ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»
Институт машиностроения, материалов и транспорта
Высшая школа машиностроения

Конспект лекции «Наследование в ООП»

по дисциплине «Программирование на языках высокого уровня»

Выполнил студент 3331506/90401 _____ Семенов Н.С.
(группа) (подпись) (ФИО)

Преподаватель _____ Ананьевский М.С.
(подпись) (ФИО)

«____» _____2021 г.

Санкт-Петербург
2021

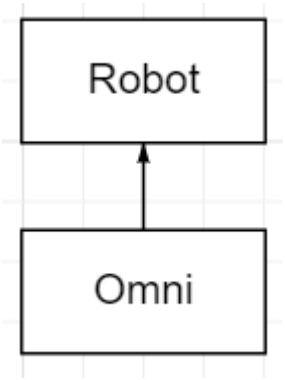
Введение

При наследовании наследнику передаются свойства родителя. Пример на основе классов Robot и Omni:

```
class Robot{
public:
    string name;
public:
    Robot(string new_name=""):name(new_name){};
    action();
    sound();
}

class Omni: public Robot{
public:
    string type;
public:
    Omni(string new_name="");
    action();
    sound();
};

int main{
    Omni omni("Omni");
    Omni noname;
};
```



Порядок вызова конструкторов:

```
Robot();
Omni("omni");
```

При создании объектов неявно вызывается конструктор родителя (не имеет аргументов), поэтому при наследовании в коде должен быть создан конструктор без аргументов, т.н. «конструктор по-умолчанию».

Если при создании Omni хотим вызвать конкретный конструктор, то пишем через двоеточие:

```
public:
    Omni(string new_name=""): Robot(new_name){};
```

Модификаторы доступа

public – доступны снаружи класса;

protected – доступны снаружи класса (но с ограничениями);

private – доступны только изнутри класса.

Доступ при наследовании:

```
class Omni: public Robot{  //При наследовании public
    public -> public
    protected -> protected
    private -> not available
};

class Omni: private Robot{  //При наследовании private
public -> public
protected -> private
private -> not available
};

class Omni: protected Robot{  //При наследовании protected
public -> protected
protected -> protected
private -> not available
};
```

Вывод: при наследовании происходит изменение доступа. В наследнике можно переписать уровень доступа (однако если поля нет, то и переписать нельзя, т.е. private не перенесётся).

Изменим доступ параметра родителя в рамках наследника:

```
class Omni: public Robot{
public:
    string type;
public:
    Omni(string new_name="");
protected:
    using Robot::action;
};
```

Так action перешёл в рамках Omni из private в protected.

Удаление лишнего функционала в рамках наследника:

```
class Omni: public Robot{
public:
    string type;
public:
    Omni(string new_name="");
    action()=delete; //Скрываем (удаляем) метод из класса
};
```

Присваивание объекту класса Robot части от Omni:

```
Omni omniMove ("omniMove");
Robot creature ("Y408T");
creature = omniMove;

/* В переменную creature скопируется часть переменной
 omniMove, которая относится к Robot (в том случае,
 если мы не перегрузим оператор) */
```

Создание ссылок (указателей):

```
Omni omniMove ("omniMove");
Robot *creature01 = &omniMove; //указатель
Robot &creature02 = omniMove; //псевдоним
```

Вызов функций при наследовании

Основываясь на использовании предыдущего кода:

```
omniMove.action(); // вызов action из Omni
creature01 -> action(); // вызов action() из Robot
creature02.action(); // вызов action() из Robot
// оба creature являются объектами Robot

void action_two_times (Robot &creature){
    creature.action;
    creature.action;
}
// В эту функцию можно передать объект класса Omni, т.к. он наследник.
```

Понижающее приведение типов

Можно представить указатель на класс Robot как указатель на класс Omni (если он действительно указывает на объект класса Omni).

```
Omni omniMove ("omniMove");
Robot *creature=&omniMove;

(dynamic=cast<Omni>(creature)).action();
```

где Omni – тип, creature – объект (указатель), action() – вызываемая функция (из Omni).

Так же выполняется и повышающее приведение типов.

Виртуальные функции

Когда мы пишем функцию в классе наследника (action(), sound() …) с тем же именем, то, фактически, мы её переопределяем. При передаче параметра Omni в функцию *void action_two_times (Robot &creature){…}* будем использовать action() из Robot, что обусловлено приведением типов. Мы же, однако, хотим, чтобы action() вызывался для Omni. Для этого требуется объявить эту функцию виртуально.

```
class Robot{
    ...
public:
    virtual action();
};

class Omni: public Robot{
    ...
    virtual action();
};
```

Если функция виртуальная, то она будет общей для всех классов. Следовательно, при любом вызове как Robot (родителя), так и Omni (наследника) будет вызываться одна и та же функция action(). Соответственно, функция должна быть виртуальной.

Виртуальные функции – это позднее связывание, поэтому они работают слегка медленнее.

В частности, деструкторы должны быть виртуальными, если имеется наследование, чтобы вызывался тот, который предназначен для данного объекта.

