.

Инкапсуляция

Пренебрегая формализмом и способствуя интуитивному восприятию, инкапсуляцию можно определить с помощью латинского *in capsula* — размещение в оболочке, изоляция, **закрытие чего-либо инородного с целью исключения влияния на окружающее**, обеспечение доступности главного, выделение основного содержания путём помещения всего мешающего, второстепенного в некую условную капсулу (чёрный ящик).

Инкапсуляция

- Другими словами, концепция инкапсуляции призвана обеспечивать **безопасность проектирования** и реализации программного обеспечения на основе локализации манипулирования объектом в областях его полей и методов.

- Иначе говоря, **свойствами объекта можно оперировать исключительно посредством его методов**. Это замечание касается как свойств, явно определенных в описании объекта, так и свойств, унаследованных данным объектом от другого (других).

Инкапсуляция

- В то же время в практике программирования **степень инкапсуляции объекта (в широком значении этого слова) определяется его описанием**, а также описанием порождающих его объектов (в действительности это, как правило, описания базовых и производных классов).

- В этой связи в языках объектно-ориентированного программирования вводится **понятие области видимости как степени доступности произвольного языкового объекта**.

Инкапсуляция

```
class Box
{
public:
        double getVolume(void)
        {
                return length * width * height;
        }
private:
        double length;      // Length of a box
        double width;       // Width of a box
        double height;      // Height of a box
};
```

Наследование

Наследование

- **Наследование классов** — очень мощная возможность в объектно-ориентированном программировании.
- Оно позволяет создавать **производные классы** (классы наследники), взяв за основу все методы и элементы базового класса (класса родителя).
- Таким образом экономится масса времени на написание и отладку кода новой программы.
- Объекты производного класса свободно могут **использовать всё**, что создано и отлажено **в базовом классе**.
- При этом, мы можем в производный класс, **дописать необходимый код** для усовершенствования программы: добавить новые элементы, методы и т.д..
- Базовый класс **останется нетронутым**.

Наследование

Базовый класс

Свойства и методы

РОДИТЕЛЬ

Дочерний класс

Свойства и методы родителя
+ СВОИ

ПОТОМОК

Наследование. Пример 1. Часть 1

```
#include <iostream>
using namespace std;
// базовый класс
class FirstClass
{
        // спецификатор доступа к элементу value
        protected:
                int value;
        public:
                FirstClass(){ value = 0;} // конструктор

                FirstClass(int n){value = n;} // конструктор

                void show_value(){cout << value << endl;}
};
```

Наследование. Пример 1. Часть 2

```
// производный класс
class SecondClass : public FirstClass
{
        public:
        // конструктор класса SecondClass
        //вызывает конструктор класса FirstClass
        SecondClass() : FirstClass(){}

        // inputS передается в конструктор
        // с параметром класса FirstClass
        SecondClass(int nS) : FirstClass(nS) {}

        // возводит value в квадрат.
        // Без спецификатора доступа protected
        //эта функция не могла бы изменить значение value
        void ValueSqr() { value *= value;}
};.
```

Наследование. Пример 1. Часть 2. Пояснения

- Важной особенностью производного класса, является то, что хоть он и может использовать все методы и элементы полей **protected** и **public** базового класса, но он не может обратиться к конструктору с параметрами.

- Если конструкторы в производном классе не определены, при создании объекта сработает конструктор без аргументов базового класса.

- А если нам надо сразу при создании объекта производного класса внести данные, то для него необходимо определить свои конструкторы.

