Основы языков C/C++

Кафедра ЭО

Петрухин О.М.

Представьте, что вы хотите хранить информацию о студенте: имя, возраст и номер группы.

```
int main(){
    std::string name;
    int age;
    int group_number;
}
```

А теперь ещё о двух

```
int main(){
    std::string name1;
    int age1;
    int group_number1;

    std::string name2;
    int age2;
    int group_number2;

    std::string name3;
    int age3;
    int group_number3;
}
```

Структуры

Вместо повторения одинаковых данных C++ предлагает инструменты для **создания** новых типов данных — **структуры** и **классы**.

Структура — это пользовательский тип данных, который позволяет объединить несколько **переменных (полей)** разных типов в одну сущность.

Данные, которые содержит в себе структура, называются полями. Каждое поле — это как переменная, только существующая в рамках структуры. У структуры есть **определение** — в нём описывается, из каких полей состоит структура — и **экземпляры**, построенные с помощью определения

- Структура определяется с помощью ключевого слова `struct`.
- Поля структуры можно использовать как обычные переменные.
- Структуры могут содержать данные разных типов.

Структуры

Синтаксис

Определение структуры имеет следующий синтаксис:

```
struct <название структуры>
{
    <тип данных поля 1> <название поля 1>;
    <тип данных поля 2> <название поля 2>;
    …
    <тип данных поля N> <название поля N>;
};
```

Как работать со структурами

Сама структура — это новый, пользовательский (то есть созданный пользователем языка) тип данных, как `std::string`, `int`, `float` и другие.

Переменные, которые имеют тип данных созданной вами структуры, называются экземплярами этой структуры. Экземпляр структуры создаётся точно так же, как переменная любого другого типа. Нужно просто указать тип структуры и имя переменной, тогда будет создан экземпляр структуры

```
struct Student{
    std::string name;
    int age;
    int group_number;
};

int main(){
    Student student1;    // Теперь в переменной student1 хранится экземпляр
                         // структуры Student
    Student student2;
    Student student3;
}
```

Инициализация экземпляра структуры

При создании экземпляра структуры его поля будут содержать значения по умолчанию. Чтобы сразу заполнить поля экземпляра структуры нужными значениями, можно использовать синтаксис инициализации структуры.

Для инициализации структуры после объявления переменной структуры нужно написать знак присвоения (знак =) и список инициализации (в фигурных скобках). В списке инициализации нужно указать значения для всех полей в том порядке, в котором они определены в вашей структуре

```
struct Student{
    std::string name;
    int age;
    int group_number;
};

int main(){
    Student student1 = {"Дмитрий", 22, 6};

    return 0;
}
```

Доступ к полям структуры

Для доступа к полям структуры используется оператор точка (.). Этот оператор позволяет читать или изменять значения полей конкретного объекта структуры.

```
struct Student{
    std::string name;
    int age;
    int group_number;
};

int main(){
    Student student1 = {"Дмитрий", 22, 6};

    student1.age = 23;
    student1.group_number = 2;
    std::cout << "Имя: " << student1.name;
    std::cout << ", возраст - " << student1.age << std::endl;
    std::cout << "Номер группы: " << student1.group_number;

    return 0;
}
```

Доступ к полям структуры

Переменная, содержащая экземпляр структуры, ведёт себя так же, как другие переменные, и тоже хранится в памяти — а значит, имеет адрес. К переменной структуры можно применить оператор взятия адреса (**&**) и получить указатель на структуру.

```
struct Student{
    std::string name;
    int age;
    int group_number;
};

int main(){
    Student student1 = {"Дмитрий", 22, 6};

    Student* p_student = &student1;

    p_student->age = 23;
    p_student->group_number = 2;

    return 0;
}
```

Классы

Класс — это более продвинутая версия структуры. Он не только объединяет данные, но и предоставляет методы (функции) для работы с этими данными. Классы позволяют реализовать инкапсуляцию, наследование и полиморфизм — основные принципы ООП.

Синтаксис

Класс объявляется с помощью ключевого слова **class**. Синтаксис объявления выглядит следующим образом:

```
class <название класса>
{
public:
    <члены класса>
};
```

Чтобы иметь возможность обращаться к членам класса за его пределами, нужно в самом начале класса указать “public:”

Классы

Вернемся к примеру со студентами:

```
class Student {
public:
    std::string name;
    int age;
    int groupNumber;

    // Метод для вывода информации
    void displayInfo() {
        std::cout << "Имя: " << name << std::endl;
        std::cout << "Возраст: " << age << std::endl;
        std::cout << "Группа: " << groupNumber << std::endl;
    }
};

int main() {
    Student student1;
    student1.name = "Мария";
    student1.age = 22;
    student1.groupNumber = 102;

    // Вызываем метод
    student1.displayInfo();

    return 0;
}
```

Классы

Класс, как и структура - это новый, пользовательский тип данных. В объявлении класса описывается его устройство. Конкретные экземпляры, называются экземпляры класса, или **объекты**.

Объектов может существовать много, тогда как объявление класса - всего одно

Зачем нужны поля?

