ООП в C++

Класс.
Конструктор копирования.
Конструктор по умолчанию.

НИУ ВШЭ, 2020

Черновая версия регламента

- - Компоненты данных — стиль C++.
- - Операция над объектом — функция-член класса.
- - Дружественная операторная функция для визуализации объекта класса.
- - Конструктор по умолчанию.
- - Конструктор копирования.
- - Сценарий: создание объектов класса — оригинала и его копии; визуализация объектов и выполнение требуемой операции над объектом.

Тигр

- Что умеет делать (методы):
  - бегать (run)
  - охотиться (hunt)

  Свойства
  - размер (size)
  - возраст (age)
  - скорость (speed)
  - окрас шерсти (skin)
  - порода (breed)

Соглашение: в дальнейшем считаем, что тигр бегает только сам по себе (мы не можем заставить его бегать), но можем заставить охотиться. Тогда метод `void run()` будет приватным, а метод `void hunt()` – публичным. Да, пример здесь не совсем удачен, скорее, в качестве приватного метода для тигра можно было бы привести пищеварение или кровообращение, но это излишнее углубление в тему.

- Класс – это чертёж какого-то объекта. Проводя параллель с нашим примером, класс «Тигр» (Tiger) – это чертёж тигра. Он описывает тигра (его внешний вид и другие особенности), а также говорит о том, что тигр может делать. Объект – это конкретный экземпляр класса (конкретный тигр). У класса есть поля (свойства) и методы (то, что может делать объект или то, что можно делать с объектом, в зависимости от фантазии разработчика).

Инкапсуляция

- Более удачный пример инкапсуляции можно привести для моего АТД (кофеварки). Определим для кофеварки методы `void warm (int temp)` и `void makeEspresso(bool hot)` Аргумент `int temp` функции **warm** – температура, до которой мы будем разогревать кофеварку, и аргумент `bool hot` для функции **makeEspresso** (сделать кофе погорячее, или нет).
  Очевидно, что мы можем «попросить» кофеварку сделать кофе путём нажатия кнопки, но мы не можем самостоятельно нагреть её до нужной температуры (она сама нагревается, нагрев выключается автоматически при достижении нужной температуры). Поэтому метод `void warm (int temp)` – приватный, а метод `void makeEspresso(bool hot)`- публичный .

Тигр. Методы (что тигр умеет делать ?)

private — Бегать
void run(int speed);

public — Охотиться
void hunt();

Тигр. Свойства (атрибуты класса)

- Вес (int weight)
- Порода (string breed)
- Размер (int size)
- Окрас шерсти (string skinColour)
- Скорость (int speed)
- голоден ли ? (bool isHungry)

Инкапсуляция. Модификаторы доступа

- **Инкапсуляция** – это ограничение доступа к компонентам объекта, объединение данных и кода, который с ними работает, в единое целое.
  Инкапсуляция позволяет сделать некоторые функции и/или атрибуты класса доступными для использования только внутри класса, а также скрыть детали реализации от пользователя.
  В качестве примера из жизни можно привести кофейный автомат в Вышке – Вы вносите деньги, выбираете напиток и затем получаете напиток. Процесс приготовления напитка от Вас скрыт – Вы подаёте на вход деньги и тип напитка и на выходе получаете заветный стаканчик.

Упрощённо можно сказать, что инкапсуляция – это «заключение» чего-то (какой-то функции или атрибута [свойства] класса) в «чёрный ящик»

Инкапсуляция. Модификаторы доступа

- В языках Java/C++ принцип инкапсуляции реализуется с помощью **модификаторов доступа**. В C++ и Java это public, private и protected.
  Замечание 1.1 Для краткости вместо «методы и атрибуты класса» будем говорить «члены класса».
  Публичные члены класса могут быть вызваны «извне» класса.

```
int main() {
    // Calling default constructor
    Tiger t1(10, 5);
    // Calling public function
    t1.hunt();
}
```

Инкапсуляция. Модификаторы доступа

- В приведённом выше примере мы вызвали публичный метод **hunt** класса **Tiger** .
  Замечание 1.2
  Приватные методы и поля недоступны извне класса. Это значит, что доступ к таким методам и полям имеют только методы данного класса. Такой код вызовет ошибку:

```
int main() {
    // Calling default constructor
    Tiger t1(10, 5);
    // Calling private function
    t1.run();
}
```

