ООП

Классы в C++

Кафедра ИВТ и ПМ

2018

План

Outline

Прошлые темы

- Опишите парадигму ООП
- Чем она отличается от парадигмы процедурного и модульного программирования?
- Из каких элементов строится программа написанная согласно парадигме объектно-ориентированного программирования ?
- Что такое класс?
- Что такое объект?
- Чем отличается класс от объекта?
- Что такое поле класса?
- Что такое метод класса?

Прошлые темы

- Для чего нужен this?
- Какие модификаторы доступа могут применяться к атрибутам класса?

Outline

Описание класса в C++

```
class ClassName {
private:
  // закрытые члены класса
  // рекомендуется для описания полей
public:
 // открытые (доступные из вне) члены класса
 // рекомендуется для описания интерфейса
protected:
  // защищенных члены класса
  // доступны только наследникам

// дружественные функции и классы
// модификатор доступа не важен
friend заголовок-функции;
friend имя_класса;
};
```

Объекты и обращение к методам

```
class MyClass {
    float _x;
    public:
        void foo() const { cout << "foo" << endl;}
        float x() const {return _x;}
};
int main(){
    MyClass c; // статическое создание объекта
    MyClass *c1 = new MyClass(); // динамическое создание объекта
    const unsigned n = 10;
    MyClass cc[n]; // массив из объектов
    float s = 0;
    for (unsigned i = 0; i<n; i++){
        cc[i].foo();
        s = s + cc[i].x();}
    vector<MyClass*> v(n); // вектор из указателей на MyClass
    for (unsigned i =0; i<n; i++)
        v[i] = new MyClass();
    s = 0;
    for (unsigned i =0; i<n; i++)
        s =  s + v[i].x();
}
```

Outline

Конструктор

Конструктор — это особый метод, инициализирующий экземпляр своего класса.

```
class MyClass{
    float x, y;
    public:

        // Это конструктор
        MyClass(){
            x = 0;
            y = 42;
            cout << "new object";}
};


...

MyClass o1;                    // new object
MyClass o2 = new MyClass(); // new object
```

Конструктор

- Имя конструктора совпадает с именем класса[1].
- Тип возвращаемого значения не указывается - конструктор ничего не возвращает
- У конструктора может быть любое число параметров.
- У класса может быть любое число конструкторов.
- Конструкторы могут доступными (public), защищенными (protected) или закрытыми (private).
- Если не определено ни одного конструктора, компилятор создаст конструктор по умолчанию, не имеющий параметров (а также некоторые другие к. и оператор присваивания)

[1] конструктор в python называется `__init__`

Деструктор

Деструктор — специальный метод класса, служащий для деинициализации объекта (например освобождения памяти).

```
class MyClass{
    float x, y;
    public:

        // Конструктор
        MyClass();

        // Деструктор
        ~MyClass();
};
```

Деструктор

- Деструктор - метод класса
- Объявление деструктора начинается с ~
- У деструкторов нет параметров и возвращаемого значения.
- В отличие от конструкторов деструктор в классе может быть только один.
- Деструктор вызывается *автоматически* при удалении объекта
- Если деструктор не определён, то он будет создан компилятором
- Такой деструктор не будет выполнять никакой работы
- Деструкторы как правило нужны если объекту необходимо освободить ресурсы, например закрыть файл; освободить память, выделенную вручную и т.п.

Деструктор. Пример

```
class MyClass{
    float x, y;
    public:
        // Конструктор
        MyClass();
        // Деструктор
        ~MyClass() {cout << "I'm Finished"; }
};

int main(){
MyClass c1, c2;

if ( 1 ){
    MyClass c3;}
// вызов деструктора c3;

cout << "End.";
// вызов деструторов c1 и c2
}
```

Outline

Конструкторы

- конструктор умолчания (default constructor)
- конструктор преобразования (conversion constructor)
- конструктор с двумя и более параметрами (parameterized constructors)
- конструктор копирования (copy constructor)
- конструктор перемещения (move constructor)

Конструкторы

Компилятор выбирает тот конструктор, который удовлетворяет ситуации по количеству и типам параметров.

В классе не может быть двух конструкторов с одинаковым набором параметров.