Переопределение функции virtual должно быть с теми же аргументами и типом. Чтобы не ошибиться, можно использовать вспомогательное слово *override*:

```
virtual action() override;
```

Если данные не совпадут, компилятор отловит ошибку. Слово *final* запрещает в дальнейшем наследовать класс и переопределять функцию:

```
virtual action() override final;
```

Так же можно и с классом:

```
class Omni final: public Robot{...};
```

Интерфейс/абстрактные классы

Идея интерфейса: пишем, что класс должен делать (методы, переменные) в том виде, чтобы код работал (компилировался), но без фактической реализации.

Абстрактный класс – тот, у которого есть хотя бы одна абстрактная функция

```
class Robot{
public:
    string get_name(){};
    virtual void action()= 0; // функция является абстрактной
};
```

Объекты абстрактного класса не создаются (ошибка компиляции). В наследниках абстрактные классы обязательно должны быть определены, иначе класс

тоже будет являться абстрактным. Можно перед названием добавлять I, т.к. интерфейс.

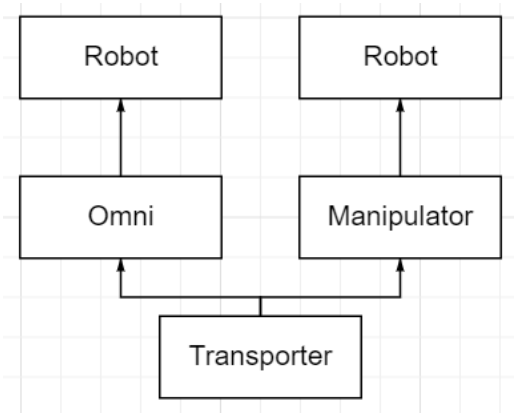
Множественное наследование

```
class Robot{
    string name;
};

class Omni: public Robot{
    int motor_count;
};

class Manipulator: public Robot{
    int motor_count;
};

class Transporter: public Omni, public  Manipulator{
};
```



Множественное наследование рекомендовано использовать только в случаях «класс + интерфейс» или «интерфейс + интерфейс».

Возникшие проблемы:

- Два экземпляра Robot внутри класса;
- Два одинаковых поля motor_count.

Решение:

При наследовании не должно быть неопределённости. То есть не должно быть одинаковых полей у непосредственных родителей и родителей вообще. Поэтому поле motor_count нужно вычеркнуть из Omni и из Manipulator.

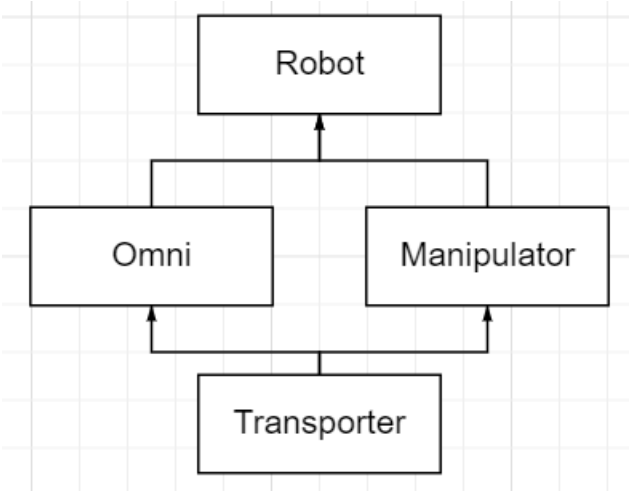
Создадим виртуальный базовый класс:

```
class Robot{
    string name;
    int motor_count;
};
```

```
class Omni: virtual public Robot{
    ...
};

class Manipulator: virtual public
Robot{
    ...
};

class Transporter: public Omni,
public  Manipulator{
};
```



Дружественные функции и классы

```
class Graph{
private:
    Node* Root;
public:
    Node* Search(Node* node);
};

class Node{
private:
    void *data;
    std::list<Node*> neighbors;
    friend class Graph;
};
/* теперь из class Graph будут видны поля и методы
 * class Node. Следовательно, это почти как один класс.*/
```

friend => дружба односторонняя. Node is Field of Graph, но не наоборот. Из Node private методы и поля Graph недоступны.

Также справедливо:

A – friend of B;

B – friend of C;

But A – no friend of C!!

Дружественным можно сделать отдельный метод:

```
class Graph{
private:
    Node* Root;
public:
    Node* Search(Node* node);
};

class Node{
private:
    void *data;
    std::list<Node*> neighbors;
    friend Node* Graph::Search(Node* node);
};
/* Search - метод другого класса Node, т.е. через него
 * доступны private поля от Node.*/
```

Анонимные объекты

```
Omni ("omniMove");
/* Создание анонимного объекта. Он временный.
 * На него нельзя получить ссылку. Однако
 * его можно переименовать.*/

Omni KUKA_OmniMove_1 = Omni ("omniMove");
Omni ("omniMove").action(); // Что-то сделал и исчез.
```

Наследование (опорные пункты):

1. Модификаторы доступа при наследовании;
2. Порядок вызова конструкторов при наследовании;
3. Порядок вызова деструкторов при наследовании;
4. Понижающее приведение типов (к родителю);
5. Повышающее приведение типов (к наследнику);
6. Обрезка объектов (Robot = Omni);

7. Virtual-методы (virtual деструктор);
8. Абстрактный класс (интерфейс);
9. Множественное наследование (виртуальные классы);
10. Дружественные методы/классы;
11. Анонимные объекты.