ООП с примерами

Введение

Прежде всего стоит ответить, зачем? Объектно-ориентированная идеология разрабатывалась как попытка связать поведение сущности с её данными и спроецировать объекты реального мира в программный код. Предполагалось, что такой код проще читать и понимать человеком.

Известно, что воспринимать окружающий мир человеку привычнее как множество взаимодействующих между собой классифицированных объектов.

Классы и объекты

Класс — это описание того, какими свойствами и поведением будет обладать объект. А **объект** — это экземпляр с собственным состоянием этих свойств.

Мы говорим «свойства и поведение», но звучит это как-то непонятно. Привычнее для программиста — «переменные и функции». На самом деле «свойства» — это такие же обычные переменные, просто они являются атрибутами какого-то объекта (их называют **полями объекта**), а «поведение» — функции объекта (их называют **методами**), которые также являются атрибутами объекта.

Примечание: разница между методом объекта и обычной функцией лишь в том, что метод имеет доступ к собственному состоянию через поля.

Свойства и методы образуют **интерфейс объекта**, а сущности с конкретным сочетанием свойств являются экземплярами объекта.

Дадим определение ООП на основе введённых понятий.

Объектно-ориентированное программирование – это методология разработки программ **как совокупности объектов**, относящихся к определенным **классам**, и взаимодействующих через **свойства** и **методы** друг друга для выполнения операций и преобразования данных, определяя поведение своих экземпляров.

Создание интерфейса классов

Для определения такого типа данных как класс в языке C++ используется ключевое слово class. Использование ключевого слова class определяет новый пользовательский тип данных.

```
class Character {
}
```

Теперь можно добавить поля и свойства для класса. Поле объявляется как обычная переменная:

```
class Character {
    int health;
    int attack;
    int armor;
};
```

Теперь у объекта есть свои поля, но к ним нельзя обратиться извне, потому что закрыт доступ (подробнее об этом — в разделе про инкапсуляцию). Чтобы его открыть, нужно использовать ключевое слово public.

```
class Character {
  public:
    int health;
    int attack;
    int armor;
}
```

Если доступ к полям открыт, то с ними можно проводить вычисления или просто получать их значение. Если же нужно запретить доступ к определенным полям — используйте уровень доступа private, тогда они будут доступны только изнутри класса.

Всё, что находится внутри фигурных скобок, относится к этому классу. Даже если класс еще пустой, уже можно создать его экземпляр — объект. Это называется объявлением или инстанцированием.

```
Character hero = new Character();
```

Это похоже на то, как создаются переменные, но вместо типа данных указывается название класса. После знака присваивания указываются ключевое слово new, которое необходимо использовать для создания нового экземпляра какого-либо класса, и конструктор — специальный метод, который позволяет создать объект.

Вы уже знаете, что определение переменной фундаментального типа данных (например, int x) приводит к выделению памяти для этой переменной, так же и создание объекта класса приводит к выделению памяти для этого объекта.

Внутри скрытых свойств объекта часто можно встретить две конструкции (подробнее об этом — в разделе про инкапсуляцию):

1. **get** *(геттер — позволяет получить значение свойства)*. В ней возвращается значение, которое идет после ключевого слова return;
2. **set** *(сеттер — позволяет изменить значение)*. В ней указывается поле, которое нужно изменить, используя значение *value*.

```
class Character {
    int health;
    int attack;
    int armor;
  public:
    int getHealth() {
       return this.health;
    }
    void setHealth(int _healthPoints) {
       health = _healthPoints;
    }
    int getAttack();
    void setHealth(int _attackPoints);
    int getArmor();
    void setArmor(int _armorPoints);
}
```

Такие манипуляции нужны для того, чтобы доступ к полям осуществлялся только так, как это нужно разработчику.

Статические поля и методы класса

Метод getHealth(), несмотря на то, что является общим для всех экземпляров класса, **применяется к конкретному объекту и работает с его конкретным полем**. Кроме таких полей и методов класса можно также определять статические поля и методы. Статические члены класса **принадлежат классу**, а не его экземплярам. То есть, они будут являться общими для всех созданных объектов.

