Лекция 4. ООП. Класс

ООП

Объектно-ориентированное программирование - парадигма программирования, которая основывается на представление в коде программы различных объектов, взаимодействующих

друг с другом.

Класс - пользовательский тип данных, шаблон (макет) для создания объектов и описания их характеристик, функций.

Объект - экземпляр класса. Объект включает данные (поля) и методы (функции).
Что позволяет хранить характеристики объекта, изменять их и взаимодействовать с другими объектами.

Основные принципы ООП:

- Инкапсуляция
- Наследование
- Полиморфизм
- Абстракция

Инкапсуляция - объединение данных и методов для работы с данными внутри класса.

Сокрытие деталей реализации класса.

```
class Budget {
public:
    void increase_balance(double value) {
        budget_ += value;
    }
private:
    double budget_;
};
```

Наследование

Наследование - механизм создания новых классов на основе существующих. Позволяет строить иерархию классов и переиспользовать код классов родителей внутри классов наследников.

```
class Animal { /* common data */};
class Cat : public Animal {};
class Dog : public Animal {};
```

Полиморфизм

Полиморфизм - возможность различного поведения сущностей C++.

Виды полиморфизма:

- статический (на этапе компиляции, шаблоны, перегрузка функций)
- динамический (во время выполнения программы, виртуальные методы)

```
class Shape {
public:
    virtual void draw() = 0;
};
class Circle : public Shape {
    void draw() override { /* рисуем круг */ }
};
```

Абстракция

Абстракция - упрощение сложных вещей через выделение основных характеристик.

Класс

Класс - пользовательский тип данных, который объединяет в себе данные (поля класса) и функции для работы с данными (методы класса), представляет собой макет для создания объектов (экземпляров) данного типа.

Синтаксис: `class <name> {<class_body>};`

- `<name>` - имя класса, пользовательского типа данных
- `<class_body>` - тело класса, включающее поля, методы, конструкторы и деструктор

Спецификаторы доступа

Для ограничения видимости полей и методов внутри класса используются спецификаторы доступа, весь код после спецификатора имеет соответствующий тип доступа:

- `public` - публичный доступ, поле или метод класса доступны извне
- `protected` - защищенный доступ, поля и методы доступны наследникам класса и внутри класса
- `private` - приватный доступ, поля и методы доступны только внутри класса.

Синтаксис внутри класса или структуры: `<specifier>:`

Указывать спецификаторы доступа можно произвольное число раз.

```
class User {
public:
    /* some data and functions for everyone */
protected:
    /* some data and functions for children classes */
private:
    /* some data and functions inside class */
};
```

Приватные поля и методы будут недоступны снаружи, то есть **НЕЛЬЗЯ** к ним обратится или вызвать через экземляр класс, используя операторы `.`, `->`.

Всё содержимое класса по умолчанию имеет спецификатор доступа `private`, несмотря на это часто принято явно указывать данный спецификатор, даже при определении полей класса в самом начале тела класса.

Отличие класса и структуры

Структура `struct` и класс `class` имеют одинаковые возможности в C++.

Отличие заключается, что содержимое структуры по умолчанию имеет публичный доступ `public`, а содержимое класса приватный `private`

Структура нужна для взаимодействия с `legacy` кодом на языке C, а также для публичных классов.

Несмотря на одинаковые возможности, принято разделять структуру и класс семантически. Так, структуру используют только с публичными полями, а класс с приватными. Создавать классы и структуры со смешанным типом полей не рекомендуется, так как это может быть не очевидно и не понятно программистам, читающим код.

Пустой класс

Пустой класс имеет размер 1 байт, поскольку объект такого класса можно создать и необходимо иметь адресс данного объекта, чтобы иметь адресс, необходимо что-то положить в память по определенному адресу.

Чаще используется пустая структура. Такая структура может понадобитсья в качестве именнованного тега. Пока будем просто считать, что иногда надо.

Поля класса

Поля класса представляют собой внутренние переменные произвольного типа.
К полям класса внутри класса можно обращаться по имени. В качестве поля можно

использовать указатели и ссылки.

