Модуль 8

- Классы, объекты и методы
- Спецификаторы доступа
- Инкапсуляция
- Конструкторы и деструкторы
- Статические поля и методы
- Константность объектов

Классы, объекты и методы

Класс - это шаблон или формальное описание для создания пользовательских объектов. Он определяет состояние (поля) и поведение (методы) объектов определенного типа. Класс можно рассматривать как тип данных, который содержит как данные, так и функции для работы с этими данными.

Использование ключевого слова class определяет новый **пользовательский тип данных — класс**.

Классы, объекты и методы

```
struct DateStruct
{
    int day;
    int month;
    int year;
};
```

```
class DateClass
{
public:
    int m_day;
    int m_month;
    int m_year;
};
```

`DateClass today{};` – ***«объект»*** **класса -** переменная класса

Классы, объекты и методы

```
struct Employee
{
    short id;
    int age;
    double salary;
};
```

```
Employee john;
john.id = 8;
john.age = 27;
john.salary = 32.17;

Employee james;
james.id = 9;
james.age = 30;
james.salary = 28.35;
```

Классы, объекты и методы

Методы классов

```
class DateClass
{
public:
    int m_day;
    int m_month;
    int m_year;
    void print() // определяем функцию-член
    {
        std::cout << m_day << "/" << m_month << "/" << m_year;
    }
};
```

Классы, объекты и методы

class vs **struct**

1) Во всех наших примерах class можно заменить на struct.
2) struct - из C, class - C++
3) struct для хранения данных, class - реализация сложных объектов
4) struct - модификатор доступа public, class - private.
5) class - RAII, struct - нет

Спецификаторы доступа

- ➔ **спецификатор public** делает члены открытыми;
- ➔ **спецификатор private** делает члены закрытыми;
- ➔ **спецификатор protected** открывает доступ к членам только для дружественных и дочерних классов (детально об этом на соответствующем уроке).

Правило:

Устанавливайте спецификатор доступа private переменным-членам класса и спецификатор доступа public — методам класса (если у вас нет веских оснований делать иначе).

Инкапсуляция

Зачем делать переменные-члены класса закрытыми?

В объектно-ориентированном программировании **инкапсуляция** (или ***«сокрытие информации»***) — это процесс скрытого хранения деталей реализации объекта. Пользователи обращаются к объекту через открытый интерфейс.

Инкапсуляция

Плюсы использования инкапсуляции

- Инкапсулированные классы проще в использовании и уменьшают сложность программ.
- Инкапсулированные классы помогают защитить ваши данные и предотвращают их неправильное использование.

```
class MyString
{
    char *m_string; // динамически выделяем строку
    int m_length; // используем переменную для отслеживания длины строки
};
```

Инкапсуляция

- Инкапсулированные классы легче изменить.
- С инкапсулированными классами легче проводить отладку.

Инкапсуляция

Функции доступа (геттеры и сеттеры)

- **геттеры** — это функции, которые возвращают значения закрытых переменных-членов класса;
- **сеттеры** — это функции, которые позволяют присваивать значения закрытым переменным-членам класса.

➔ **Предоставляйте функции доступа только в том случае, когда нужно, чтобы пользователь имел возможность получать или присваивать значения членам класса.**

➔ **Геттеры должны использовать тип возврата по значению или по константной ссылке. Не используйте для геттеров тип возврата по неконстантной ссылке.**

Конструкторы

Конструктор — это особый тип метода класса, который автоматически вызывается при создании объекта этого же класса.

Конструкторы обычно используются для инициализации переменных-членов класса значениями, которые предоставлены по умолчанию/пользователем, или для выполнения любых шагов настройки, необходимых для используемого класса (например, открыть определенный файл или базу данных).

- ➔ конструкторы всегда должны иметь тоже имя, что и класс (учитываются верхний и нижний регистры);
- ➔ конструкторы не имеют типа возврата (даже void).