Например, добавим к нашему классу Character статическое поле count, которое будет увеличиваться на 1 каждый раз, когда создаётся новый экземпляр, то есть будем подсчитывать количество существующих уже персонажей. Статическое поле создаётся с помощью служебного слова static.

```
class Character {
    int health;
    int attack;
    int armor;
  protected:
    static size_t count;
  public:
    ...
}
```

Особенности статических полей класса:

1. Обращение к статическому полю ведётся через имя класса

```
Character::count
```

2. Тем не менее, обращаться к статическому полю можно и через указатель this

```
this->count
```

3. Статическое поле необходимо инициализировать. Инициализация происходит в любом месте cpp-файла с реализацией класса. В нашем случае

```
size_t Character::count = 0;
```

4. Статическая переменная объявлена как protected. То есть, обратиться извне к ней сможет только наследник. Как и с обычными нестатическими переменными, методы класса могут без проблем обращаться к своей собственной переменной.

Теперь необходимо каким-то образом получить доступ к нашей переменной. Для этого напишем метод, возвращающий значение count.

```
class Character {
    int health;
    int attack;
    int armor;
  protected:
    static size_t count;
  public:
    static size_t getCount();
    ...
}
```

```
size_t Character::count = 0;

size_t Character::getCount() {
    return Character::count;
}
```

Так как метод обращается к статическому полю, то он не нуждается в экземпляре класса для того, чтобы его можно было вызвать. Поэтому метод getCount сделан статическим.

Обращаться к методу теперь можно через имя класса Character::getCount() Тем не менее, обращаться к статическому методу можно и через экземпляр класса(hero.getCount()), но это **плохая практика**.

Если метод статический, то к нему можно обращаться до того, как был создан хотя бы один экземпляр переменной. Поэтому статический метод не может обращаться к нестатическим полям класса. Кроме того, статический метод просто не будет знать, к какому экземпляру обратиться – внутри него нет доступа до указателя this.

Для использования статического метода, таким образом, не нужно создавать отдельного экземпляра класса, а обращение ведётся через имя класса и оператор ::

Статические поля и методы, также как и нестатические, могут иметь модификаторы доступа private, protected и public. При использовании статических методов запрещён модификатор const метода, так как метод по определению не может обращаться к нестатическим полям и изменять их.

Базовые понятия для написания простых классов

1. **this** — это специальная локальная переменная (внутри методов класса!), которая позволяет объекту обращаться из своих методов к собственным атрибутам.

Пример: снаружи (например, в приложении, использующем данный класс) обращение к полю здоровья будет выглядеть так

```
hero.health
```

Если же объект захочет сам обратиться к своему полю x, **в его методе** обращение будет иметь вид this.health (как в методе getHealth() примера выше) или же просто health (как в методе setHealth(int) примера выше). То есть использовать слово this необязательно, но оно делает код более читаемым.

2. **constructor** — это специальный метод, который автоматически вызывается при создании объекта.

Конструктор может принимать любые аргументы, как и любой другой метод. В каждом языке конструктор обозначается своим именем. В языке C++ имя конструктора должно совпадать с именем класса. Назначение конструкторов — произвести первоначальную инициализацию объекта, заполнить нужные поля, возможно, произвести выделение требуемой памяти.

Основные типы конструкторов:

— конструктор по умолчанию;
— конструктор инициализации;

— конструктор копирования.

Существуют и другие виды конструкторов, например, конструктор преобразования типа, конструктор перемещения.

