Lec 14

Штуки которые поменялись в C++

Глобальные и статические переменные

Глобальные

- Глобальные переменные - находятся в секции данных
- Глобальные переменные - ВСЕГДА инициализированы, если мы сами не инициализировали, то они инициализируются 0
- Глобальные переменные в C - константы времени компиляции (нельзя написать например `x = f(2)` в глобальной переменной || `static int y = x` тоже нельзя).
  В C++ можно x = f(2)
- Стандарт НЕ говорит в каком порядке будут инициализироваться глобальные переменные (не стоит их много разводить)
- В рамках одного файла переменные инициализируются в порядке определения

```
int x = 0;

f(int x)
{
    static int y = 2;
    return x + y;
}

int main()
{

}
```

Статические (некоторые способы вернуться к правилам C)

var = initializer

- constinit: initializer - *константа по времени компиляции*.
- constexpr: var и initializer - *константы времени компиляции* считается во время компиляции. Внутри constexpr можно юзать только constexpr. constexpr находится на СТЭКЕ;
- consteval: гарантирует, что после компиляции var не будет использоваться;
- const constexpr: не имеет смысла;
- static constexpr: ограничивает оптимизатор;
  ; <комментарий>

Критерий	`constexpr`	`constinit`
Обязательная инициализация	✅ При **компиляции**	✅ При **компиляции**
Изменяемость значения	❌ Нельзя изменить	✅ Можно менять
Может быть функцией/методом	✅ Да	❌ Нет, только для переменных
Может участвовать в вычислениях во время компиляции	✅ Да (значения можно использовать в `constexpr` контексте)	❌ Нет
Устраняет порядок инициализации (`static initialization fiasco`)	❌ Нет	✅ Да
Обязательна константность	✅ Да (`const`)	❌ Нет (`mutable`)

```
int x = 0;

1.
    f(int x)
    {
        static int y = x; // В момент старта проги место под y выдел
        // Это дорого, здесь на самом деле будет на уровне компилято
        return x + y;
    }

2.
    f(int x)
    {
        constinit static int y = x; // Не скомпилится т.к. y - стала
        return x + y;
    }
int main()
{

}
```

```
3. Хороший пример инициализации constexpr через функцию
    constexpr int f(int x) // функция f инициализируется во время ко
    int main()
        {
            return x + 1;
        }
    constexpr int z = f(3); // сработает, идеально
    {

    }
4. Плохой пример 3 варианта
    int f(int x)
    int main()
        {
            return x + 1;
        }
    constexpr int z = f(3); // сработает, НО f не инициализируется в
    {

    }
```

SKKV время жизни временных объектов

декларация:

```
    s f();
```

вызов:

```
    const s &a = f();
```

```
    s &&a = f();
```

в обоих случаях a - это ссылка на умирающий объект. Время жизни объекта, на который указывает "a", продлится до тех пор пока мы не перестанем хотеть его использовать.

❤️‍🩹Пришла Виктория!!!!!!!!!!!!!!

Да да . . . оно действительно как в питоне

```
int q[2] = {1, 2};

auto [a, b] = q; // копирование q -> a,b
```

```
auto &[a, b] = q; // взяие ссылок на q

a = 10; // q[0] = 10
```

best практика (все еще как в питоне):

```
auto f() {
    return {2, 4.5};
}

auto [a, b] = f()
```

range based for (по массивам и структурам / классам и т.д.)

функционал:

1. нельзя получить индекс элемента
2. нельзя изменить исходное содержимое
3. тип значения в цикле будет одинаковым для всех элементов

```
int q[10] = {1, 4, 3}; // {1, 4, 3, '\0', '\0', ...}

for (auto x : q) {
    x // 1, 4, 3
}
```

под капотом:

1. создается указатель
2. создается итератор - как указатель в массиве, но может указывать на элементы, которые не лежат последовательно в памяти
3. итерируемся по элементам
4. auto один единственный раз приведется к типу "t" и затем каждый объект будет этим "t"
5. различия для массивов и структур:
    1. массивы - более простая логика указателей
    2. структуры - более сложная логика указателей

best practise - "распаковать" структуру

```
struct {
    int x = 2;
    double y = 4.5;
} c;

auto [a, b] = c;
```