Конструкторы

Виды конструкторов

- **конструктором по умолчанию**

```
MyClass() {}
```

- **Конструкторы с параметрами**

```
MyClass(int x, double y) : a(x), b(y) {}
```

- **Конструктор копирования**

```
MyClass(const MyClass& other) : a(other.a), b(other.b) {}
```

- **Конструктор перемещения**

```
MyClass(MyClass&& other) noexcept : a(std::move(other.a)), b(std::move(other.b)) {}
```

Конструкторы

Неявный конструктор

```
class Date
{
private:
    int m_day = 12;
    int m_month = 1;
    int m_year = 2018;
};
```

```
class Date
{
private:
    int m_day = 12;
    int m_month = 1;
    int m_year = 2018;
public:
    Date() // неявно генерируемый конструктор
    {
    }
};
```

Конструкторы

Делегирующий конструктор

```
class Boo
{
private:
public:
    Boo()
    {
        // Часть кода X
    }
    Boo(int value)
      : Boo() // используем конструктор по умолчанию Boo() для
выполнения части кода X
    {
        // Часть кода Y
    }
};
```

Конструкторы

Инициализация

```
double value2(4.5); // прямая инициализация
char value3 {'d'} // uniform-инициализация
```

Uniform-инициализация предотвращает неявное преобразование, то есть ошибки при преобразовании, которые могут привести к потере данных или непреднамеренному поведению.
Например, если попытаться инициализировать целочисленную переменную типа `int` значением с плавающей точкой, прямая инициализация допустит это, но uniform-инициализация вызовет ошибку компиляции.

Конструкторы

```
class Distance {
private:
    int meters;
public:
    explicit Distance(int m) : meters(m) {} // конструктор с явным
указанием ключевого слова explicit

    int getMeters() const {
        return meters;
    }
};
```

Конструкторы

Default

```
class MyClass {
public:
    // Использование default для генерации конструктора по умолчанию
    MyClass() = default;

    // Другие конструкторы
    MyClass(int value) { /*...*/ }

    // деструктор
    ~MyClass() = default;
};
```

Конструкторы

delete

```
class MyClass {
public:
    MyClass(const MyClass&) = delete; // Запрещение конструктора
копирования
    void operator=(const MyClass&) = delete; // Запрещение оператора
присваивания
};
```

Деструкторы

Деструктор — 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.

- ➔ деструктор должен иметь тоже имя, что и класс, со знаком тильда (~) в самом начале;
- ➔ деструктор не может принимать аргументы;
- ➔ деструктор не имеет типа возврата.
- ➔ Для каждого класса дектруктор может быть только один

Деструкторы

Идиома программирования RAII

Идиома RAII (англ. *«**R**esource **A**cquisition **I**s **I**nitialization» = «Получение ресурсов есть инициализация»*) — это идиома **объектно-ориентированного** программирования, при которой использование ресурсов привязывается к времени жизни объектов с **автоматической продолжительностью жизни**.

Деструкторы

Функция exit()

При вызове **exit** программа завершится, и никакие деструкторы не будут вызваны!!!

Правила использования конструкторов

1. Правило 3 (Rule of Three): Это правило утверждает, что если класс определяет один из следующих специальных методов-членов, то вероятно потребуется определить и реализовать другие два:
    - Деструктор
    - Конструктор копирования
    - Оператор присваивания копирования
2. Правило 4 (Rule of Four): Это расширение "Правила 3" для классов, использующих конструктор перемещения и оператор присваивания перемещения. Если класс определяет один из следующих специальных методов-членов, то вероятно потребуется определить и реализовать другие три:
    - Деструктор
    - Конструктор копирования
    - Оператор присваивания копирования
    - Конструктор перемещения
    - Оператор присваивания перемещения
3. Правило 0 (Rule of Zero): Это принцип, согласно которому класс должен стараться не определять специальные методы-члены управления ресурсами (конструкторы, деструкторы, операторы присваивания), а вместо этого полагаться на правильное использование умных указателей, контейнеров стандартной библиотеки и других RAII (Resource Acquisition Is Initialization) ресурсов для автоматического управления памятью и ресурсами.
4. Правило 6 (Rule of Six): Это аналогичное "Правилу 4", но с учетом перемещения. Оно утверждает, что если класс определяет один из следующих специальных методов-членов, то вероятно потребуется определить и реализовать другие пять:
    - Деструктор
    - Конструктор копирования
    - Оператор присваивания копирования
    - Конструктор перемещения
    - Оператор присваивания перемещения
5. Правило 5 (Rule of Five): Если класс определяет один из следующих специальных методов-членов, то вероятно потребуется определить и реализовать другие четыре:
    - Деструктор
    - Конструктор копирования
    - Оператор присваивания копирования
    - Конструктор перемещения
    - Оператор присваивания перемещения

Статические поля и методы

Константность объектов