В случае ссылок необходимо их инициализировоать при создании объекта. Например, можно проинициализирвоать адресом объекта из глобальной области видимости. А еще это можно сделать в списке инициализации при конструировании объекта.

Существуют разные стили кода к именованию полей класса. Часто встречается:

- `m_<name>` - добавляют `m_` в качестве префикса к перименной (`m` - `member`)
- `<name>_` - добавляют `_` в качестве постфикса к переменной.

```
// inside class
int m_value;
int value_;
```

Поля класса хранятся в классе и инициализируются в порядке их объявления.

Поля уникальны для каждого экземпляра класса.

Инициализация полей значением по умолчанию

Аналогично структурам рекомендуется всегда инициализировать поля внутри класса.

```
class Time {
private:
    int hour_ = 0;
    int minute_{0}; // uniform
};
```

Иначе в полях класса также может лежать мусор.

Конструктор

Конструктор это особый метод класса, который используется для конструирования объекта.

Синтаксис: `<class_name>(<arguments>) {<ctor_body>}`

- `<class_name>` - имя конструктора должно совпадать с именем класса
- `<arguments>` - аргументы конструктора.
- `<ctor_body>` - тело конструктора.

В зависимости от аргументов конструктора выделяют различные типы конструктора. Основные способы конструирования объекта:

- Параметризованный конструктор

- Конструктор по умолчанию
- Конструктор копирования
- Копирующий оператор присваивания
- Конструктор перемещения
- Перемещающий оператор присваивания

Важно понимать, что если конструкторы не определены, то компилятор самостоятельно сгенерирует конструкторы. Но если определен, хотя бы один конструктор, то компилятор скорее всего этого не сделает.

Важно понимать, что при входе в тело конструктора все поля уже проинициализированы и в теле может происходить только присваивание новых значений полям класса. Следовательно, в теле уже нельзя изменить константное поле или инициализировать ссылку.

Проинициализировать константу и ссылку можно не только значением по умолчанию или значением (адресом) переменной из глобальной области видимости. Для этого в синтаксисе конструктора предусмотрен список инициализации.

Список инициализации полей класса

Список инициализации полей (*member initializer list*) позволяет инициализировать поля в момент создания объекта. В списке инициализации доступны аргументы конструктора и имена полей класса. Список инициализации указывается между сигнатурой и телом конструктора, и выглядит как перечисление после символа `:` через запятую полей класса и внутри `()` или `{}` их инициализирующих значений.

Синтаксис: `<class_name>(<arguments>) : <initializer_list> {<ctor_body>}`

- `<initializer_list>` - список инициализации = `<field>(<value>), <field>{<value>}`

```
class InitList {
public:
    InitList(int val) : a_(val), b_(val), c_(val) {}
public:
    int a_; int b_; int c_;
};
```

Причем имена полей класса и имена параметров могут полностью совпадать, конфликта имен не будет, поскольку компилятор понимает, что нужно инициализировать поля.

```
class InitSameName {
public:
    InitSameName(int a, int b, int c) : a(a), b(b), c(c) {}
public:
    int a; int b; int c;
};
```

Также, следует отметить, что в качестве инициализирующего значения, может использоваться не только переменная или константа, но также выражение (*expression*) и результат вызова функции.

ВАЖНО, что инициализация происходит в порядке полей класса, и не зависит от порядка в списке инициализации. Поэтому важно самостоятельно отслеживать правильный порядок инициализации.

```
class BadOrderInit {
public:
    BadOrderInit(int val) : c(val), b(c + 1), a(10) {}
public:
    int a; int b; int c;
};
```

- `c` используется неинициализированной при инициализации `b` (**UB**)

Если поля класса объявлены со значением по умолчанию, то они будут проигнорированы для полей в списке инициализации.

```
class BadOrderInit {
public:
    BadOrderInit(int val) : c(val), b(c + 1), a(10) {}
public:
    int a = 7; int b = 7; int c = 7;
};
```

