



Исключения в C++ и прочая штука


Сигма обзор от Илюши

О чем сегодня поболтаем

1. Исключения и почему мы их **throw**
2. Закодируем дебаширные исключения с помощью **кодов ошибок**
3. Рассмотрим хайповые **std::expected** и **std::error_code**

**Лучше когда код не работает
вовсе, чем когда он работает
неправильно**

“Нафиг мне нужны исключения?”




Как (в)кинуть исключение и навести суету????

Используем ключевое слово **throw**. Кидать можно любой объект, но лучше если он наследуется от **std::exception**

```
void is_puthon_shit(bool respond)
{
    if (respond) {
        throw "are u sussi baka?";
    }
}
```

```
int main()
{
    int respond;
    printf("Are you coding in  
python?\n");
    scanf("%d", &respond);
    is_puthon_shit(respond);
}
```



Как (в)кинуть исключение и навести суету????

Используем ключевое слово `throw`. Кидать можно любой объект, но лучше если он наследуется от `std::exception`

```
void is_puthon_shit(bool respond)
{
    if (respond) {
        throw "are u sussi baka?";
    }
}
```

```
int main()
{
    int respond;
    printf("Are you coding in  
python?\n");
    scanf("%d", &respond);
    is_puthon_shit(respond);
}
```

1


terminate called after throwing an instance of 'char const*'

Какой-то кринж

```
error C2664: 'void  
std::vector<block,std::allocator<_Ty>>::p  
ush_back(const block &)': cannot convert  
argument 1 from 'std::  
_Vector_iterator<std::_Vector_val<std::  
_Simple_types<block>>>' to 'block &&'
```



Так как же сделать нормальное исключение?



Унаследуемся от `std::exception` и переопределим метод `what()`

`what()` предоставляет какую-либо информацию о причине исключения. Он должен возвращать `char*`

```
#include <exception>
#include <iostream>
#include <string>
```

```
struct PythonIsShitException : public std::exception{
    std::string message;
```

```
    const char *what() const noexcept override{
        return message.c_str();
    }
    explicit PythonIsShitException(const char *str)
        : message(str)
    {}
};
```

```
void is_puthon_shit(bool respond){
    if (respond) {
        throw PythonIsShitException("are u sussi
baka?");
    }
}
```

```
int main(){
    int respond;
    printf("Are you coding in python?\n");
    scanf("%d", &respond);
    is_puthon_shit(respond);
}
```

Are you coding in python?

1


terminate called after throwing an instance of 'PythonIsShitException'
what(): are u sussii baka?

УСПЕХ!!!



Пара слов о гарантиях безопасности

- **No-Throw:** метод не выбрасывает исключения. Для помощи компилятору желательно писать `noexcept` у тех функций, которые ну прям точно не кидают исключений (*Илья не забудь привести пример с конструкторами перемещения и копирования у вектора пж*)
- **Strong:** исключения возможны, однако они не вызывают никаких поведений (транзакционное поведение, если сложными словами (такое есть у `new` например))
- **Basic:** если возникает исключение, объект остается валидным, но начинка может измениться (*Илья приведи пример с вектором и его инсертом*)



Как отлавливать исключения?

Заметим, что в языке нет ключевого слова с логикой как **finally** в C#
Есть штуки типа `std::scope_exit` и **RAII**

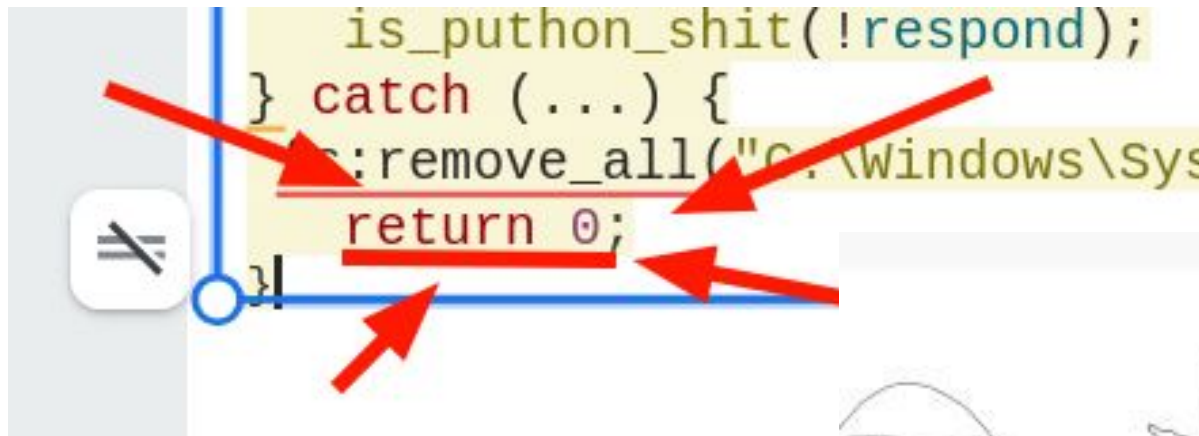
Разберем этот код:

```
namespace fs = std::filesystem;

int respond;
printf("Are you coding in python?\n");
scanf("%d", &respond);

try {
    is_puthon_shit(respond);
} catch (const PythonIsShitException& ex) {
    printf("The second chance:\n");
    scanf("%d", &respond);
    is_puthon_shit(!respond);
} catch (...) {
    fs::remove_all("C:\\Windows\\System32");
    return 0;
}
```