Допускается использовать несколько конструкторов с одинаковым именем, но различными параметрами.

```
class Character {
    int health;
    int attack;
    int armor;
  public:
    Character();                    // конструктор по умолчанию
    Character(int, int, int);       // конструктор инициализации
    Character(const Character&);    // конструктор копирования
    ...
}

Character::Character() {
  health = 100;
  attack = 100;
  armor = 100;
}

Character::Character(int _health, int _attack, int _armor) {
  health = _health;
  attack = _attack;
  armor = _armor;
}

Character::Character(const Character& hero) {
  health = hero.health;
  attack = hero.attack;
  armor = hero.armor;
}
```

Если в классе имеются поля, требующие выделения памяти, выделение этой памяти производится в конструкторах класса. Рассмотрим тот же пример с классом Character, у которого появляется еще два поля: количество навыков персонажа и массив этих навыков. В конструкторах этого класса появляется участки кода с выделением памяти под навыки.

```
class Character {
    int health;
    int attack;
    int armor;
    int numOfSkills;
    string* skills;
  public:
    Character();                        // конструктор по умолчанию
    Character(int, int, int, int*);     // конструктор инициализации
    Character(const Character&);        // конструктор копирования
    ...
}

Character::Character() {
  health = 100;
  attack = 100;
  armor = 100;
  numOfSkills = 3;
  skills = new string [3];
```

```
  skills[0] = "high intelligence";
  skills[1] = "magic";
  skills[2] = "craft";
}

Character::Character(int _health, int _attack, int _armor,
  int _numOfSkills , int* _skills) {
  health = _health;
  attack = _attack;
  armor = _armor;
  numOfSkills = _numOfSkills;
  skills = new string [numOfSkills];
  for (int i = 0; i < numOfSkills; ++i) {
     skills[i] = _skills[i];
  }
}

Character::Character(const Character& hero) {
  health = hero.health;
  attack = hero.attack;
  armor = hero.armor;
  numOfSkills = hero.numOfSkills;
  skills = new string [numOfSkills];
  for (int i = 0; i < numOfSkills; ++i) {
     skills[i] = hero.skills[i];
  }
}
```

ВАЖНОЕ ЗАМЕЧАНИЕ!

Некоторые типы данных (например, константы и ссылки) должны быть инициализированы сразу, например,

```
class Character {
    int health;
    int attack;
    int armor;
    int numOfSkills;
    string* skills;

    const int bonusToMagicAttack;

public:
   Character() {
       health = 100;
       attack = 100;
       armor = 100;
       numOfSkills = 3;
       skills = new string [3];
       skills[0] = "high intelligence";
       skills[1] = "magic";
       skills[2] = "craft";

       bonusToMagicAttack = 20; // ОШИБКА: константам нельзя
                                // присваивать значения
    }
    ...
};
```

Для решения этой проблемы в C++ добавили метод инициализации переменных-членов класса через список инициализации членов, вместо присваивания им значений после объявления.

Примечание. Вообще говоря, инициализировать переменные можно тремя способами: через копирующую инициализацию, прямую инициализацию, uniform-инициализацию.

```
int value1 = 3;          // копирующая инициализация
double value2(4.5);      // прямая инициализация
char value3 {'d'}        // uniform-инициализация
```

Использование списка инициализации почти идентично выполнению прямой инициализации. Пример написания конструктора инициализации с использованием списка инициализации:

```
Character::Character(int _health, int _attack, int _armor, int _numOfSkills ,
int* _skills) : health(_health), attack(_attack), armor(_armor),
numOfSkills(_numOfSkills), bonusToMagicAttack(20) {
  skills = new string [numOfSkills];
  for (int i = 0; i < numOfSkills; ++i) {
     skills[i] = _skills[i];
  }
}
```

Список инициализации членов находится сразу же после параметров конструктора. Он начинается с двоеточия (:), а затем значение для каждой переменной указывается в круглых скобках. Больше не нужно выполнять операции присваивания в теле конструктора. Также обратите внимание, что список инициализации членов не заканчивается точкой с запятой.

Приведённый код работает, поскольку нам разрешено инициализировать константные переменные (но не присваивать им значения после объявления!).

Методы constructor работают с внутренним состоянием, а во всем остальном не отличаются от обычных функций. Даже синтаксис объявления совпадает.