- значение `7` будет проигнорированно, по-прежнему **UB**

Списки инициализации могут быть неполными. Тогда недостающие поля будут сконструированы со значениями по умолчанию, а при их отсутствии инициализируются мусором.

```
class BadOrderInitNoAB {
public:
    BadOrderInitNoAB(int val) : c(val) {}
public:
    int a; int b = 7; int c = 7;
};
```

- в поле `b` будет значение `7`, в `a` будет мусор

Список инициализации позволяет сконструировать константное поле и поле ссылку извне:

```
class RefConst {
public:
    RefConst(int value, int& ref, const double& cref)
        : id_(value), ref_(ref), const_ref_(cref) {}
private:
    const int id_;
```

```
    int& ref_;
    const double& const_ref_;
};
```

Параметризованный конструктор

Конструктор, который имеет параметры (аргументы) называют параметризованным конструктором (конструктором с параметрами). Аргументов может быть несколько и они могут иметь значения по умолчанию, Таким образом, конструктор может быть перегружен.

```
class Time {
public:
    Time(int hour, int minute, int second)
        : hour_(hour), minute_(minute), second_(second) {}
private:
    int hour_, minute_, second_;
};
```

Если конструктор имеет у всех аргументов значение по умолчанию, то такой конструктор перегружает конструктор по умолчанию.

```
class Time {
public:
    Time(int hour = 0, int minute = 0, int second = 0)
        : hour_(hour), minute_(minute), second_(second) {}
private:
    int hour_, minute_, second_;
};
```

Для создания объекта класса необходим вызов конструктора. Синтаксис вызова констуктора:

```
Time t1(1, 1, 1);
Time t2{1, 1, 1};
Time t3 = {1, 1, 1}
Time t4 = Time{1, 1, 1};
Time t5 = Time(1, 1, 1);
```

Аналогично для всех его вариантов перегрузки.

Конструктор по умолчанию

Конструктор по умолчанию представляет собой конструктор без аргументов. Часто для простых класов конструктор имеет пустое тело. Удобно использовать значение по умолчанию для инициализации.

Синтаксис: `<class_name>() {<ctor_body>}`

Часто имеет пустое тело для тривиальных случаев.

Если не определен ни один конструктор, то компилятор самостоятельно сгенерирует данный конструктор.

```
class DefaultCtor {
public:
    DefaultCtor() {}
private:
    int value = 0;
};
```

Вызов конструктора:

```
DefaultCtor obj;
DefaultCtor obj2{};
DefaultCtor obj3 = {};
DefaultCtor obj4 = DefaultCtor{};
DefaultCtor obj5 = DefaultCtor();
```

- во всех этих случаях вызовется только конструктор по умолчанию один раз

Конструктор копирования

Конструктор копирования необходим для создания копии объекта из объекта того же типа. Представляет собой конструктор, принимающий в качестве аргументов константную ссылку того же типа, что и сам класс.

Синтаксис: `<class_name>(const <class_name>& <other>) {<ctor_body>}`

```
class Time {
public:
    Time(const Time& other)
        : hour_(other.hour_), minute_(other.minute_), second_(other.second_) {}
private:
    int hour_, minute_, second_;
};
```

Поля другого объекта того же класса доступны внутри методов класса даже если они приватные.

Вызывается конструктор копирования:

- при передаче в функцию по значению
- при возврате объекта соответствующего значения по значению
- при конструировании одного объекта из другого

```
Time t;
Time t1 = t; // copy ctor
Time t2(t);  // copy ctor
Time t3{t};  // copy ctor
```

Указатель на себя `this`

Внутри класса, в методах, в том числе конструкторах, можно получить указатель на себя (объект класса, который вызывает данный метод) с помощью ключевого слова `this`.

Можно использовать `this`, как в качестве значения по умолчанию, так и в списке инициализации.

```
class Self {
public:
    Self() : self(this) {};
    Self* GetPtr() { return self; }
    Self& GetRef() { return *this; }
private:
    Self* self = this;
};
```

Можно считать что указатель на себя передается первым неявным аргументом в конструкторы,
методы и операторы класса.