Конструктор по умолчанию (Default constructor)

`MyClass()`

- Не имеет параметров.
- Может быть только один.
- Может отсутствовать.
- Создаётся компиляторам, если отсутствует

Когда вызывается

```
class MyClass {...};
...

MyClass c0 = MyClass();
MyClass c1;
MyClass cv[16];       // к. будет вызван 16 раз
list<MyClass> cl(10) // к. будет вызван 10 раз
```

Конструктор с параметрами (Parametrized constructor)

- Принимает несколько параметров

Общий вид:

```
MyClass(T1 t1, T2 t2, T3 t3, .... )
```

T1, T2, T3, ... - некоторые типы

Конструктор преобразования (Conversion constructor)

Общий вид:

`MyClass(T t)`

T - некоторый тип

- Принимает один параметр
- Тип параметра должен отличатся от самого класса
- Такой конструктор как бы преобразует один тип данных в экземпляр данного класса
- Может вызываться при инициализации объекта значением принимаемого типа

  `MyClass c = t`

Конструктор с параметрами (parametrized constructor). Пример

```
class Point{
    float _x, _y;
    public:
        Point() { _x = 0; _y = 0; }

        Point(float x) { _x = x; }

        Point(float x, float y){
                _x = x;
                _y = y;}

        Point(const vector<float> &v){// v - вектор из двух значений
            if (v.size() == 2){_x = v[0]; _y = v[1];}
            else throw "Vector Size Error";}
        // ...
};
```

Пример

```
int main(){
    Point p1;                       // к. по умолчанию
    Point p11 = Point();            // к. по умолчанию (явный вызов)
    Point *pp1 = new Point();       // к. по умолчанию

    Point p2(2);                    // к. преобразования
    Point *pp2 = new Point(42); // к. преобразования

    Point p3(1.5, -1);              // к. с параметрами
    Point *pp3 = new Point(-10.7, 127.2); // к. с параметрами

    vector<float> v = {1,2};
    Point p4(v);                    // к. преобразования
    Point *pp4 = new Point(v);  // к. преобразования

    // Явный вызов конструкторов
    Point pa[3] = {Point(), Point(), Point(2.3)};
}
```

Пример

```
int main(){
    Point *pp5;           // к. не вызывается
    Point *pp6 = &p1;     // к. не вызывается
    Point *pp7 = &p3;     // к. не вызывается
    vector<Point* > vp0;// к. не вызывается

    Point p5 = 5;         // к. преобразования
    Point p6 = v;         // к. преобразования

    Point pp[3] = {0.2, 3, -4.2}; // к. преобразования
    vector<Point> vp3 = {0.2, 3, -4.2};// к. преобразования

    vector<Point> vp;             // к. не вызывается
    vector<Point> vp2(10);        // к. по умолчанию
}
```

Конструктор копирования (copy constructor)

- Один параметр: ссылка на экземпляр данного класса
- Необходим, если копировать

```
MyClass(MyClass &c)
```

Конструктор перемещений (move constructor)

```
MyClass(MyClass &&c)
```

Outline

Оператор присваивания копированием (assignment operator)

```
MyClass& operator=(MyClass& data)
```

- используется для присваивания одного объекта текущему (существующему)
- генерируется автоматически компилятором если не объявлен
- сгенерированный компилятором, выполняет побитовое копирование
- должен очищать поля цели присваивания (и правильно обрабатывать самоприсваивание)

Оператор присваивания перемещением (move assignment operator)

```
MyClass& operator = (const MyClass &c)
```

- используется для присваивания *временного* объекта существующему
- "забирает"временный объект "в себя"; временный объект перестаёт существовать
- генерируется автоматически компилятором если не объявлен
- сгенерированный компилятором, выполняет побитовое копирование
- должен очищать поля цели присваивания (и правильно обрабатывать самоприсваивание)

Когда вызывается?
Когда существующему объекты присваиваю значение временного объекта.

Правило пяти

Если класс или структура определяет один из следующих методов, то они должны явным образом определить все методы:

- Конструктор копирования
- Конструктор перемещения
- Оператор присваивания копированием
- Оператор присваивания перемещением
- Деструктор

Спецификаторы default и delete

Спецификаторы **default** и **delete** заменяют тело метода.

Спецификатор **default** означает реализацию по умолчанию (компилятором). Может быть применён только к конструкторам, деструктору и операторам присваивания.

