Lec 12

Исключения

- Мы должны сами бросать исключения с помощью throw, если сработает исключение плюсовое (например мы поделим на 0), то обычно не хорошо, лучше неявно проверять
- При помощи throw, если мы находимся внутри функции, то программа выйдет из этой функции и будет идти до ближайшего catch (либо же крашнется, если его нет)
- try в 32-битной ==версии - ==замедляет работу программы, в 64-битной версии - всё норм
- оптимизатор будет более пессимистичен и будет хуже оптимизировать проги т.к. он не уверен что всё будет работать так, как должно из-за существования throw
- Исключение - компиляторозависимая штука
- Лучше всего если try и catch разбросаны внутри одного треда т.к. если инфа находится в другом треде, то деструктор для другого треда может не сработать

```
try
{
    try
    {
        if (error)
            throw 5;
    }
    catch (int e)
    {

    }
    catch (int *e) //Можно делать несколько кетчей, он будет выбран.
    {

    }
} // Если ни одиг из кетчей внутри не сработал - будут обрабатыватьс
catch
{
```

```
}
// Если и тут кетч не сработал, то будет сработано плюсовое исключен
```

Память не выделилась

- Если попытаемся выделить память но её не будет хватать, то вернётся не NULL, а просто прога крашнется (в отличие от C)

C

```
S *p = malloc(sizeof(S))
if (!p)
{
    printf("error");
}
```

C++

```
try
{
    S *p = new S;
}
catch()
{
    printf("error");
    throw; // Тут мы можем внутри функции (допустим createS) сделать
}
```

C++

> **"Я торжественно обещаю не бросать исключение.**
> **Если же я брошу исключение — убейте мою программу."**
> *— noexcept*

```
struct S
{
    int f() noexcept
    {
    }
}
```

Исключение - компиляторозависимая штука

exe dll dll
main() -> a() -> b()

```
main()
{
    try
    {
    }
    catch
}
```

Всё может крашнуться, если .dll скомпилены другим компилятором

Виды исключений

1.

```
uint div(uint a, uint b)
{
    if (b == 0)
    {
        throw "division by zero" // Будет просто указатель на const
    }
    return a/b;
}
```

2. Возврат кода ошибки

```
typedef int err_t; // Типы ошибок
err_t fiv(uint a, uint b, uint *res)
{
    if (b == 0)
        return ERR_DIV
    *res = a/b;
    return ERR_OK;
}
```

3. Возврат значения

```
int div (int a, int b, err_t *err)
{
    if (b == 0)
    {
        if (err)
            *err = ERR_DIV;
        return 0;
    }
    else
    {
        if (err)
            *err = ERR_OK;
        return a/b;
    }
}
```

B C

2*. Цивилизованно

```
uint x, y, z;
err_t e;
z = div(a, b, &e);
if (e != ERR_OK)
```

2*. Варварски

```
z = div(x, y, NULL);
```

3*. Цивилизованно

```
uint x, y, e;
e = div(x, y, &z);
if (e != ERR_OK)
```

3*. Варварски

```
div(x, y, &z);
```

В C++

```
try
{

}
catch(const char *e)
{

}
```

Сложные типы в кетче

```
catch (E e) // Плохо, создастся копия объекта
{

}

catch (E &e) // Хорошо, используем указатель
{
```

```
}
```

RAII - парадигма использования исключений.

выделение памяти - привязано к созданию объектов

```
// Так себе
Struct S
{
    int *p;
    S (int *z) : p(z) {}
    ~S() {delete[] p;}
}

S x(new int[a]);
S y(new int[b]);

// Супер
Struct S
{
    int *p;
    S (size_t s) : p(new int[s]) {}
    ~S() {delete[] p;}
}

S x(1);
S y(b);
```

Приведение типов внутри структуры

```
struct s
{
    S() {}
    S(int x) {}
}

int main()
```

```
{
    S s = 5;
    s = 2; // Сконструируется новый объект из инта (конверсия типов)
}
```

Запрет на приведение

```
struct s
{
    S() {}
    explicit S(int x) {}
}

int main()
{
    S s = 5;
    s = 2; // Не сработает т.к. S(int x) - explicit
}
```

Неявное преобразование к другому типу

```
struct s
{
    S() {}
    explicit S(int x) {}
    operator int() {return 3;}
}

int main()
{
    S s(5);
    int x = 5;
}
```

Оператор присваивания

S - содержит указатель на данный, но ничего о них не знает, мы сквозь него смотрим на данные. Он не владеет своими данными.

