

Национальный исследовательский университет
«Высшая Школа Экономики»
Факультет компьютерных наук

«С++ как первый язык программирования»

Лекция 5

Лекторы: Роман Халкечев, Кирилл Лунев

Цели лекции и семинара

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



A word cloud of C++ keywords. The words are arranged in a cluster, with 'destructor' at the top, 'function' on the left, 'class' on the right, and 'public' at the bottom. Other words like 'private', 'struct', 'constructor', 'data', 'type', 'enum', and 'C++' are interspersed among them. The colors range from light blue to green.

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

План лекции

- Перечисления (ключевые слова **enum**, **enum class**)
- Структуры (ключевое слово **struct**)
- Классы (ключевое слово **class**)

Задача

В процессе разработки программы для службы доставки, Вам оказалось необходимым адаптировать логику многих функций в зависимости от дня недели.

Например, цена доставки может зависеть от того в будний или выходной день осуществляется эта самая доставка.

Предложите метод хранения данных о дне недели в Вашей программе.



Пречисления

Перечисление - пользовательско-интегральный тип, содержащий именованные целочисленные константы

```
enum Days {Mo, Tu, We, Th, Fr, Sa, Su};
```

Как будто бы объявили 7 целочисленных констант.

```
Days day = Mo;
```

Значения - это порядковый числовой номер, по умолчанию начиная с нуля.

```
int x = day;
```

```
Days anotherDay = static_cast<Days>(2);
```

Примеры использования

```
bool IsWeekend(Days day) {  
    return day == Sa || day == Su;  
}
```

**Next weekend is weekend
again...!
JUHUUUUUUUUU...!**



```
enum Countries {Belarus, Kazakhstan, Russia, Turkey, Ukraine};

void RunCurrencyConverter(Countries country) {

    switch (country) {

        case Russia:

            //run currency converter for Russia

            break;

        case Belarus:

            //run currency converter for Belarus

            ...

    }

}
```

Недостатки традиционного `enum`

- Экспорт значений в окружающую область видимости (может привести к конфликту имён)
- Неявно преобразовываются в целый тип

```
enum Days {Mo, Tu, We, Th, Fr, Sa, Su};
```

```
int main() {  
    Days day = Mo;  
  
    int x = day;  
  
}
```


Перечисления в C++11

В C++11 введены **strongly-typed enums** - `enum class`.

- Больше не экспортируют свои значения в окружающую область видимости
- Больше не преобразуются неявно в целый тип

```
enum class Days {Mo, Tu, We, Th, Fr, Sa, Su};
```

```
int main() {  
    Days day = Days::Mo; // Вместо: Days day = Mo  
  
    int x = static_cast<int>(day); // Вместо: int x = day;  
  
}
```

Задача

В процессе разработки программы для NBA, Вам оказалось необходимым оперировать данными о баскетболистах - имя, рост и вес.

Предложите метод хранения этих данных в Вашей программе.



Структуры

Структура - пользовательский тип данных, объединяющий несколько переменных (возможно разных типов), а так же функций под одним именем.

Объявление структуры:

```
struct TBasketballPlayer {  
    std::string Name;  
  
    double Height;  
  
    double Weight;  
  
};
```

Создание переменных (объектов) типа структуры

Теперь можно создавать переменные типа `TBasketballPlayer` так же, как переменные встроенного типа: `double`, `char` и т.д.

```
TBasketballPlayer player;
```

```
TBasketballPlayer* pointer = &player;
```

```
TBasketballPlayer& reference = player;
```

```
TBasketballPlayer team[12];
```

Обращение к элементам объекта структуры

Для обращения к отдельным элементам структуры нужно использовать оператор принадлежности точка (.):

```
TBasketballPlayer player;
```

```
player.Name = "Michael Jordan";
```

```
player.Height = 1.98;
```

```
player.Weight = 98.0;
```

Обращение к элементам объекта структуры

Для обращения к отдельным элементам структуры через указатель на экземпляр структуры нужно использовать оператор стрелочка (->):

```
TBasketballPlayer player;
```

```
TBasketballPlayer* pointer = &player;
```

```
(*pointer).Name = "Michael Jordan";
```

```
pointer->Name = "Michael Jordan";
```

```
pointer->Height = 1.98;
```

```
pointer->Weight = 98.0;
```

Инициализация элементов объекта структуры

Инициализировать элементы структуры можно так:

```
TBasketballPlayer player;
```

```
player = { "Michael Jordan", 1.98, 98.0 };
```

Или при создании:

```
TBasketballPlayer commander = { "Michael Jordan", 1.98, 98.0 };
```

Важно при создании с квалификатором `const`:

```
const TBasketballPlayer defender = { "Sam Bowie", 2.16, 107.0 };
```

Если нам понадобилось написать функцию, обрабатывающую данные об игроке, то можем сделать так:

```
void Eat(TBasketballPlayer* player, double weight) {  
    player->Weight += weight;  
}
```

...

```
TBasketballPlayer player = { "Michael Jordan", 1.98, 98.0 };  
Eat(&player, 1.5);
```


Лучше сделать её частью структуры:

```
struct TBasketballPlayer {  
    std::string Name;  
  
    double Height;  
  
    double Weight;  
  
    void Eat();  
};  
  
void TBasketballPlayer::Eat(double weight) {  
    Weight += weight;  
}
```

Доступ к методам объекта структуры осуществляется так же, как к данным:

```
TBasketballPlayer player = { "Michael Jordan", 1.98, 98.0 };
```

```
player.Eat(1.5);
```

```
TBasketballPlayer* pointer = &player;
```

```
pointer->Eat(1.5);
```

Было:

```
void Eat(TBasketballPlayer* player, double weight);  
  
Eat(&player, 1.5);
```

Стало:

```
void TBasketballPlayer::Eat(double weight);  
  
player.Eat(1.5);
```

Как функция понимает у какого объекта она вызвана?