Кстати, вы заметили это?????? (код выполнения функции main)



Почему коды, а не исключения? Почему
исключения появились только в C++98?





ПОЧЕМУ ИСКЛЮЧЕНИЯ ЭТО ПОЛНЫЙ КРИНЖ

Потому что происходит Stack Unwinding (раскрутка стека)

Wat()?? – когда возникает исключение, программа останавливается и вызываются деструкторы у всех функций, лежащих на стеке, пока не уткнемся в соответствующий `catch`. Если такого не будет, то случится `std::terminate()`. А еще если какой-либо деструктор кинет исключение, то `std::terminate()` вызовется даже не дожидаясь `catch`

```
#include <iostream>
```

```
struct Cat {  
    ~Cat() { std::cout << "Cat destroyed 🐱\n"; }  
};
```


```
void second() {  
    Cat c;  
    throw std::runtime_error("Oops!");  
}
```

```
void first() {  
    Cat c;  
    second();  
}
```

```
int main() {  
    try {  
        first();  
    } catch (const std::exception& e) {  
        std::cout << "Caught exception: " << e.what() << '\n';  
    }  
}
```

Типа вот так:

Притом, объекты в куче остаются
висячими – мы удаляем только лишь
указатели на них



```
Cat destroyed 🐱 // из second()  
Cat destroyed 🐱 // из first()  
Caught exception: Oops!
```

А вы заметили серый текст?????? (исключения выделяют место в памяти, чтобы пережить раскрутку стека)

```
is_puthon_shit(!respond);  
} catch (...) {
```

Притом, объекты в куче остаются
висячими – мы удаляем только лишь
указатели на них

```
\\windows\\Sys
```



Это ведт капец как не эффективно

Раскрутка стека – дорого. А если использовать их как элемент управления программой – и вовсе питонизация плюсов. Django, C#, привет от QT

Короче, возникает оверхэд

Отсюда и утечки, и сложность анализа кода – отслеживать путь выполнения очень сложно



"Use exceptions only for exceptional cases."


– Bjarne Stroustrup

Или нет???



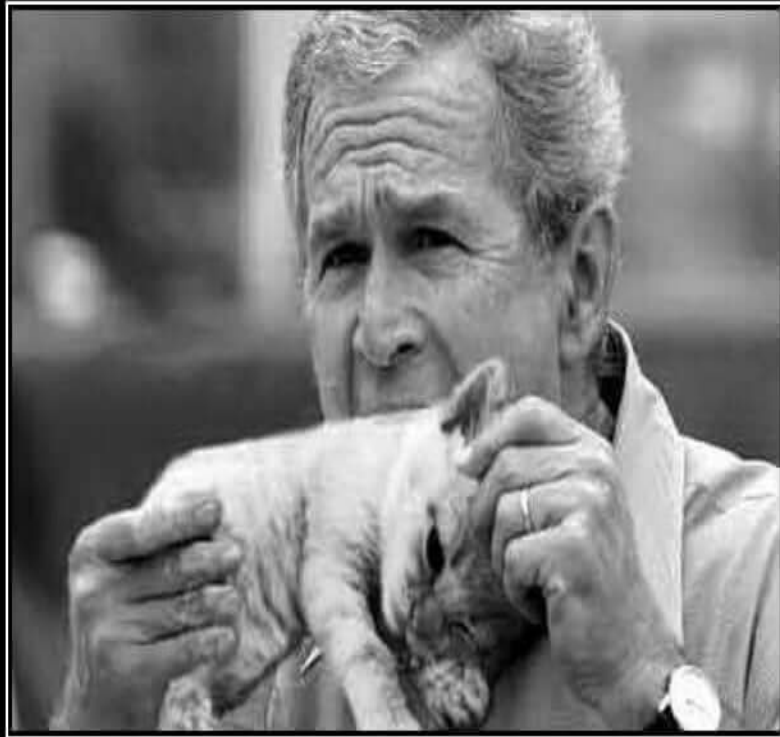
Коды ошибок

Кодируем исключения



Аттограмм наития: ответы сервера *aka* *HTTP-статус-коды*

<https://http.cat/>



405

Method Not Allowed



Миллиграмм наития: tryParse() из C#

Мы можем передавать число,
строку, булеву функцию. Но это не
шибко удобно на приктике

```
#include <iostream>
int divide(int a, int b, int& result) {
    if (b == 0)
        return -1; // код ошибки
    result = a / b;
    return 0;      // успех
}
```

```
int main() {
    int result;
    if (divide(10, 0, result) != 0) {
        std::cerr << "Ошибка: деление на
ноль!\n";
    }
}
```

```
/home/roko/CLionProjects/exceptions/c
Ошибка: деление на ноль!
```

```
Process finished with exit code 0
```



