Лекция 7.

Классы

Пусть есть структура `Rect`:

```
struct BadRect {
    int width = 0;
    int height = 0;
    int area = 0;
};

int GetRectAreaSlow(const BadRect& rect) {
    // ...
}
```

Структуры – это всего лишь набор переменных, которые никак не связаны между собой, независимы, но если мы захотим как-то связать их между собой, то структуры не будут так удобны.
Например, мы захотим поменять ширину прямоугольника из `rect`; мы можем поменять отдельно поле с шириной, но потом нужно будет пересчитать площадь — то есть нам нужно после замены размера запускать еще одну функцию после этого. Это неудобно и про это легко забыть.
Для таких более сложных структур, где нужно поддерживать инвариант значений, существуют классы.

```
class Rect {
public:
    int SetWidth(int new_width) {
        width_ = new_width;
        UpdateArea();
    }

    int Area() const {  // const не позволяет менять приватные переменные
        return area_;
    }

private:
    void UpdateArea() {
        area_ = widht_ * height_;
    }

private:
    int width_ = 0;
    int height_ = 0;
    int area_ = 0;
};
```

Классы практически не отличаются от структур; единственное отличие — люди договорились, что структура считается набором полей, а классы еще имеют связи между полями, то есть в ней еще могут быть функции, поддердивающие эти связи.

Все функции, написанные внутри класса, доступны только внутри класса и только для объекта, для которого вызываются.

Модификаторы доступа.

- `private`: поля и функции можно менять/читать/вызывать только изнутри класса. Другими словами, к ним нет доступа снаружи; доступ к ним имеют только методы самого класса.
- `public`: можно менять снаружи; так же методы в этом разделе можно вызывать снаружи.
- `protected`: *пока не было*

Для функций есть свой модификатор досутпа: модификатор `const`. В константных функциях нельзя менять значения переменных.

```
int foo() const {

}
```

Но иногда нам может потребоваться менять некоторые переменные внутри константных методов. Тогда можно объявить переменную как `mutable int area` — и ее можно будет менять даже в `const` функции. Пример:

```
// код про кэш; пример для использования mutable переменных. Базы данных и кэш.
class RectDataBase {
public:
    using RectId = uint64_t;

    Rect GetRect(RectId) {
        // ...
    }
}
// Мораль: есть кэш, то его надо менять, но сам метод константный, потому что ничто другое
// там не меняется. Тогда константность можно нарушить для одного значения.
// Кэш объявляется mutable
```

Удобная штука: `using RectId = uint64_t;`. Таким образом мы можем обращаться к `RectId` как к типу данных, равному по свойствам `uint64_t`. Это упрощает читаемость кода.

Также можно создавать функции, которые имеют возможность читать приватные переменные класса. Они прописываются в поле `private`, со всей сигнатурой и ключевым словом `friend`.

```
class Rect {
public:
    // ...

private:
    int width_ = 0;
    int height_ = 0;
    int area_ = 0;

    friend int RectAreaUnsafe(const& Rect& rect)
};

int RectAreaUnsafe(const& Rect& rect) {
  return rect.area_;
}
```

Это иногда пользуется для перегрузки операторов, если они определяются вне класса.

Инициализация класса

При создании объекта мы часто сразу записываем новые значения, для них хорошо бы иметь и предподсчет, чтобы приписать значения приватным переменным. То есть нужна некоторая функция. Такая функция называется конструктор.

Конструктор вызывается автоматически, сразу при создании объекта; в конструктор надо сразу передать все важные значения. Код просто не скомпилируется, пока в конструктор не передадут все требуемые конструктором переменные.

```
class Rect {
public:
    Rect(int width, int height) {  // это конструктор
        width_ = width;
        height_ = height;
        UpdateArea();
    }

private:
    void UpdateArea() {
        area_ = widht_ * height_;
    }

private:
  int width_ = 0;
  int height_ = 0;
  int area_ = 0;
}
```

Можно еще немного по-другому сразу записать в поля значения:

```
Rect(int width, int height)  // другой вид конструктора
  : width_{width}
  , height_{height}
{}
```

Можно пользоваться также и перегрузкой функций, писать несколько конструкторов, и тогда комиплятор сам будет выбирать нужный конструктор. (Как перегрузка функций: ищет, какой конструктор подходит под набор аргументов)

```
Rect(int width, int height) {  // конструктор
    width_ = width;
    height_ = height;
    UpdateArea();
}

Rect() {  // например конструктор по умолчанию: если ему ничего не передали
    widht_ = 0;
    height_ = 0;
}

Rect()  // этот конструктор вызывет первый конструктор и передаст ему (0, 0)
    : Rect(0, 0)
{}  //     и результат будет такой же, как и у второго.
```

Но для вызова конструктора без аргументов нужно писать без скобок: `Rect r1;`, а не `Rect r1();`

Есть одна нечевидная штука. Если есть функция, принимает в себя объект класса, то вызов функции может выглядеть неочевидно:

```
void foo(Rect r){
    ...
}

int main() {
    foo(123);
}
```

То есть вроде бы функция принимает в себя какое-то целое число, но на деле она вызывет конструктор от одного аргумента. Это очень сложно читается, и такое стоит избегать. Для этого в конструкторах, которые можно вызывать с одним аргументом, нужно указывать ключевое слово `explicit`, и тогда мы будем обязаны явно создавать объект.

```
class Rect {
    // ...
    explicit Rect(int width) {
        // ...
    }
    // ...
}

int main() {
    foo(Rect{123});
}
```

Деструкторы

Как понятно из названия, метод, который вызывается в конце жизни объекта.

```
class Noisy {
public:
    Noisy(int idx) {
        // что выполнится при создании объекта
    }
    ~Noisy() {
        // что выполнится в конце существования объекта
    }
}
```

Объект удаляется, как и переменные, в конце блока с `{...}`.

Если было создано много объектов, то удаляются они в порядке стэка (*First In; Last Out*). И тогда объектом, который был создан позже, могут пользоваться объекты, которые были созданы ранее.