Пример 1.1

- **Все члены класса по – умолчанию являются приватными. Внутри класса могут быть объявлены две «секции» - private и public.**

```
class Tiger
{
private:
    int age; // Age of tiger
    int size; // Size of tiger (in metres)
    int speed; // Speed of tiger (metres per second)
    string breed; // Breed (Bengal, Amur or smth like that)
    string skinColour; // Colour of skin (black, white or orange)
    bool isHungry; // A tiger can be hungry or not

    // running
    void run(int t_speed) {}

public:
    // hunting
    void hunt(int t_speed = 30) {}
};
```

Пример 1.2

- **Однако можно объявить только секцию public и вынести туда публичные члены класса (атрибуты и методы)**

```
class Tiger
{
  int age; // Age of tiger
  int size; // Size of tiger (in metres)
  int speed; // Speed of tiger (metres per second)
  string breed; // Breed (Bengal, Amur or smth like that)
  string skinColour; // Colour of skin (black, white or orange)
  bool isHungry; // A tiger can be hungry or not

  // running
  void run(int t_speed) {}

public:
    // hunting
    void hunt(int t_speed = 30) {}
};
```

Все члены класса, не заключенные в «секции» (public, private, protected) по умолчанию являются приватными.

Реализация методов класса

```
class Tiger
{
private:
    int age; // Age of tiger
    int size; // Size of tiger (in metres)
    int speed; // Speed of tiger (metres per second)
    string breed; // Breed (Bengal, Amur or smth like that)
    string skinColour; // Colour of skin (black, white or orange)
    bool isHungry; // A tiger can be hungry or not

    // running
    void run(int t_speed) {
        speed = t_speed;
        cout << "The tiger is running. Speed: " << t_speed << endl;
    }
```

Реализация методов класса

```
public:
    // Friendly operator function for output. Definition.
    friend ostream &operator<<( ostream &output, const Tiger& T );
    // hunting
    void hunt(int t_speed = 30) {
        if (!isHungry) cout << "Tiger is not hungry" << endl;
        else {
            if (age <2) cout << "Tiger is hungry, he makes known about it
tigress" << endl;
            else {
                cout << "Tiger is hungry, it begins to hunt." << endl;
                run(t_speed);
            }
        }
    }
```

Конструкторы

- Конструктор класса – это особый метод класса (или функция), которая создаёт объект класса. Этот метод вызывается автоматически при создании объекта класса. Существует множество способов создать объект, поэтому в языке C++ существуют 3 различных вида конструкторов (конструктор копирования, конструктор перемещения, конструктор по умолчанию). Для лабораторных работ нам понадобятся только два из них: конструктор копирования и конструктор по умолчанию.

Конструктор по умолчанию

- Конструктор по умолчанию, как следует из названия, нужен, чтобы инициализировать (создать) объект на основе заранее заданных аргументов. Т.е. **каждый** аргумент *конструктора по умолчанию* должен иметь значение по умолчанию, однако пользователь вправе переопределить любой из них. В C++ существует правило: имя конструктора и имя класса должны совпадать. Это относится, вообще говоря, к любым конструкторам (как по умолчанию, так и копирования). Конструктор по умолчанию может быть **только один.**

Пример конструктора
по умолчанию с аргументами

```
class Tiger
{
private:
    int age;
    int size;
    int health;
    int damage;
    int speed;
    string breed;
    string name;
    string skinColour;

public:
    Tiger(int t_age=5, int t_size=9, int t_health=100,int t_speed=0, string t_breed="bengal",
string t_name = "Valery", string t_skinColour = "orange") // default constructor with arguments
    {

    }
```

Пример конструктора по умолчанию с аргументами

```
public:
    Tiger(int t_age=5, int t_size=9, int t_health=100,int t_speed=0, string
t_breed="bengal", string t_name = "Valery", string t_skinColour = "orange")
// default constructor with arguments
    {
        age = t_age;
        size= t_size;
        health = t_health;
        speed = t_speed;
        breed = t_breed;
        name = t_name;
        skinColour = t_skinColour;
   }
```

В теле метода (в фигурных скобках) слева от знака равенства – название поля класса (свойства), справа – его значение.

Пример конструктора
по умолчанию без аргументов

```
public:
     Tiger() // default constructor without arguments
       {
    age = 10;
    size = 5;
    health = 50;
    speed = 0;
    breed = "amur";
    name = "Vasya";
    skinColour = "white";
      }
```

Конструктор по умолчанию может и вовсе не иметь аргументов.

Конструктор копирования

В отличии от конструктора по-умолчанию, конструктор копирования принимает в качестве аргумента **константную ссылку (const&)** на объект того же класса и копирует содержимое (поля) одного объекта в другой объект.