Через указатель можно явно обращаться к полям класса, но как правило, так не делают

```
// inside methods
this->self;
```

Копирующий оператор присваивания

Оператор присвания необходим при присваивании одного созданного объекта другому. Если один из объектов не создан, то он не будет вызываться, а будет вызываться конструктор копирования, даже если в инструкции есть `=`.

Как правило, оператор возвращает ссылку на себя (экземпляр текущего класса), что позволяет испоьзовать цепочку из операторов `=`. Для этого необходимо вернуть из оператора разыменованный указатель на себя `return *this;`.

Синтаксис: `<class_name>& operator=(const <class_name>& <other>) {<ctor_body>}`

Поскольку язык не запрещает присвоить объект самому себе, как правило, в копирующем операторе присваивания выполняют проверку на самоприсваивание. Особенно это

критично для классов владеющих ресурсами (выделяющих память), что может привести к **UB**

```
class Time {
public:
    Time& operator=(const Time& other) {
        if (this == &other) {
            return *this;
        }
        hour_ = other.hour_;
        minute_ = other.minute_;
        second_ = other.second_;
        return *this;
    }
private:
    int hour_, minute_, second_;
};
```

Вызов оператора:

```
Time t1, t2, t3;
t1 = t2;      // copy assignment
t1 = t1;      // copy assignment
t1 = t2 = t3; // copy assignment
auto t4 = t1; // copy ctor (not a copy assignment!)
```

Деструктор

Особый метод, вызываемый перед разрушением объекта, когда заканчивается время жизни объекта.

Синтаксис: `~<class_name>() {}`

Если в конструкторе выполнялось ручное выделение ресурсов, то в деструкторе необходимо обязательно освободить ресурсы. Иначе деструктор остается тривиальным и генерируется компилятором по умолчанию.

Деструкторы вызываются в обратном порядке по отношению к конструируемым объектам при выходе из области видимости. Последний сконструированный объект, будет разрушен первым.

Конструктор преобразования

Конструктором преобразования называется конструктор, принимающий один аргумент другого произвольного типа. Данный конструктор разрешает неявное преобразование из указанного типа в тип класса.

```
class Integer {
private:
    int value;
public:
    Integer(int v) : value(v) {}
    Integer(char c) : value(static_cast<int>(c)) {}
};
```

Таким образом, если функция принимает пользовательский класс, а класс имеет конструктор преобразования от другого типа, то в функцию можно передать непосредственно этот другой тип, произойдет неявное преобразование с помощью соответствующего конструктора:

```
int DoSomething(Integer i) {}

int main() {
    Integer i{3};
    int value = 5;
    char c = 'I';
    DoSomething(i);     // OK
    DoSomething(value); // OK
    DoSomething(5);     // OK
    DoSomething(c);     // OK
    DoSomething('i');   // OK
}
```

ВАЖНО понимать, что при наличии конструктора присваивания из другого типа, компилятор **НЕ** будет генеировать оператор присваивания из данного типа, его необходимо определять самостоятельно.

Ключевое слово `explicit`

Ключевое слово `explicit` используется как спецификатор перед именем конструктора и позволяет запретить неявное преобразование и сообщает компилятору, что данный конструктор можно вызывать только явно.

```
class Integer {
private:
    int value;
public:
    Integer(int v) : value(v) {}
    Integer(char c) : value(static_cast<int>(c)) {}
    explicit Integer(double d) : value(static_cast<int>(d)) {}
};
```

Неявно такой конструктор вызвать нельзя:

```
//Integer i2 = 3.14;          // compile error
Integer i3 = Integer{3.14}; // OK

int DoSomething(Integer i) {}

int main() {
    double d = 3.14;
    //DoSomething(d);             // compile error
    //DoSomething(3.14);          // compile error
    DoSomething(Integer{3.14}); // OK
    DoSomething(Integer(3.14)); // OK
}
```

Также спецификатор `explicit` можно использовать с оператором преобразования, об этом после знакомства с методами.

Конструктор от `std::initializer_list` (*C++11*)

В C++11 появился контейнер список инициализации `std::initializer_list`, который позволяет инициализировать класс набором элементов. Что вызывает неоднозначность при наличии параметризированных конструкторов какой конструктор вызывать.