Спецификатором **delete** помечают те методы, работать с которыми нельзя.

Спецификаторы default и delete

```
class Foo{
public:
    Foo() = default;
    Foo(const Foo&) = delete;
    Foo operator = (const Foo& f) = delete;
};

...

Foo o1, o2; // вызов констр. созданного компилятором
o1 = o2; // Ошибка компиляции! Оп-р присваивания запрещён.
Foo o3(01); // Ошибка компиляции! Констр. копирования запре
```

Вопросы

- Зачем нужны конструкторы?
- Как запретить создание объекта на основе уже существующего?
- Как запретить любой другой способ создания объекта?
- Зачем нужны конструкторы перемещения? В чём их отличие от к. копирования?
- Когда вызывается конструктор, а когда оператор присваивания?
- Что если не описать ни одного конструктора?
- Что если не описать ни одного оператора присваивания?

Outline

Перегрузка операторов (operator overloading)

```
Type operator opr ( parameters );
```

type - тип возвращаемого значения
opr - обозначение оператора, например * или =
parameters - параметры, описываются также как и для метода

Число параметров функции должно соответствовать арности оператора. Например для бинарных операторов параметра два.

Когда перегруженный оператор является методом класса, тип первого операнда должен быть указателем на данный класс (всегда *this), а второй должен быть объявлен в списке параметров.
Операторы в C и C++

Перегрузка операторов

```
class T{
...
    public:
        T operator+ (const T&) const { ... }
};


...

T a,b,c;

// этот код будет транслирован компилятором в
c = a + b;

// этот
c = a.operator+(b);
```

Перегрузка операторов (overloading)

Когда оператор делать методом, а когда дружественной функцией?

Унарные операторы и бинарные операторы типа “X=” рекомендуется реализовывать в виде методов класса, а прочие бинарные операторы — в виде дружественных функций.
Так стоит делать потому, что оператор-метод всегда вызывается для левого операнда.

Outline

Outline

Наследование (Inheritance)

Наследование - построение новых классов на основе уже существующих.

Базовый класс (предок) — класс на основе которого строится определение нового класса - **производного класса (потомка)**.

Наследование? Зачем?

Опишем класс для вектора $V = (V_x, V_y)$ на плоскости

```
// класс - вектор на плоскости
class Vector2D{
    float _x, _y;
public:
    Vector2D();
    void setX(float x) {_x = x;}
    void setY(float y) {_y = y;};
    float x() const {return _x;}
    float y() const {return _y;}
    float abs() const {return sqrt(_x*_x + _y*_y);}
};
```

Что если потребуется создать класс для представления вектора $V = (V_x, V_y, V_z)$? Придётся писать часть кода заново?

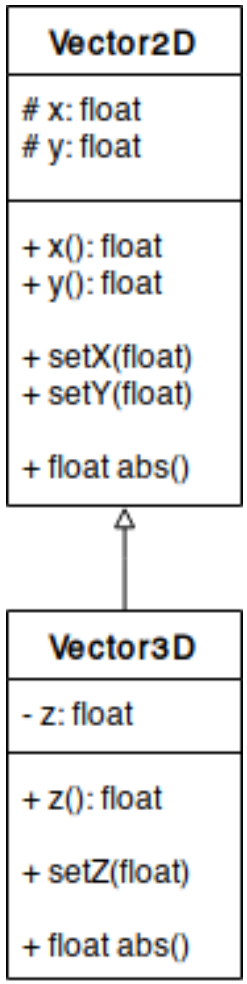
Наследование? Зачем?

Если потребуется создать класс для представления вектора $V = (V_x, V_y, V_z)$ то новый класс можно построить на основе старого, в котором уже будут методы и поля старого класса.

Наследование позволяет построить новый класс на основе имеющегося, *добавив* в него новые поля и методы.