Обратимся к примеру, когда в нашем классе было статическое поле count, производящее подсчёт числа экземпляров класса. Если данное поле имеет место быть, то в реализации конструкторов нужно увеличивать значение данного счётчика, например,

```
Character::Character(const Character& hero) {
  health = hero.health;
  attack = hero.attack;
  armor = hero.armor;
  numOfSkills = hero.numOfSkills;
  skills = new string [numOfSkills];
  for (int i = 0; i < numOfSkills; ++i) {
     skills[i] = hero.skills[i];
  }
  Character::count++;
}
```

3. **destructor** — это специальный тип метода класса, который выполняется при удалении объекта класса. В то время как конструкторы предназначены для инициализации класса, деструкторы предназначены для очистки памяти после него.

Когда объект автоматически выходит из области видимости или динамически выделенный объект явно удаляется с помощью ключевого слова delete, вызывается деструктор класса (если он существует) для выполнения необходимой очистки до того, как объект будет удален из памяти.

Для простых классов (тех, которые только инициализируют значения обычных переменных-членов) деструктор не нужен, так как C++ автоматически выполнит очистку самостоятельно. Если же объект класса содержит любые ресурсы (например, динамически выделенную память или файл/базу данных), или, если вам необходимо выполнить какие-либо действия до того, как объект будет уничтожен, деструктор является идеальным решением, поскольку он последнее, что происходит с объектом перед его окончательным уничтожением.

```
class Character {
    int health;
    int attack;
    int armor;
    int numOfSkills;
    string* skills;
  public:
    Character();                          // конструктор по умолчанию
    Character(int, int, int, int*);       // конструктор инициализации
    Character(const Character&);          // конструктор копирования

    ~Character();                         // деструктор
    ...
}

Character::~Character() {
  // динамически удаляем массив, который выделили в конструкторе ранее
  delete[] skills;
}
```

Замечание. При наличии статического поля count, не забудьте внутри деструктора уменьшить значение счётчика на 1.

ПРИМЕЧАНИЕ на тему «А ЗАЧЕМ НУЖНЫ КОНСТРУКТОРЫ И ДЕСТРУКТОРЫ?»

Используя конструкторы и деструкторы, ваши классы могут выполнять инициализацию и очистку после себя автоматически без вашего участия! Это уменьшает вероятность возникновения ошибок и упрощает процесс использования классов.

4. Namespace — это пространство имен, в котором находится класс. Оно необходимо для того, чтобы не возникало конфликтов с именами классов и переменных из подключаемых библиотек. Указание именного пространства производится в виде

название пространства имён (класса) **::**

Базовые принципы ООП

Все основанные на объектах языки (C#, Java, C++, Smalltalk, Visual Basic и т.п.) должны отвечать трем основным принципам ООП, которые перечислены ниже:

1. **Инкапсуляция** — это свойство системы, позволяющее объединить данные и методы, работающие с ними в классе, и скрыть детали реализации от пользователя.
2. **Наследование** — это свойство системы, позволяющее описать новый класс на основе уже существующего с частично / полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс — потомком, наследником или производным классом.

3. **Полиморфизм** — это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Часто бытует мнение, что в ООП стоит выделять еще одну немаловажную характеристику — абстракцию. Официально её не вносили в обязательные черты ООП, но списывать ее со счетов не стоит. Заметим, что абстракция присуща для любого программирования, а не только для ООП.

4. ? **Абстрагирование** — это способ выделить набор значимых характеристик объекта, исключая из рассмотрения не значимые. Соответственно, абстракция — это набор всех таких характеристик.

Рассмотрим каждый принцип детально.

Базовые принципы ООП: абстракция

Абстракция – это концепция, согласно которой у объекта выделяются характеристики, которые отличают его от всех других объектов, чётко определяя его концептуальные границы.