Конструктор по умолчанию имеет приоритет перед конструктором от списка инициализации.

Список инициализации имеет приоритет перед параметризированными конструкторами при использовании `{}`.

```
class Vector {
public:
    Vector() {};
    Vector(size_t count);
    Vector(int a, int b);
    Vector(std::initializer_list<int> list);
private:
    std::vector<int> data;
};
```

Вызов конструкторов:

```
Vector v = {1, 2, 3, 4, 5}; // ctor std::initializer_list
Vector v2{1, 2, 3};         // ctor std::initializer_list
Vector v3(10);              // ctor Vector(size_t)
Vector v4{10};              // ctor std::initializer_list
Vector v5 = {10};           // ctor std::initializer_list
Vector v6(10, 20);          // ctor Vector(int a, int b)
Vector v7{10, 20};          // ctor std::initializer_list
Vector v8 = {10, 20};       // ctor std::initializer_list
Vector v9 = 10;             // ctor Vector(size_t) implicit cast
Vector v10;                 // default ctor
```

```
Vector v11{};               // default ctor
Vector v12 = {};            // default ctor
```

Делегирующий конструктор (*C++11*)

Делегирующий конструктор - конструктор, который на месте списка инициализации использует другой конструктор данного класса. В таком случае можно указать только один целевой конструктор, дополнительно списки инициализации указать нельзя.

```
class Time {
public:
    Time(int hour, int minute, int second)
        : hour_(hour), minute_(minute), second_(second) {}
    Time(int hour) : Time(hour, 0, 0) {}
private:
    int hour_, minute_, second_;
};
```

Делегирующий конструктор **НЕ** может быть рекурсивным.

Ключевое слово `default` (*C++11*)

С помощью ключевого слова `default` можно явно попросить компилятор сгенерировать конструктор (деструктор), указав после сигнатуры `= default`. Это более выразительно, чем писать `{}` для конструктора по умолчанию. Рекомендуется к использованию.

```
class Value {
public:
    Value(int x) : x_(x) {}
    Value() = default;
    Value(const Value& other) = default;
    Value(Value&& other) = default;
    Value& operator=(const Value& other) = default;
    Value& operator=(Value&& other) = default;
    ~Value() = default;
private:
    int x = 0;
};
```

Ключевое слово `delete` (*C++11*)

С помощью ключевого слова `delete` можно явно попросить компилятор удалить функцию (запретить её использование), указав после сигнатуры `= delete`. Это более выразительно, чем прятать конструкторы в приватную область класса. Рекомендуется к использованию.

Можно использовать не только для конструкторов, деструкторов, но и для любых методов, операторов, шаблонных функций, функций вне классов.

```
class Value {
public:
    Value(int x) : x_(x) {}
    Value() = delete;
    Value(const Value&) = delete;
    Value& operator=(const Value&) = delete;
private:
    int x = 0;
};
```

Например, если класс не подразумевает сравнения на равенство или других операторов можно явно указать для них `delete`.

Методы

Внутри класса можно определять функции, которые могут работать с полями класса, в том числе закрытыми. Данные функции называются методы.

Синтаксис аналогичен определению обычным функциям.

Публичный метод можно вызвать через операторы `.` для экземпляра и `->` для указателяю.

Приватные методы, можно вызывать внутри класса.

Можно вызывать методы в списках инициализации. Например, метод, который будет контролировать допустимость значения или выполнять дополнительные преобразования.

Определение методов вне класса

Методы можно объявить внутри класса, а определить снаружи класса. Содержимое класса имеет свою область видимости. Для определения снаружи класса перед именем конструктора, метода, оператора используется имя класса и оператор разрешения области видимости `::`

```
class Time {
public:
    Time();
    Time(int hours, int minutes, int seconds = 0);
    int GetHours();
    void SetHours(int hours);
private:
    int hours_ = 0;
    int minutes_ = 0;
    int seconds_ = 0;
```

```
};

Time::Time() = default;
Time::Time(int hours, int minutes, int seconds)
    : hours_(hours), minutes_(minutes), seconds_(seconds) {}

int Time::GetHours() { return hours; }
void Time::SetHours(int hours) { hours_ += hours; }
```