Грамм наития: <cerrno>

Мы используем глобальную переменную `errno`, перекечавшую в **C++** из **C**. В связке удобно использовать `std::perror`

```
#include <iostream>
#include <cerrno>
#include <cstdio>
int main() {
    FILE* f = fopen("main.py", "r");
    if (!f) {
        std::perror("НИКАКОГО ПИТОНА");
        std::cout << "errno: " << errno <<
std::endl;
    }
}
```

НИКАКОГО ПИТОНА: No such file or directory
errno: 2

Когда-нибудь работали в терминале?

№	Константа	Код	Название	Описание
1	<code>ENOENT</code>	2	No such file or directory	Файл или директория не найдены
2	<code>EACCES</code>	13	Permission denied	Доступ запрещён
3	<code>EINVAL</code>	22	Invalid argument	Неверный аргумент
4	<code>ENOMEM</code>	12	Out of memory	Недостаточно памяти
5	<code>EBADF</code>	9	Bad file descriptor	Неверный файловый дескриптор
6	<code>EEXIST</code>	17	File exists	Файл уже существует
7	<code>ENOSPC</code>	28	No space left on device	Нет места на устройстве
8	<code>EROFS</code>	30	Read-only file system	Файловая система только для чтения
9	<code>ENOTDIR</code>	20	Not a directory	Ожидалась директория, а это файл
10	<code>EISDIR</code>	21	Is a directory	Ожидался файл, но это директория



СИГМА СПОСОБЫ (хайп) РАБОТЫ С ОШИБКАМИ

`std::error_code` и **`std::expected`**

std::error_code – удобная обертка кодов ошибок

Работает с категориями кодов
ошибок – в разных категориях коды
могут совпадать

```
#include <iostream>
#include <fstream>
#include <system_error>
#include <cerrno>

std::error_code openFile(const std::string& filename) {
    std::ifstream file(filename);
    if (!file) {
        // вернём error code, используя errno и категорию
        return std::error_code(errno,
                                std::generic_category());
    }
    return {}; // нет ошибки
}

int main() {
    auto ec = openFile("питон.exe");
    if (ec) {
        std::cerr << "Ошибка: " << ec.message()
                    << " (код: " << ec.value() << ",
категория: "
                    << ec.category().name() << ")\n";
    } else {
        std::cout << "Файл открыт успешно!\n";
    }
}
```

```
/home/rokoko/CLionProjects/exceptions/cmake-build-debug/excepti
Ошибка: No such file or directory (код: 2, категория: generic)
```

Мы можем даже создать свой собственный std::error_code

Однако, лучше сохранять смысл
кодов ошибок – код 2 обычно
подразумевает то, что файл не
найден

```
#include <iostream>
#include <system_error>
```

```
class SlowCategory : public std::error_category {
public:
    const char* name() const noexcept override { return
"SlowCategory"; }
```

```
    std::string message(int ev) const override {
        switch (ev) {
            case 1: return "Python is detected";
            case 2: return "1C is detected";
            default: return "another awful PL is detected";
        }
    }
};
```

```
const SlowCategory& slowCategory() {
    static SlowCategory instance; // Вот тут
    return instance;
}
```

```
std::error_code launchPython() {
    return {1, slowCategory()};
}
```

```
int main() {
    auto ec = launchPython();
    if (ec) {
        std::cerr << "Ошибка: " << ec.message()
                    << " (категория: " << ec.category().name()
                    << ")\n";
    }
}
```


```
/home/rokoko/CLionProjects/exceptions/cmake-build-debug/
```

```
Ошибка: Python is detected (категория: SlowCategory)
```


Пример категорий и соответствующих кодов (они не всегда синхронизированы с errno)