Поля используются для того, чтобы хранить данные, специфичные для одного экземпляра класса

В дальнейшем данные, хранящиеся в этих полях, используются внешним по отношению к классу кодом (любым, кто использует этот класс) или внутренним (методами - с ними сейчас познакомимся)

Методы – это функции, “принадлежащие” экземпляру класса (объекту)

Имея на руках экземпляр класса, можно вызвать его метод с помощью оператора . (точка)

Классы

Рассмотрим ещё один пример:

```
class Person
{
public:
    std::string first_name;
    std::string last_name;
    int age;

    void print_person(){
        std::cout << "Имя: " << first_name << " Фамилия: " << last_name <<
        "Возраст: " << age << std::endl;
    }
};

int main(){
    Person person;
    person.print_person(); // Имя: Фамилия: Возраст: -858993460
}
```

Конструктор класса

В предыдущем примере мы получали очень странный результат после вызова метода *print_person*. При этом если вы забудете присвоить значение какому-то важному полю после создания объекта, вы можете получить неправильное поведение целого объекта.

Для того, чтобы избежать таких ситуаций, существуют **конструкторы**.

Конструктор — это специальный метод класса, который автоматически вызывается при создании объекта. Его основная задача — инициализировать поля класса или выполнить другие подготовительные действия.

Особенности конструктора:

- Имя конструктора совпадает с именем класса.
- Конструктор не имеет возвращаемого типа (даже `void`)
- Класс обязан иметь хотя бы один конструктор. При этом конструкторов у класса может быть несколько
- Конструкторы одного и того же класса различаются между собой количеством и типом параметров

Конструктор класса

Обобщённый синтаксис конструктора представлен ниже

```
class <имя класса>
{
    <имя класса>([<аргументы>])
    {
        <тело конструктора>
    }
}
```

Создадим конструктор с параметрами для нашего класса Person

```
class Person{
public:
    std::string first_name;
    std::string last_name;
    int age;
    Person(std::string first_name, std::string last_name, int age){
        this->first_name = first_name;  // инициализация
        this->last_name = last_name;    // инициализация
        this->age = age;    // инициализация
    } // это конструктор с параметрами
};
```

Ключевое слово this

Есть ещё одна особенность методов - в них доступно ключевое слово `this`.

С помощью этого ключевого слова можно получить доступ к объекту, у которого был вызван этот метод.

Ключевое слово `this` имеет тип указателя на тот класс, в рамках которого оно используется. Поэтому для получения доступа к членам объекта, на который указывает `this`, нужно использовать оператор -> (стрелочка)

```
class Person{
public:
    std::string first_name;
    std::string last_name;
    int age;

    void change_age(int age){
        age = age; // такой код будет
                   // работать неправильно
    }
};
```

```
class Person{
public:
    std::string first_name;
    std::string last_name;
    int age;
    void change_age(int age){
        this->age = age; // тут всё правильно
    }
};
```

Конструктор по умолчанию

Если в классе программист не написал явно ни одного конструктора, то компилятор сам сгенерирует **конструктор по умолчанию (default constructor)** – пустой, без параметров.

А вот если программист написал сам хотя бы один конструктор, то конструктор по умолчанию сгенерирован уже не будет.

```
class Person{
public:
    std::string first_name;
    std::string last_name;
    int age;
    Person(std::string first_name, std::string
last_name, int age){
        ...
    } // это конструктор с параметрами
};

int main(){
    Person person; // Ошибка!
}
```

Конструктор по умолчанию

Решение 1:

```
class Person{
public:
    std::string first_name;
    std::string last_name;
    int age;
    Person(std::string first_name,
std::string last_name, int age) { ... }
    // это конструктор с параметрами
};

int main(){
    Person person("Пётр", "Петров", 30); //
Теперь собирается
}
```

Решение 2:

```
class Person{
public:
    ...
    Person(std::string first_name, std::string
last_name, int age) { ... }
    // это конструктор с параметрами

    Person(){
        first_name = "Неизвестно";
        last_name = "Неизвестна";
        age = 18;
    } // это конструктор без параметров
};

int main(){
    Person person; // Ошибка!
}
```

Модификаторы доступа

Модификаторы доступа для членов класса указываются в рамках класса, после них ставится двоеточие.

Указанный модификатор применяется ко всем членам класса, расположенным ниже него и выше следующего модификатора доступа (если таковой имеется).

```
class MyClass {
public:
    <член 1>
    <член 2>
private:
    <член 3>
    <член 4>
public:
    <член 5>
}
```

Модификаторы доступа private

Модификатор доступа `private` означает, что помеченные им члены доступны только коду внутри класса.

То есть если вы пометили поле класса как `private` - обратиться к этому полю (то есть считать из него значение или записать в него новое значение) можно будет только изнутри самого класса (а именно из методов или конструкторов)

```
class TestClass{
private:
    int priv_field;
    void priv_method() { std::cout << priv_field; } // можно
};

int main(){
    TestClass test;
    std::cout << test.priv_field; // ошибка доступа!
    test.priv_method(); // ошибка доступа!
}
```

Модификаторы доступа public

Модификатор доступа `public` означает, что помеченные им члены доступны только коду внутри класса.

То есть если вы пометили поле класса как `public` - обратиться к этому полю (то есть считать из него значение или записать в него новое значение) можно будет только изнутри самого класса (а именно из методов или конструкторов)

```
class TestClass{
public:
    int pub_field;
    void pub_method() { std::cout << pub_field; } // можно
};

int main(){
    TestClass test;
    std::cout << test.pub_field; // можно
    test.pub_method(); // можно
}
```

КАИ
Крылья для жизни!

Спасибо за внимание!