Наследование. Пример 1. Часть 3

```
int main() {

        FirstClass q1(3); // объект базового класса
        cout << "q1 = ";
        q1.show_value();

        SecondClass q2(4); // объект производного класса
        cout << "q2 = ";
        q2.show_value(); // вызов метода базового класса

        q2.ValueSqr(); // возводим value в квадрат
        cout << "q2^2 = ";
        q2.show_value();

// базовый класс не имеет доступа к методам производного класса
        //q1.ValueSqr();

        cout << endl;
return 0;
}
```

Наследование. Спецификаторы доступа

Доступ из	private	protected	public
тела класса	открыт	открыт	открыт
производных классов	закрыт	открыт	открыт
внешних фун. и клас.	закрыт	закрыт	открыт

Наследование. Спецификаторы доступа

- С помощью спецификатора **public** можно определить общедоступные открытые члены классы, которые доступны извне и их можно использовать в любой части программы.

- С помощью спецификатора **private** можно определить закрытые переменные и функции, которые можно использовать только внутри своего класса.

- Однако иногда возникает необходимость в таких переменных и методах, которые были бы доступны классам-наследникам, но не были бы доступны извне. И именно для определения уровня доступа подобных членов класса используется спецификатор **protected**.

Наследование. Спецификаторы доступа

Если вы создаёте класс, который в дальнейшем планируете использовать, как **базовый**, то объявляйте в нём поле **protected** вместо **private**.

Иначе объекты производного класса не смогут обращаться к элементам базового.

Наследование. Что надо помнить

- **Наследование** — это определение **производного класса**, который может обращаться ко всем элементам и методам базового класса за исключением тех, которые находятся в поле private;

- Производный класс еще называют **потомком** или **подклассом**, а **базовый — родитель** или **надкласс**;

- **Производный класс** имеет доступ ко всем элементам и методам базового класса, а базовый класс может использовать только свои собственные элементы и методы.

Наследование. Что надо помнить

Синтаксис определения производного класса

```
class   Имя_Производного_Класса :
          спецификатор_доступа     Имя_Базового_Класса
                                          { /*код*/ };
```

```
// производный класс
class SecondClass : public FirstClass
{
                /*код*/
};.
```

Наследование. Что надо помнить

- В производном классе необходимо **явно определять свои конструкторы**, деструкторы и перегруженные операторы присваивания из-за того, что они не наследуются от базового класса.

- Но их можно вызвать явным образом при определении конструктора, деструктора или перегрузки оператора присваивания производного класса, например таким образом (для конструктора):

```
Конструктор_Производного_Класса(/*параметры*/) :
        Конструктор_Базового_Класса( /*параметры*/) { }
```

```
// inputS передается в конструктор
        // с параметром класса FirstClass
        SecondClass(int nS) : FirstClass(nS) {}
```

Наследование. Перегруженные методы класса-потомка

- если в классе родителе и в его классах потомках встречаются методы с **одинаковым именем**, то для объектов класса потомка компилятор будет использовать методы именно класса потомка.

- Перегруженные методы класса потомка, **могут вызывать методы класса родителя**. В таком случае важно помнить, что необходимо правильно определить область действия **с помощью оператора разрешения области видимости ::**

Наследование. Перегруженные методы класса-потомка

Наглядно, если бы мы перегрузили в классе SecondClass функцию show_value() — это выглядело бы так:

```
void show_value()
{
        if (value != 0)
                FirstClass::show_value();
        else cout <<" *** "<<value << endl;
}
```

Эта запись указывает компилятору — если значение value не равно нулю - вызвать метод show_value() класса FirstClass.

Наследование. Выводы

- **Наследование** помогает экономить массу времени на написание и отладку кода с нуля.
- Используя наследование, можно использовать уже готовый и отлаженный код и подстраивать его под новые задачи.
- При этом программа будет занимать намного меньше строк, что значительно улучшит её читабельность.