Благодаря неявному указателю на вызывающий объект: `this`

Ключевое слово this

```
struct TBasketballPlayer {  
    std::string Name;  
  
    double Height;  
  
    double Weight;  
  
    void Eat(double weight);  
};
```



TBasketballPlayer* const this

Ключевое слово this

На самом деле эти две реализации эквивалентны:

```
void TBasketballPlayer::Eat(double weight) {  
    Weight += weight;  
}
```

```
void TBasketballPlayer::Eat(double weight) {  
    this->Weight += weight;  
}
```

Ключевое слово this

Без this не обойтись, когда нужно вернуть ссылку на сам объект:

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

Константные методы

У константных объектов могут быть вызваны только те методы, которые их не меняют:

```
const TBasketballPlayer defender = { "Sam Bowie", 2.16, 107.0 };  
  
defender.Eat(1.5); // Ошибка компиляции
```

Константные методы

Опишем функцию, не меняющую объект:

```
struct TBasketballPlayer {  
    ...  
    void Play();  
};  
  
void TBasketballPlayer::Play() {  
    std::cout << "I'm playing" << std::endl;  
}
```


Константные методы

```
const TBasketballPlayer defender = { "Sam Bowie", 2.16, 107.0 };  
defender.Play(); // Ошибка компиляции
```

Компилятор сам не может понять - меняет метод объект или нет:

Поэтому необходимо явно указывать компилятору на то, что метод константный - то есть не меняет объект.

Это делается с помощью ключевого слова `const`.


Константные методы


Опишем функцию, не меняющую объект:


```
struct TBasketballPlayer {  
    ...  
    void Play() const;  
};  
  
void TBasketballPlayer::Play() const {  
    std::cout << "I'm playing" << std::endl;  
}
```

Константные методы

```
struct TBasketballPlayer {  
    std::string Name;  
    double Height;  
    double Weight;  
    void Eat(double weight);  
    void Play() const;  
};
```

 TBasketballPlayer* const this

 Ключевое слово const

 const TBasketballPlayer* const this

Новое назначение ключевого слова `const`:

```
struct TBasketballPlayer {  
    std::string Name;  
  
    double Height;  
  
    double Weight;  
  
    void Play() const;  
};  
  
void TBasketballPlayer::Play() const {  
    std::cout << "I'm playing" << std::endl;  
}
```

Значения элементов структуры по умолчанию

```
struct TBasketballPlayer {  
    std::string Name = "Noname";  
    double Height = 0.0;  
    double Weight = 0.0;  
};  
  
TBasketballPlayer player;  
  
player.Name == "Noname"; // True
```

Конструктор

```
struct TBasketballPlayer {  
  
    ...  
  
    TBasketballPlayer() {  
  
        Name = "Noname";  
  
        Height = 0.0;  
  
        Weight = 0.0;  
  
    }  
  
};
```

Примеры

```
TBasketballPlayer defender;
```

```
TBasketballPlayer commander = TBasketballPlayer();
```

```
TBasketballPlayer team[12];
```

```
defender.Name == "Noname"; // True
```

```
commander.Name == "Noname"; // True
```

```
team[0].Name == "Noname"; // True
```

```
struct TBasketballPlayer {  
  
    ...  
  
    TBasketballPlayer(const std::string& name, double height, double weight);  
  
};  
  
TBasketballPlayer::TBasketballPlayer(const std::string& name, double height,  
                                       double weight) {  
  
    Name = name;  
  
    Height = height;  
  
    Weight = weight;  
  
}
```


Теперь создавать переменные стало проще:

```
TBasketballPlayer defender = TBasketballPlayer("Sam Bowie", 2, 99);
```

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

Конструктор по умолчанию по-прежнему нужен:

```
TBasketballPlayer player;
```

```
TBasketballPlayer team[12];
```

Конструктор копирования

```
struct TBasketballPlayer {  
  
    ...  
  
    TBasketballPlayer(const TBasketballPlayer& player);  
  
};  
  
TBasketballPlayer::TBasketballPlayer(const TBasketballPlayer& player) {  
  
    Name = player.Name;  
  
    Height = player.Height;  
  
    Weight = player.Weight;  
  
}
```

Создание объекта - копии другого объекта:

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

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

Передача в функцию по значению:

```
void foo(TBasketballPlayer player) {
```

```
    ...
```

```
}
```

```
foo(commander);
```

Более формально

Конструктор - функция-член структуры, которая вызывается всякий раз, когда создается объект этой структуры.

Конструктор какой-либо структуры имеет то же имя, что и его структура и не имеет возвращаемого значения.

Благодаря перегрузке функций есть возможность иметь сразу несколько **конструкторов** с различными сигнатурами.

Как правило, **конструктор** используется для инициализации элементов объекта структуры.

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 отделить 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 главы 9-10 в книге Стивена Прата «Язык программирования С++»
- Лабораторная работа с прошлого семинара