

Национальный исследовательский университет
«Высшая Школа Экономики»
Факультет компьютерных наук

«С++ как первый язык программирования»

Лекция 6

Лекторы: Роман Халкечев, Кирилл Лунев

Цели лекции и семинара

- Продолжить знакомство с пользовательскими типами данных - перечислениями, структурами и классами.
- Научиться создавать пользовательские типы данных и работать с ними.

destructor
private
struct
function
constructor
class
data
C++
type
enum
public

План лекции

- Структуры (ключевое слово **struct**)
- Классы (ключевое слово **class**)

Деструктор

```
struct TBasketballPlayer {  
    ...  
    ~TBasketballPlayer() {}  
};
```

Вызывается при уничтожении объекта.

Обычно нужен для освобождения ресурсов, занимаемых объектом.

Более формально

Деструктор - функция-член структуры, которая вызывается всякий раз, когда уничтожается объект этой структуры.

Деструктор какой-либо структуры имеет имя **~StructName()**, где **StructName** имя структуры, не имеет возвращаемого значения и аргументов.

Каждая структура имеет лишь один ***деструктор***.

Как правило, ***деструктор*** используется для освобождения ресурсов, занятых объектом структуры.

Оператор присваивания

```
struct TBasketballPlayer {  
  
    ...  
  
    TBasketballPlayer& operator=(const TBasketballPlayer& player) {  
  
        Name = player.Name;  
  
        Height = player.Height;  
  
        Weight = player.Weight;  
  
        return *this;  
  
    }  
  
};
```

Важные замечания

- **Конструктор копирования, оператор присваивания и деструктор** имеются в каждой структуре. Если их не определил разработчик структуры, это делает компилятор.
- Если разработчик не создал вообще ни одного конструктора, то компилятор так же создаст **конструктор по умолчанию** (не принимающий аргументов). Он вызывает конструкторы по умолчанию каждого из членов структуры.

Интегральные типы и семантика конструктора

```
TBasketballPlayer commander("Michael Jordan", 1.98, 98.0);
```

```
TBasketballPlayer defender(commander);
```

```
int first;
```

```
first = 10;
```

```
int second = 10;
```

```
int third(10);
```


Более правильная реализация конструктора

<pre>TBasketballPlayer() { Name = "Noname"; Height = 0.0; Weight = 0.0; }</pre>	<pre>TBasketballPlayer() : Name("Noname") , Height(0.0) , Weight(0.0) {}</pre>
---	--

Плохо

Хорошо

Более правильная реализация конструктора

Версия конструктора с аргументами

```
TBasketballPlayer(const string& name, double height, double weight)  
: Name(name), Height(height), Weight(weight) {}
```

Важное замечание:

Поля инициализируются ровно в той последовательности, в которой они объявлены в структуре.

Бывает, что не всё так просто

```
struct TBasketballPlayer {  
  
    ...  
  
    int Id;  
  
};  
  
TBasketballPlayer(const string& name, double height, double weight, int id)  
: Name(name), Height(height), Weight(weight), Id(playerId) {  
  
    PeriodicallyUpdate(); // Периодическое обновление данных по id игрока  
  
}
```

Вопрос

Где должна быть объявлена функция `PeriodicallyUpdate()`?

Например, как часть структуры TBasketballPlayer

```
struct TBasketballPlayer {  
  
    std::string Name;  
  
    double Height;  
  
    double Weight;  
  
    int Id;  
  
    TBasketballPlayer(const string& name, double height, double weight, int  
id);  
  
    void PeriodicallyUpdate();  
  
};
```

Квалификаторы доступа

- Функция PeriodicallyUpdate() нужна для поддержания актуальной информации об игроке. Как именно мы поддерживаем информацию в актуальном состоянии не должно интересовать конечного пользователя нашего класса. Это лишь деталь реализации, поэтому доступ до неё должен быть скрыт.
- Что если мы хотим всегда получать данные об игроках из базы данных? Как обезопасить себя и пользователя от непреднамеренной порчи данных? Доступ до данных должен быть ограничен.

Для решения этих проблем и существуют квалификаторы доступа.

Квалификаторы доступа

```
struct TBasketballPlayer {  
  
    private:  
  
        std::string Name;  
  
        double Height;  
  
        double Weight;  
  
        int Id;  
  
        void PeriodicallyUpdate();  
  
    public:  
  
        TBasketballPlayer(int id);  
  
};
```

Доступ к данным через открытый интерфейс

```
struct TBasketballPlayer {  
  
    ...  
  
public:  
  
    std::string& GetName() const {  
  
        return Name;  
  
    }  
  
    ...  
  
};
```


Квалификаторы доступа

- Теперь данные в безопасности. Объект класса не может прийти в несогласованное состояние.
- Реализация скрыта от пользователя, пользователь видит только открытый интерфейс. Мы вольны менять реализацию сколько угодно без правок клиентского кода.
- Всё, что только возможно нужно делать `private`.

Классы

- Класс отличается от структуры лишь тем, что все его поля по умолчанию скрыты, а у структуры наоборот - открыты. В остальном они идентичны.
- Следуя идеологии ООП, нужно разделить открытый интерфейс от деталей реализации разрабатываемого типа данных. Это пример *инкапсуляции*. Поэтому, обычно для больших и сложных пользовательских типов используют классы.
- Структуры бывают полезными для небольших или вспомогательных типов данных.

```
class TBasketballPlayer {  
  
    std::string Name;  
  
    double Height;  
  
    double Weight;  
  
    int Id;  
  
    void PeriodicallyUpdate();  
  
public:  
  
    TBasketballPlayer(int id);  
  
    std::string& GetName() const;  
  
    ...  
  
};
```

```
int playerId = 101;

TBasketballPlayer player(playerId);

std::vector<TBasketballPlayer> team;

team.push_back(player);

std::cout << team[0].GetName() << std::endl;
```

Квалификаторы доступа

- Данные и методы класса с квалификатором `public` - доступны извне (любому пользователю класса).
- Данные и методы класса с квалификатором `private` - доступны только внутри (только функциям класса).
- Есть еще квалификатор доступа `protected`, для управления доступом при наследовании - его изучим позднее (доступ есть у самого класса и у всех наследников этого класса).

Вопросы?

Домашнее задание

- Прочсть главу 10 в книге Стивена Прата «Язык программирования C++»
- Лабораторная работа с прошлого семинара