Запрет наследования

- Иногда наследование от класса может быть нежелательно.
- И с помощью спецификатора **final** можно запретить наследование:

```
class User final
{
};
```

После этого нельзя унаследовать другие классы от класса User.
В такой ситуации возникнет ошибка:

```
class VipUser : public User
{
};
```

class User

тип класса final не может использоваться в качестве базового класса

Наследование. Пример 2

Базовый класс

Фермер
Unit

ЗДОРОВЬЕ

Дочерний класс

Воин
Soldier

ЗДОРОВЬЕ

УРОН

СКОРОСТЬ

Дочерний класс

Лошадь
Soldier

ЗДОРОВЬЕ

СКОРОСТЬ

Дочерний класс

Всадник
Horseman

ЗДОРОВЬЕ

УРОН

СКОРОСТЬ

Наследование. Пример 2. Часть 1

```
#include<iostream>
using namespace std;
//базовый класс фермера
class Unit {

protected:
	int health; //здоровье
public:

	void showHealth() {
		cout << "Unit health:" << health << endl; };

	Unit() :health(10) {}//конструктор по умолчанию

	Unit(int a) :health(a) {}//конструктор с параметром

};
```

Наследование. Пример 2. Часть 2

```
// класс воин
class Soldier: virtual public Unit
{
protected:
        int damage;// урон
        int speed; // скорость
public:
        void showDamage() {
                cout << "Sodier damage:" << damage << endl; }

        void showSpeed() {
                cout << "Soldier speed:" << speed << endl; }

        // конструктор по умолчанию
        Soldier() :damage(20),speed(3) {}
};
```

Наследование. Пример 2. Часть 3

```
// класс лошадь
class Horse: virtual public Unit
 {
 protected:
        int speed; // скорость

 public:
        void showSpeed() {
                cout << "Horse speed:" << speed << endl; }

        Horse() :speed(30) {} // конструктор по умолчанию
};
```

Наследование. Пример 2. Часть 4

```
// класс всадник
// чтобы не наследовались 2 копии Unit используется virtual
// так как Horse и Soldier основываются на Unit
class Horseman: public Horse, public Soldier
{
private:
	int speed;

public:
	void showSpeed() {
		cout << "Horseman speed:" << speed << endl; }

	// скорость как у лошади
	Horseman() :speed(Horse::speed) {}
};
```

Наследование. Пример 2. Часть 5

```
int main()
{
	Horseman lancelot;

	// работает конструкторы по умолчанию
	lancelot.showHealth(); //10 - здоровье как у фермера
	lancelot.showDamage(); //20 - урон как у воина
	lancelot.showSpeed();  //30 - скорость как у лошади

return 0;
}
```

Полиморфизм

Полиморфизм

Полиморфизм - иметь много форм

- Как правило, полиморфизм происходит, когда есть иерархия классов, и они связаны по наследству.

- С ++ полиморфизм означает, что вызов функции члена вызовет другую функцию, которая будет выполняться в зависимости от типа объекта, который вызывает функцию.

Полиморфизм

Полиморфизм - иметь много форм

- С помощью атрибута **virtual** указывается, что метод может быть переопределен в производных классах.
- Атрибут достаточно указать один раз при объявлении в базовом классе.
- Если виртуальный метод был объявлен, но не определен, то класс называется абстрактным, а метод чисто виртуальным.
- При этом методу присваивается значение 0.
- Естественно объекты абстрактных классов создавать нельзя.

Полиморфизм. Пример 1

Написать иерархию классов, описывающих служащих в компании.

Она должна состоять из абстрактного базового класса Employee и производных от него классов Manager, Agent и Worker.

Базовый класс должен иметь чисто виртуальный метод рассчета заработной платы, переопределенный в каждом из производных классов.

- заработная плата Manager постоянна и равна 13000,
- заработная плата Agent определяется как оклад 5000 + 5% объема продаж, который хранится в специальном поле класса Agent,
- заработная плата Worker определяется как 100×число отработанных часов, которое также хранится в отдельном поле (классы Agent и Worker должны иметь конструкторы по вещественному числу, устанавливающие значение своего поля).

В основной программе завести массив из 9 указателей на Employee и заполнить его указателями на объекты производных классов (первые 3 — Manager, следующие 3 — Agent и последние 3 — Worker).
Вывести на экран величину заработной платы всех 9 служащих.