Наследование

UML диаграмма классов



Любой экземпляр класса `Vector3D` теперь включает в себя всё, что есть в классе `Point2D`

Наследование

```
class Vector2D{  // вектор на плоскости
protected:
    float _x, _y;  // компоненты вектора
public:
    Vector2D() {}
    void setX(float x) {_x = x;}
    void setY(float y) {_y = y;}
    float x() const {return _x;}
    float y() const {return _y;}
    float abs() const {return sqrt(_x*_x + _y*_y);}
};
class Vector3D : public Vector2D{ // построим новый класс на основе Vector2D
    float _z;   // поля _x и _y унаследованы
public:         // x(), setX() и др. методы тоже унаследованы
    Vector3D() {}
    void setZ(float z) {_z = z;}
    float z() const {return _z;}
    // метод вычисления длинны вектора здесь должен быть свой
    float abs() {return sqrt(_x*_x + _y*_y + _z*_z);}
};

Vector3D v1;
v1.setX(3);
v1.setZ(4);
v1.abs(); // 5
```

Преобразование типов при наследовании

```
Vector2D v1(10, 20);
Vector2D *v11;

Vector3D v2(100, 200, 300);
Vector3D *v22;

// так можно. но все, что не входит в Vector2D будет отброшено
v1 = v2;
v11 = &v2; // и так можно.
v11->setX(42); // v2 = (42, 200, 300). но z так не поменять

// а так нельзя: откуда взять z?
v2 = v1;
// это тоже нельзя
v22 = v11;
```

Наследование. Пример 2

```
class B{
    int x_;
  protected:
    int y;
  public:
    B() { cout << "Base constructor";
      x_ = 42; y = 9000;}
    void setX(int x_) {x = x_;}
    int x() const {return x_;}
    int getY() {return y;}
    void foo() const {cout << "Base";}
};
```

```
class D : public B{
// поле x_ унаследовано,
// но к нему нет прямого доступа
// к полю у есть прямой доступ
// только внутри этого класса
public:
// в списке инициализации возможен
// вызов конструктора базового класса
  D() : B() { setX(1729); }
  void bar() const
    {cout << "Delivered";}
};
```

```
B base;
D del;
base.foo();   // Base
// вызов унаследованного метода
del.foo();    // Base
del.bar();    // Delivered
del.x();      // 1729
del.getY();   // 9000
```

Наследование и конструкторы, деструкторы ...

Эти методы хоть и наследуются, но не избавляют от написания аналогичных в производном классе:

- Конструкторы
- Деструктор
- Операторы присваивания

Например в конструкторе производного класса можно вызывать конструктор базового класса, но нельзя вызывать второй *вместо* первого.

Наследование и конструкторы, деструкторы ...

- Конструктор по умолчанию базового класса вызывается автоматически перед вызовов конструктора производного класса.
- Если базовых классов несколько (многоуровневое наследование) то сначала вызывается конструктор самого базового класса.
- Деструкторы вызываются в обратном порядке: от производного класса к базовым

Наследование и конструкторы, деструкторы ...

```
class Vector2D{
    public:
        Vector2D(float x, float y) {_x = x; _y = y;}
    // ...
};

class Vector3D : public Vector2D{
public:
    // Вызов конструктора базового класса в конструкторе производного
    Vector3D(float x, float y, float z) : Vector2D(x,y) {_z = z;}
    // ...
```

Наследование и операторы

Операторы наследуются?

```
class Vector2D{
    public:
    // ...
    Vector2D operator + (const Vector2D& v);
};

class Vector3D : public Vector2D{
    // ...
};

Vector3D v1, v2;
Vector3D v3 = v1 + v2;
```

Компилируется?

Наследование и операторы

Операторы наследуются?

```
class Vector2D{
    public:
    // ...
    Vector2D operator + (const Vector2D& v);
};

class Vector3D : public Vector2D{
    // ...
};

Vector3D v1, v2;
Vector3D v3 = v1 + v2;
```

Компилируется?

Ошибка: не определено оператора сложения для класса Vector3D.

Ошибка станет очевиднее, если записать вызов оператора как

```
Vector3D v3 = v1.operator + (v2);
```

Наследование и модификаторы наследования

When the component is declared as:	When the class is inherited as:	The resulting access inside the subclass is:
public	public	Public
protected		protected
private		none
public	protected	protected
protected		protected
private		none
public	private	private
protected		private
private		none

Outline

Перекрытие (overriding) и перегрузка (overloading) имён

Перекрытие имён в классе = переопределение имен

```
class B {
      public:    void foo() {cout << "base";}
                 void bar() {cout << "bar";}
                 void bar(string s) {cout << "bar " << s;}      };

class D: public B{
  public: void foo(){cout << "delivered";}
          void bar(int x) {cout << "bar";}                    };

B base;
D delivered;
base.foo();          // base
delivered.foo();     // delivered
// Если нужно вызывать метод базового класса в производном:
delivered.B::foo(); // base
base.bar();          // bar
base.bar("x");       // barx
delivered.bar();     // bar
delivered.bar(42);   // bar42
```

метод foo() - переопределён; метод bar() - перегружен

Множественное наследование

Множественное наследование - наследование от нескольких классов одновременно.