Идея заключается в том, чтобы отделить способ использования составных объектов в программе от деталей их конкретной реализации в виде простых объектов. Абстракция отражает такую особенность предметной области, как идентичность и подобие объектов, и предполагает выделение только тех признаков и поведений, которые существенны с точки зрения их программной реализации.

Эта концепция предписывает программистам отделять программную реализацию от реального объекта с помощью свойств и методов класса, декларируя только то, что существенно с точки зрения программной реализации.

Умение балансировать между простотой архитектуры и гибкостью приложения — это искусство. Выбирая золотую середину, следует опираться не только на собственный опыт и интуицию, но и на контекст текущего проекта.

Неверный выбор уровня абстракции ведет к одной из двух проблем:

— **если абстракции недостаточно**, дальнейшие расширения проекта будут упираться в архитектурные ограничения, которые ведут либо к рефакторингу и смене архитектуры, либо к обилию костылей (оба варианта обычно несут за собой боль и финансовые потери);

— **если уровень абстракции слишком высок**, это приведет к оверинжинирингу [1](overengineering) в виде чересчур сложной архитектуры, которую трудно поддерживать, и излишней гибкости, которая никогда в этом проекте не пригодится. В этой ситуации любые простейшие изменения в проекте будут сопровождаться дополнительной работой для удовлетворения требований архитектуры (это тоже порой несет определенную боль и финансовые потери).

[1] Слово «оверинжиниринг» проникло и в русский язык, но пока его чаще используют компьютерщики для обозначение излишне громоздкого («индусского») кода. Специалисты говорят, что когда-то индийским программистам платили за строчки написанных ими программ, поэтому они выдавали сложный, объемный, витиеватый код. Оверинжиниринг в производственном смысле имеет два аспекта: изготовление продукта слишком сложным и затратным способом – «из пушки по воробьям», и сам продукт может быть сверхинжиниринговым — с более высоким качеством, чем это требуется клиенту, с излишними свойствами и опциями.

Уровни абстракции (на примере Character и игрового приложения)

Первый слой. В игре есть один класс персонажа, все свойства и поведение описаны в нем. Это совсем деревянный уровень абстракции, подходит для казуальной игры, которая не предполагает никакой особой гибкости.

Второй уровень. В игре есть базовый персонаж игры с основными способностями и классы персонажей со своей специализацией (типа маг, воин, шаман), которая описывается дополнительными свойствами и методами. Игроку предоставляется возможность выбора, а разработчикам упрощается добавление новых классов.

Третий уровень. Помимо классификации персонажей вводится агрегация с помощью системы слотов (например, оружее, свитки, зелья). Часть поведения будет определяться тем, что игрок установил в своего персонажа. Это дает игроку еще больше возможностей для кастомизации игровой механики персонажа, а разработчикам дает возможность добавлять эти самые модули расширения, что в свою очередь упрощает работу гейм-дизайнерам по выпуску нового контента.

Четвертый уровень. В компоненты можно тоже включить собственную агрегацию, предоставляющую возможность выбора материалов и деталей, из которого собираются эти компоненты. Такой подход даст игроку возможность не только набивать персонажей нужными комплектующими, но и самостоятельно производить эти комплектующие из различных деталек. Такой уровень абстракции в играх редок, и не без резона. Это сопровождается значительным усложнением архитектуры, а регулировка баланса в таких играх превращается в ад.

Пример задачи на построение абстракций

Абстракция 1 слоя

```
class Character {
    int health;
    int attack;
    int armor;
    static size_t count;
}
```

Абстракция 2 уровня

```
class Character {
    int health;
    int attack;
    int armor;
    static size_t count;
}

class Shaman : public Character {
    int mana;
    int totemsHealth;
    int totemsAttack;
}

class Warrior : public Character {
    int weaponDurability;
    int weaponAttack;
    int shieldPoints;
}
```