Т.е. в качестве «инициализатора» выступают не введённые (или заранее указанные) пользователем данные, а уже существующий объект класса. При выполнении операции присваивания происходит неявный вызов конструктора копирования.

Конструктор копирования

- Заметим, что мы можем не определять конструктор копирования внутри класса, тогда, при копировании объекта, компилятор будет копировать поле за полем (по сути, конструктор копирования нужен в случае, если логика копирования отличается от стандартной, здесь же, при реализации конструктора копирования, мы будем руководствоваться логикой компилятора).

Конструктор копирования

```
class Tiger
{
private:
    …
    …

public:
   Tiger(Tiger const &instance):
    age(instance.age), size(instance.size), health(instance.health),
    speed(instance.speed), breed(instance.breed),
    name(instance.name), skinColour(instance.skinColour)
{

}
…
};
```

Здесь **instance** – объект, который мы будем копировать (оригинал)

Конструктор копирования

```
class Tiger
{
private:
    …
    …

public:
   Tiger(Tiger const &instance):
    age(instance.age), size(instance.size), health(instance.health),
    speed(instance.speed), breed(instance.breed),
    name(instance.name), skinColour(instance.skinColour)
{

}
…
};
```

Здесь **instance** – объект, который мы будем копировать (оригинал)

Дружественная операторная функция

Дружественные функции в C++ - это нарушение принципа инкапсуляции (попытка заглянуть внутрь «чёрного ящика») в частных случаях. С помощью дружественных функций можно разрешить определённым функциям, находящимся вне класса, доступ к приватным полям и методам класса.
Операторная функция – это возможность перегрузить операторы для конкретного класса (заставить их изменить своё поведение в частных случаях). На самом деле, мы уже работали с операторными функциями, когда использовали оператор побитового сдвига для записи в файл.
Операторные функции – это проявление полиморфизма операторов, т.е. один и тот же оператор может вести себя по разному в зависимости от типов операндов (операнды – то, что стоит слева (или справа) от оператора.

Дружественная операторная функция

В дружественную операторную функцию необходимо передавать ссылку на константу `Tiger const &instance` (константой в данном случае является объект класса). Так мы можем гарантировать, что операторная функция не сможет изменить приватные атрибуты класса (в отличии от примера с кроватью, где в операторной функции мы теоретически можем изменить значения приватных полей класса, потому что автор решения забыл сделать ссылку константной). Момент спорный – использовать обычную или константную ссылку – решать Вам.

Пример полиморфизма операторов:

```
a << 3; // сдвиг влево
cout << a; // вывод (отправка в поток)
```

Дружественная операторная функция. Объявление.

```
class Tiger
{
private:
    int age; // Age of tiger
    int size; // Size of tiger (in metres)
    int speed; // Speed of tiger (metres per second)
    string breed; // Breed (Bengal, Amur or smth like that)
    string skinColour; // Colour of skin (black, white or orange)
    bool isHungry; // A tiger can be hungry or not


public:
    // Friendly operator function for output. Definition.
    friend ostream &operator<<( ostream &output, const Tiger& T );
```

Дружественная операторная функция. Реализация

```
// Friendly operation function for output. Implementation.
ostream &operator<<( ostream &output, const Tiger& T ) {
    output << "Breed: " << T.breed << endl;
    output << "Colour of skin: " << T.skinColour << endl;
    output << "Age: " << T.age << endl;
    output << "Speed: " << T.speed << endl;
    output << "Size: " << T.size << endl;
    output << (T.isHungry ? "The tiger is hungry" : "The tiger is not hungry") << endl;
    return output;
}
```

Замечание 1.4: реализация дружественной операторной функции должна находиться **вне класса**, а объявление – внутри класса

Замечание 1.5: можно было бы не возвращать поток, но это приводит к неопределённому поведению (undefined behaviour), т.е. мы не знаем, что произойдёт – скомпилируется этот код или нет, будет ли он работать так, как задумано. Это зависит от особенностей компилятора и «железа» на котором выполняется программа. Неопределённого поведения нужно избегать, не повторяйте моих ошибок.

Демонстрация работы

```
int main() {
    // Calling default constructor
    Tiger t1(10, 5);
    // Calling public function
    t1.hunt();
    // Tiger t2 is a copy of t1
    Tiger t2 = t1;
    // Visualization
    cout << t1;
    cout << t2;
    // Calling method of the class
    t1.hunt(50);
}
```

Результат работы

```
Tiger is not hungry
Breed: bengal
Colour of skin: orange
Age: 10
Speed: 0
Breed: bengal
Colour of skin: orange
Age: 10
Speed: 0
Tiger is not hungry
```