Аргументы методов, имеющие значения по умолчанию указываются только при объявлении, при определении нельзя указать значения по умолчанию

CV-квалификация методов

Методы могут иметь CV-квалификацию. Методы, которые не изменяют полей класса, а только предоставляют информацию о них следует помечать квалификатором `const` после сигнатуры и перед телом метода:

```
class Size {
public:
    size_t GetSize() const { return size; };
    void AddSize(size_t size) { size_ += size; };
private:
     size_t size_ = 0;
};
```

Методы помеченные квалификатором `const` можно вызывать у константных объектов класса. Компилятор отслеживает, что в данном методе нет измененеий полей класса.

Методы можно перегрузить только по квалификатору `const`.

Не изменяет поля класса и может быть вызван для константного объекта:

```
int Class::foo() const;
```

Может изменять поля класса и может быть вызван для `volatile` объекта:

```
int Class::foo() volatile;
```

Может быть вызван как для `const`, так и для `volatile` объекта, так и для `const volatile` объекта:

```
int Class::foo() const volatile;
```

Оператор преобразования

В классе возможно определить оператор преобразования, который позволяет преобразовывать
пользовательский класс в другой тип.

Синтаксис: `<explicit> operator <type>() const {<body>}`

- `<explicit>` - можно запретить неявное преобразование
- `<type>` - тип к которому выполняется приведение

Рекомендуется помечать `const` поскольку данный оператор не должен менять полей класса и вызываться от констант данного класса.

Как правило рекомендуется запрещать неявное преобразование к типу (использовать `explicit`), поскольку можно обнаружить много неожиданных мест в коде, где неявно произведено преобразование.

Исключением обычно является оператор `bool` для удобства использования в условиях.

Перегрузка операторов внутри класса

Поскольку первым аргументом неявно передается ключевое слово `this`, то перегрузка бинарных операторов внутри класса имеет один аргумент:

```
Class& operator+=(const Class& other);
Class& operator-=(const Class& other);
Class& operator*=(const Class& other);
Class& operator/=(const Class& other);
```

Операторы арифмесстических операций часто переопределяют на основе работы присваивающих
операторов:

```
Class operator+(const Class& other) const {
    Class result = *this;  // copy ctor
    result += other;       // operator +=
    return result;
}
```

Операторы префиксного и постфиксного инкремента/декремента переопределяются следующим образом:

```
Class& operator++();   // ++obj
Class operator++(int); // obj++
Class& operator--();   // --obj
Class operator--(int); // obj--
```

- постфиксный оператор возвращает копию, поэтому у возвращаемого значения нет `&`

Перегрузка операторов вне класса

Операторы можно перегрузить вне класса, тогда сигнатура перегружаемого оператора пишется в привычной манере. Но для реализации таких операторов у класса должны быть методы задающие и считывающие значение полей (геттеры и сеттеры). Бывает, что их нет, тогда перегрузить класс не получится или получится на основе определенных операторов составного присваивания внутри класса.

Перегрузка инкремента и декремента вне класса:

```
Class& operator++(const Class& obj);   // ++obj
Class operator++(const Class& obj, int); // obj++
Class& operator--(const Class& obj);   // --obj
Class operator--(const Class& obj, int); // obj--
```

Ключевое слово `friend`

Внутри класса с помощью ключевого слова `friend` (*friend declaration*) можно объявить дружественную функцию, класс или дружественный метод другого класса.

Сущности объявленные дружественными будут иметь доступ к `private` и `protected` полям класса.

Дружественность работает в одностороннем порядке.

```
friend void SomeMethod(int);
friend struct SomeStruct;
friend class SomeClass;
friend OtherClass;                // C++11
friend int OtherClass::Method();
```

Ключевое слово `mutable`

Спецификатор типа `mutable` разрешает изменять поле класса, объявленное с ним, даже в константных методах и для константных объектов.

Например, это может быть поле представляющее собой счетчик операций и необходимо его изменять даже в константном методе.

Также может использоваться в лямбда-выражениях