Абстракция 3 уровня

```
class Character {
    int health;
    int attack;
    int armor;
    static size_t count;

    Backpack objects;           // слоты для объектов
}

class Backpack {
    size_t count;               // количество объектов в рюкзаке
    const size_t capacity;      // вместимость рюкзака (по весу)

    Object* content;            // содержимое рюкзака
}

class Object {                  // игровой объект
    string name;                // название объекта
    const size_t weight;        // вес объекта
}


class Potion : public Object {  // зелья
    string type;                // тип зелья
    int pointsBonus;            // бонусные очки, прибавляемые зельем
    int timeOfAction;           // время действия
}

class Weapon: public Object {   // оружие
    int weaponDurability;       // прочность оружия
    int weaponAttack;           // атака оружия
}


class Shaman : public Character {
    int mana;
    int totemsHealth;
    int totemsAttack;
}

class Warrior : public Character {
    int weaponDurabilityBonus;
    int weaponAttackBonus;
    int shieldPoints;
}
```

Базовые принципы ООП: полиморфизм

Полиморфизм — свойство системы, позволяющее иметь множество реализаций одного интерфейса. Рассмотрим на примере.

Положим, у нас есть три персонажа: воин, маг и шаман. Персонажи у нас в игре боевые, стало быть обладают методом attack(). Игрок, нажимая у себя на джойстике кнопку «воевать», сообщает игре, чтобы та вызвала метод attack() у персонажа, за которого играет игрок. Но поскольку персонажи разные, а игра интересная, каждый из них будет атаковать каким-то своим способом. Например, Олег — объект класса шаман, а шаманы умеют призывать для атаки тотемы.

Польза полиморфизма в данном примере заключается в том, что код игры ничего не знает о реализации его просьбы, кто как должен атаковать, его задача просто вызвать метод attack(), сигнатура которого одинакова для всех классов персонажей. Это позволяет добавлять новые классы персонажей, или менять методы существующих, не меняя код игры. Это удобно.

Базовые принципы ООП: наследование

Наследование — это механизм системы, который позволяет наследовать одними классами свойства и поведение других классов для дальнейшего расширения или модификации.

Что если, мы не хотим играть за одинаковых персонажей, а хотим сделать общий образ, но с разным наполнением? ООП позволяет нам такую шалость путем разделения логики на сходства и различия с последующим выносом сходств в родительский класс, а различий в классы-потомки.

Так, в нашем примере различия между шаманами и воинами заключаются только в том, что шаманы атакуют и защищаются тотемами, а воины атакуют оружием и имеют щит. Все остальные свойства и поведение не будут иметь никакой разницы. В таком случае можно спроектировать систему наследования так: общие черты (здоровье, атака, броня) будут описаны в базовом классе «Персонаж», а различия в двух дочерних классах «Шаман» и «Воин».

Если в классе-потомке переопределить уже существующий метод в классе-родителе, то сработает перегрузка. Это позволяет не дополнять поведение родительского класса, а модифицировать. В момент вызова метода или обращения к полю объекта, поиск атрибута происходит от потомка к самому корню — родителю. То есть, если у шамана вызвать метод Attack(), сначала поиск метода производится в классе-потомке — Shaman, и если его там нет, поиск поднимается на ступень выше — в класс Character.

Любопытно, что чрезмерно глубокая иерархия наследования может привести к обратному эффекту — усложнению при попытке разобраться, кто от кого наследуется, и какой метод в каком случае вызывается. К тому же, не все архитектурные требования можно реализовать с помощью наследования. Поэтому применять наследование следует без фанатизма.

Существуют рекомендации, призывающие предпочитать композицию наследованию там, где это уместно.

Композиция — включение объектом-контейнером объекта-содержимого и управление его поведением; последний не может существовать вне первого.

Как при описании отношений двух сущностей определить, когда уместно наследование, а когда — композиция? Можно воспользоваться популярной шпаргалкой: спросите себя, сущность А является сущностью Б? Если да, то скорее всего, тут подойдет наследование. Если же сущность А является частью сущности Б, то наш выбор — композиция. Применительно к нашей ситуации это будет звучать так: Шаман является персонажем? Да, значит выбираем наследование. Багаж является частью персонажа? Да, значит — композиция.