Неправильно

```
struct S
{
    const char *p;
    S() : p(NULL) {};
    S(const char *a) : p(a) {};
    void print() {printf("%s", p)}
}

S f()
{
    char buffer[25] = {}
    S s(buf);
    return s;
}

int main()
{
    S s = f();
    s.printf();
    S x("123"); // Пока всё очев и норм
}
```

Правильно

```
struct S
{
    const char *p;
    S() : p(NULL) {};
    S(const char *a)
    {
        char *b = new char[strlen(a) + 1];
        strcpy(b, a);
        p = b;
    };
    ~S() {delete[] p;}
    void print() {printf("%s", p)}
}

void S f(S &z)
{
    char buffer[25] = {}
```

```
    S s(buf);
    return s;
}

// ПЛОХО
int main()
{
    S s = f(); // Объект, который вернётся - заменяется новым объект
    s = f();
}

int main()
{
    S x("123"); // Объект, который вернётся - заменяется новым объек
    S y("abc");
    x = y; // x.p теряется => ресурсы у текли + деструктор у освобод
}
```

Переопределения "="

```
S &operator= (const S &a)
{
    delete[] p;
    char b = new char[strlen(a.p) + 1];
    strcpy(b, a.p);
    p = b;
}

// Конструктор копирования
S(const S &a)
{
    char *b = new char[strlen(a.p) + 1];
    strcpy(b, a.p);
    p = b;
}
```

Я хочу убрать метод:

```
S(const S &a) = delete;
```

Правило 3-х мужиков

Если сделал 1 из 3-х:

1. Оператор присваивания
2. Деструктор
3. Конструктор копирования
   То ты хочешь сделать оставшиеся два (т.к. без скорее всего всё в моменте может сломаться)

КАСТЫ

Cast тип	Назначение	Что можно преобразовывать	Безопасность	Пример испо
`static_cast`	Стандартные преобразования типов	`int` → `double`, указатели в иерархии классов (вверх/вниз)	✅ Безопасен при правильном использовании	`double d = static_cast<`
`const_cast`	Удаление/ добавление `const` или `volatile`	`const T*` → `T*`, `volatile T*` → `T*`	⚠️ Потенциально опасен	`int* p = const_cast<i`
`reinterpret_cast`	Побитовая интерпретация, низкоуровневое преобразование	`int ⇔ указатель`, указатели разных типов	❌ Очень опасен — используйте с осторожностью	`void* vp = reinterpret_ (ptr);`
`dynamic_cast`	Приведение по иерархии классов с проверкой типа во время выполнения	Только указатели и ссылки на классы с виртуальными методами	✅ Безопасен, но медленнее	`Base* b = dynamic_cast (derivedPtr)`

Нельзя использовать каст в C++:

Каст по иерархии классов:
static_cast если можем доказать тип

C

```
(тип) object;
```

C++

```
const_cast<тип>(value);
char *p1;
const char *p2;
p2 = p1; //  норм
p1 = p2;  // ОШИБКА

В C:
p1 = (char*)p2;

В C++
p1 = const_cast<char*>(p2);
```

RTTI

Run-Time Type Information — это механизм **определения информации о типах объектов во время выполнения программы**, а не на этапе компиляции. **RTTI — это дополнительная информация, хранящаяся рядом с каждым объектом**, если он относится к классу с виртуальными функциями (в C++). Эта информация позволяет узнать тип объекта **во время выполнения**, а не на этапе компиляции. Очень дорого.