# Лекция 10. static и inline

Александр Смаль

**CS центр**
24 октября 2017
Санкт-Петербург

# Глобальные переменные

Объявление глобальной переменной:

```
extern int global;

void f () {
    ++global;
}
```

Определение глобальной переменной:

```
int global = 10;
```

Проблемы глобальных переменных:

- Масштабируемость.
- Побочные эффекты.
- Порядок инициализации.

# Статические глобальные переменные

Статическая глобальная переменная — это глобальная переменная, доступная только в пределах модуля.

Определение:

```
static int global = 10;

void f () {
    ++global;
}
```

Проблемы статических глобальных переменных:

- Масштабируемость.
- Побочные эффекты.

# Статические локальные переменные

Статическая локальная переменная — это глобальная переменная, доступная только в пределах функции.

Время жизни такой переменной — от первого вызова функции `next` до конца программы.

```
int next(int start = 0) {
    static int k = start;
    return k++;
}
```

Проблемы статических локальных переменных:

- Масштабируемость.
- Побочные эффекты.

# Статические функции

Статическая функция, доступная только в пределах модуля.

Файл 1.cpp:

```
static void test() {
    cout << "A\n";
}
```

Файл 2.cpp:

```
static void test() {
    cout << "B\n";
}
```

Статические глобальные переменные и статические функции проходят *внутреннюю линковку*.

# Статические поля класса

Статические поля класса — это глобальные переменные, определённые внутри класса.

Объявление:

```
struct User {
    ...
private:
    static size_t instances_;
};
```

Определение:

```
size_t User::instances_ = 0;
```

Для доступа к статическим полям не нужен объект.

# Статические методы

Статические методы — это функции, определённые внутри класса и имеющие доступ к закрытым полям и методам.

Объявление:

```
struct User {
    ...
    static size_t count() { return instances_; }
private:
    static size_t instances_;
};
```

Для вызова статических методов не нужен объект.

```
    cout << User::count();
```

# Ключевое слово `inline`

Советует компилятору встроить данную функцию.

```
inline double square(double x) { return x * x; }
```

- В месте вызова `inline`-функции должно быть известно её определение.
- `inline` функции можно определять в заголовочных файлах.
- Все функции, определённые внутри класса, являются `inline`.
- При линковке из всех версий `inline`-функции (т.е. её код из разных единиц трансляции) выбирается только одна.
- Все определения одной и той же `inline`-функции должны быть идентичными.
- `inline` — это совет компилятору, а не указ.

# Правило одного определения

Правило одного определения
(One Definition Rule, ODR)

- В пределах любой единицы трансляции сущности не могут иметь более одного определения.
- В пределах программы глобальные переменные и не-`inline` функции не могут иметь больше одного определения.
- Классы и `inline` функции могут определяться в более чем одной единице трансляции, но определения обязаны совпадать.

Вопрос: к каким проблемам могут привести разные определения одного класса в разных частях программы?

# Класс Singleton

```
struct Singleton {
    static Singleton & instance() {
            static Singleton s;
            return s;
    }

    Data & data() { return data_; }

private:
    Singleton() {}

    Singleton(Singleton const&);
    Singleton& operator=(Singleton const&);

    Data data_;
};
```

# Использование Singleton-а

```
int main()
{
    // первое обращение
    Singleton & s = Singleton::instance();
    Data d = s.data();

    // аналогично d = s.data();
    d = Singleton::instance().data();
    return 0;
}
```