Эта шпаргалка помогает в большинстве случаев, но бывают и другие факторы, на которые стоит опираться при выборе между композицией и наследованием. Кроме того, эти методы можно комбинировать для решения разного типа задач.

Еще одно важное отличие наследования от композиции в том, что наследование имеет статическую природу и устанавливает отношения классов только на этапе интерпретации / компиляции. Композиция же позволяет менять отношение сущностей в настоящий момент.

Множественное наследование[2]

Мы рассматривали ситуацию, когда два класса унаследованы от общего потомка. Но в некоторых языках можно сделать и наоборот — унаследовать один класс от двух и более родителей, объединив их свойства и поведение. Возможность наследоваться от нескольких классов вместо одного — это множественное наследование.

Дополнительные понятия, о которых стоит знать

Делегация — перепоручение задачи от внешнего объекта внутреннему;

Агрегация — включение объектом-контейнером ссылки на объект-содержимое; при уничтожении первого последний продолжает существование.

Об этих понятиях рекомендуется почитать самостоятельно.

Пример задачи на полиморфизм и наследование

Примечание. Методы и свойства, которые можно переопределить, называются виртуальными. В родительском классе для них указывается модификатор **virtual.** Подробнее наследование и переопределение методов будет рассмотрено дальше. Упростим нашу абстракцию до 2-го уровня.

```
class Character {
    int health;
    int attack;
    int armor;
    static size_t count;

  public:
    virtual void Attack() {
       cout << "Персонаж атакует!" << endl;
    }
}

class Shaman : public Character {
    int mana;
    int totemsHealth;
    int totemsAttack;
   public:
    virtual void Attack() {
       cout << "Шаман призывает боевой тотем!" << endl;
    }
}
```

[2] С данным типом наследования нужно быть аккуратным, оно может повлечь за собой ромбовидную проблему и неразбериху с конструкторами. Помните, что задачи, которые решаются множественным наследованием, можно решать другими механизмами, например, интерфейсом.

```
class Warrior : public Character {
    int weaponDurability;
    int weaponAttackBonus;
    int shieldPoints;
   public:
    virtual void Attack() {
       cout << "Воин ударяет оружием!" << endl;
    }
}
```

Это упрощённый вариант перегрузки функции Attack(). Реализуем вариант, когда атака влияет на статы атакующего и атакуемого персонажей на примере шамана.

```
class Character {
    int health;
    int attack;
    int armor;
    static size_t count;

  public:
    virtual void Attack(Character opponent) {
       cout << "Персонаж атакует!" << endl;
    }

    virtual void WereAttacked(int damage) {
       cout << "Персонаж был атакован!" << endl;
    }
}

class Shaman : public Character {
    int mana;
    int totemsHealth;
    int totemsAttack;
   public:
    virtual void Attack(Character opponent) {
       cout << "Шаман призывает боевой тотем!" << endl;
       totemsHealth = 10;
       totemsAttack = 10;
       mana--;
       opponent.WereAttacked(totemsAttack);
    }
    virtual void WereAttacked(int damage) {
       cout << "Персонаж был атакован!" << endl;
       if (totemsHealth >= demage) {
           totemsHealth -= demage;
       } else {
           if (armor == 0) {
              health -= (demage - totemsHealth);
              totemsHealth = 0;
           } else {
              if (armor >= demage - totemsHealth) {
                  armor -= (demage - totemsHealth);
                  totemsHealth = 0;
              } else {
                  health -= (demage - armor - totemsHealth);
                  armor = 0;
                  totemsHealth = 0;
              }

          }
    }
}
```

Пояснения к данной программе: WereAttacked распределяет урон - сначала он приходится на здоровье тотема, если тотем погибает, проверяется есть ли у персонажа броня, если есть – урон приходится на неё, если брони и тотема уже нет, а урон остался – урон наносится и по здоровью персонажа.