Полиморфизм. Пример 1. Часть 1

```
#include <iostream>
using namespace std;

struct Employee {
        // виртуальный деструктор
       virtual ~Employee() {}
        // виртуальный конструктор
       virtual double salary() = 0; };

       // класс Manager.
       //Поскольку зарплата Manager постоянна,
       // поля не нужны.
       struct Manager : Employee {
              double salary() { return 13000; }
       };

       double salary() { return 100 * time; }
};
```

Полиморфизм. Пример 1. Часть 2

```
// класс Agent
// Поскольку для вычисления его зарплаты требуется знать объем
// продаж, что является характеристикой Agent, то уместно
// завести соответствующее поле в классе Agent.
struct Agent : Employee {
        // поле
        double amount;
        // конструктор
        Agent(double a) : amount(a) {}
        // зараплата
        double salary() { return 5000 + 0.05*amount; }
};

struct Worker : Employee { double time;
        Worker(double t) : time(t) {}
        double salary() { return 100 * time; }
};
```

Полиморфизм. Пример 1. Часть 3

```
int main() {
setlocale(0,"" );
        Employee *M[9];
        M[0] = new Manager;
        M[1] = new Manager;
        M[2] = new Manager;
        M[3] = new Agent(100000);
        M[4] = new Agent(110000);
        M[5] = new Agent(120000);
        M[6] = new Worker(140);
        M[7] = new Worker(150);
        M[8] = new Worker(160);

for (int i = 0; i<9; i++) {
        cout << "Зарплата " << i << "-го сотрудника составляет "
        << M[i]->salary() << " р." << endl;
        delete M[i]; }
}
```

```
Зарплата 0-го сотрудника составляет 13000 р.
Зарплата 1-го сотрудника составляет 13000 р.
Зарплата 2-го сотрудника составляет 13000 р.
Зарплата 3-го сотрудника составляет 10000 р.
Зарплата 4-го сотрудника составляет 10500 р.
Зарплата 5-го сотрудника составляет 11000 р.
Зарплата 6-го сотрудника составляет 14000 р.
Зарплата 7-го сотрудника составляет 15000 р.
Зарплата 8-го сотрудника составляет 16000 р.
```

Полиморфизм. Пример 2. Часть 1

```
#include<iostream>
using namespace std;

class Animal {
public:
	void Info() { cout << "Animal" << endl; }
	virtual void Say() { cout << "Nothing to say" << endl; }
};

class Cat : public Animal {
public:
	void Info() { cout << "Cat" << endl; }
	virtual void Say() { cout << "Meow" << endl; }
};

class Dog : public Animal {
public:
	void Info() { cout << "Dog" << endl; }
	virtual void Say() { cout << "Bow-wow" << endl; }
};
```

Полиморфизм. Пример 2. Часть 2

```
int main(){

Dog *dog1 = new Dog();
Animal *dog2 = new Dog();

        // Невиртуальный метод - вызовется метод класса,
        // указанного у переменной
        dog1->Info(); // напишет Dog
        dog2->Info(); // напишет Animal

        //Виртуальный метод - вызовется метод класса,
        // который в нём реализован
        dog1->Say(); // напишет Bow-wow
        dog2->Say(); // напишет Bow-wow

cout << endl;
return 0;
}
```

Полиморфизм. Пример 2. Часть 3

```
int main(){

        Dog *dog = new Dog();
        Cat *cat = new Cat();
        Animal *animal = new Animal();

// заполним массив указателей разными животными
        Animal *animals[3];
        animals[0] = dog;
        animals[1] = cat;
        animals[2] = animal;

// вызовется правильный метод
// у невиртуальных методов так сделать нельзя!
// Полиморфизм в действии
        for (int i = 0; i<3; i++) animals[i]->Say();

return 0;
}
```

```
Bow-wow
Meow
Nothing to say
```

Спасибо за внимание!