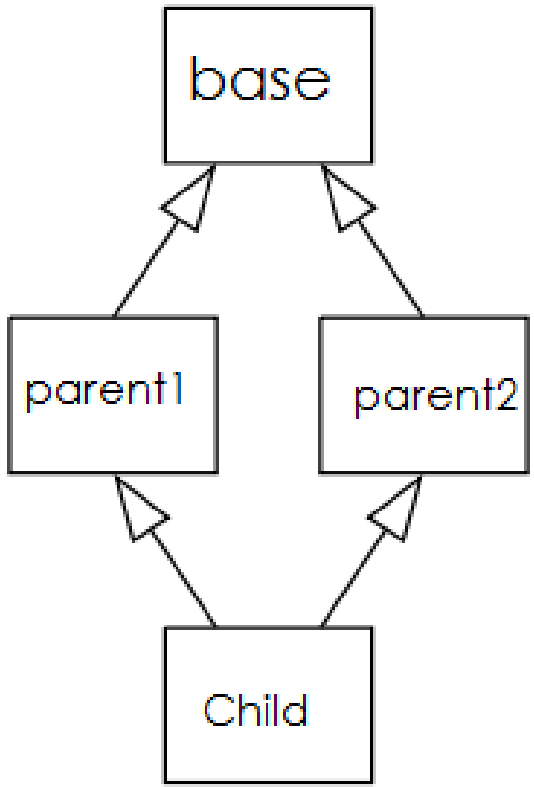
```
class Z: public X, public Y { . . . };
```

При множественном наследовании возникает проблема неоднозначности из-за совпадающих имен в базовых классах.

Поэтому лучше наследоваться от интерфейсов и классов-контейнеров.

Deadly Diamond of Death

Проблема ромба [wiki]



если метод класса Child вызывает метод, определенный в классе A, а классы B и C по-своему переопределили этот метод, то от какого класса его наследовать: B или C ?

Outline

Outline

Виртуальные методы

Виртуальный метод - метод, который может быть переопределён в классах-наследниках так, что конкретная реализация метода для вызова будет определяться во время исполнения.

Чистый виртуальный (абстрактный) метод - виртуальный метод для которого не приведена реализация.

Виртуальные методы

Зачем нужны?

- Реализует динамический полиморфизм
- Упрощает интерфейс

У целого набора классов может быть метод с одним именем и набором параметров (или несколько таких), который решает одну и ту же задачу, но специфичным для каждого класса способом. Какая конкретно реализация метода должна быть вызвана определяется во время выполнения программы.

Ранее и позднее связывание

Статическая типизация (раннее связывание) — определение типа на этапе компиляции.

Динамическая типизация (позднее связывание) — определения типа во время выполнения программы.

Динамический полиморфизм

Реализуется с помощью виртуальных методов.

Вопросы

- Чем отличается переопределение виртуальных методов от переопределения виртуальных?
- Для чего нужен динамический полиморфизм? Приведите примеры.
- Как задействовать позднее связывание?

Ссылки и литература

1. Г. Буч. Объектно-ориентированный анализ и проектирование с примерами приложений. 720 с. 2010 г. 700 страниц. Теория. Примеры на C++. Картинки! Вторая половина книги - примеры OOA и OOD с UML диаграммами.
2. MSDN - Microsoft Developer Network
3. Qt 5.X. Профессиональное программирование на C++. Макс Шлее. 2015 и более поздние издания г. 928 с. Книга периодически обновляется с выходом новых версий фреймворка Qt.
4. www.stackowerflow.com - система вопросов и ответов
5. draw.io — создание диаграмм.

Материалы курса

Слайды, вопросы к экзамену, задания, примеры

github.com/VetrovSV/OOP