Базовые принципы ООП: инкапсуляция

Инкапсуляция — это контроль доступа к полям и методам объекта. Под контролем доступа подразумевается не только можно/нельзя, но и различные валидации, подгрузки, вычисления и прочее динамическое поведение.

Во многих языках частью инкапсуляции является сокрытие данных. Для этого существуют модификаторы доступа (опишем те, которые есть почти во всех ООП языках):

— public — к атрибуту может получить доступ любой желающий,
— private — к атрибуту могут обращаться только методы данного класса,
— protected — то же, что и private, только доступ получают и наследники класса в том числе.

Как правильно выбрать модификатор доступа? В простейшем случае так: если метод должен быть доступен внешнему коду, выбираем public. В противном случае — private. Если есть наследование, то может потребоваться protected в случае, когда метод не должен вызываться снаружи, но должен вызываться потомками.

Аксессоры (геттеры и сеттеры)

Об этих понятиях мы говорили ранее, рассмотрим немного подробнее.

Геттеры и сеттеры — это методы, задача которых контролировать доступ к полям. Геттер считывает и возвращают значение поля, а сеттер — наоборот, принимает в качестве аргумента значение и записывает в поле. Это дает возможность снабдить такие методы дополнительными обработками. Например, сеттер при записи значения в поле объекта, может проверить тип, или входит ли значение в диапазон допустимых (валидация). В геттер же можно добавить, ленивую инициализацию или кэширование, если актуальное значение на самом деле лежит в базе данных. Применений можно придумать множество.

В некоторых языках есть синтаксический сахар, позволяющий такие аксессоры маскировать под свойства, что делает доступ прозрачным для внешнего кода, который и не подозревает, что работает не с полем, а с методом, у которого под капотом выполняется SQL-запрос или чтение из файла. Так достигается абстракция и прозрачность.

Абстрактные классы

Кроме обычных классов в некоторых языках существуют абстрактные классы. От обычных классов они отличаются тем, что нельзя создать объект такого класса. Зачем же нужен такой класс, спросит читатель? Он нужен для того, чтобы от него могли наследоваться потомки — обычные классы, объекты которых уже можно создавать.

Абстрактный класс наряду с обычными методами содержит в себе абстрактные методы без имплементации (с сигнатурой, но без кода), которые обязан имплементировать программист, задумавший создать класс-потомок. Абстрактные классы не обязательны, но они помогают установить контракт, обязующий имплементировать определенный набор методов, дабы уберечь программиста с плохой памятью от ошибки имплементации.

Заключение

В условиях современных требований наличие в вашем коде слова class не делает из вас ООП-программиста. Если вы не используете механизмы (полиморфизм, композицию, наследование и т. д.), а вместо этого применяете классы лишь для группировки функций и данных, то это не ООП. То же самое можно решить какими-нибудь структурами данных или именными пространствами. Не путайте, иначе на собеседовании будет стыдно.

ООП в целом не стоит применять там, где это бессмысленно или может навредить. Повсеместное применение без знания дела может стать причиной серьезных архитектурных проблем во многих проектах. В проектировании не существует однозначных рецептов на все случаи жизни, где что применять уместно, а где неуместно. Это будет постепенно укладываться в голове с опытом.

Полезные приложения

Плюсы и минусы ООП

Плюсы	Минусы
Легко читается. Не нужно выискивать в коде функции и выяснять, за что они отвечают. Потребляет больше памяти.	Объекты потребляют больше оперативной памяти, чем примитивные типы данных.
Быстро пишется. Можно быстро создать сущности, с которыми должна работать программа.	Снижает производительность. Многие вещи технически реализованы иначе, поэтому они используют больше ресурсов.
Проще реализовать большой функционал. Так как на написание кода уходит меньше времени, можно гораздо быстрее создать приложение с множеством возможностей.	Сложно начать. Парадигма ООП сложнее функционального программирования, поэтому на старт уходит больше времени.
Меньше повторений. Не нужно писать однотипные функции для разных сущностей	