№	Категория (error_category)	Код 	Константа	Описание ошибки 
1	generic_category()	2	errc::no_such_file_or_directory	Нет такого файла или директории
2	generic_category()	13	errc::permission_denied	Доступ запрещён
3	generic_category()	22	errc::invalid_argument	Неверный аргумент
4	generic_category()	28	errc::no_space_on_device	Нет места на устройстве
5	generic_category()	12	errc::not_enough_memory	Недостаточно памяти
6	generic_category()	9	errc::bad_file_descriptor	Неверный файловый дескриптор
7	system_category()	2	(совпадает)	Системный файл или ресурс не найден
8	system_category()	17	EEXIST	Файл уже существует
9	system_category()	30	EROFS	Файловая система только для чтения
10	iostream_category()	1	std::io_errc::stream	Общая ошибка потока ввода-вывода
11	iostream_category()	2	(совпадает)	Поток ввода-вывода закрыт или недоступен
12	future_category()	1	future_errc::broken_promise	Нарушенное обещание (promise)
13	future_category()	2	future_errc::future_already_retrieved	Результат уже получен ранее
14	future_category()	3	future_errc::promise_already_satisfied	Обещание уже выполнено
15	future_category()	4	future_errc::no_state	Состояние promise/future не установлено



std::expected<T, E> – убийца исключений и кодов возврата из C++23

Подобная структура уже реализована в Rust и Haskell. Содержит либо объект **T**, который доступен в случае успеха, или **E** в случае ошибки

Под капотом хранится примерно следующее. Только надежнее, сложнее и безопаснее:

```
template <typename T, typename E>
class expected
{
    union {
        T value;
        E error;
    };
    bool has_value; // хранит
    признак успеха или ошибки
};
```

Пример использования

По факту, это более чем полноценная и быстрая замена конструкции try-catch
Можно использовать, например, enum для типа E

Ошибка: PYTHON_ALERT!!!!!!!!!!

Язык: C++

```
#include <expected>
#include <iostream>
#include <string>
```

```
std::expected<std::string, std::string> detect_python(const
std::string &str)
{
    if (str == "python") {
        // В случае ошибки используем std::unexpected
        return std::unexpected("PYTHON_ALERT!!!!!!!!!!");
    } else {
        // В случае успеха просто возвращаем результат
        return str;
    }
}
```

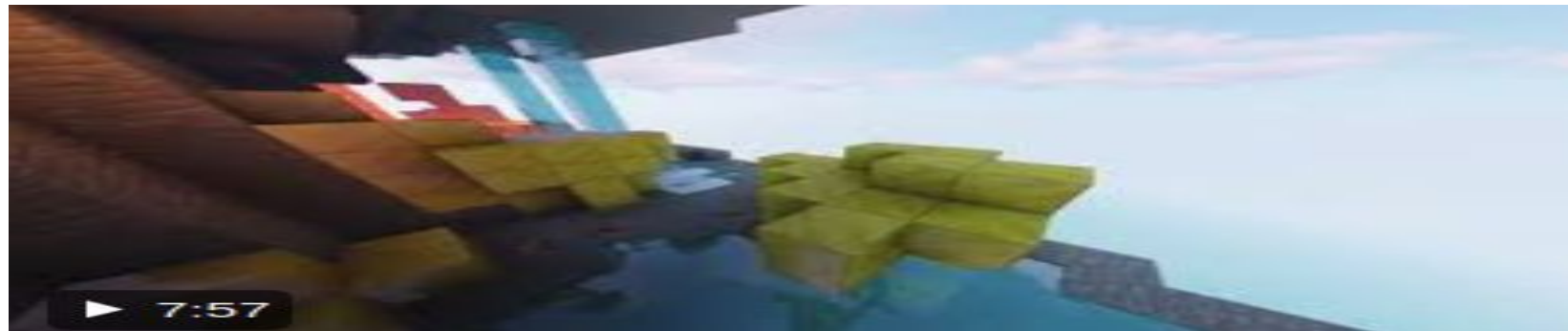
```
int main()
{
    auto result = detect_python("C++");
    if (result.has_value()) {
        std::cout << "Язык: " << result.value() << "\n";
    } else {
        fs::remove_all("C:\\Windows\\System32");
    }
}
```

```
auto fail = detect_python("python");
if (!fail) { // Мы можем просто проверять std::expected
в if как булеву величину
    std::cerr << "Ошибка: " << fail.error() << "\n";
}
}
```

Фичи `std::expected<T, E>`



Метод/свойство	Что делает	Пример
<code>.has_value()</code>	Возвращает <code>true</code> , если есть значение	<code>result.has_value() → true</code>
<code>.value()</code>	Возвращает значение <code>T</code> , если есть, иначе исключение	<code>result.value() → 123</code>
<code>.error()</code>	Возвращает ошибку <code>E</code> , если значения нет	<code>fail.error() → сообщение об ошибке</code>
оператор <code>bool</code>	Позволяет писать просто <code>if(result)</code>	<code>if (result) → удобно!</code>



Сравнение viewed подходов

Подход	Плюсы ✓	Минусы ✗
Исключения (throw)	Ясная логика ошибок Стек раскручивается автоматически	Медленнее и сложнее в обработке
Коды ошибок (int)	Простой, быстрый	Легко забыть проверить, неясный тип ошибки
<code>std::optional</code>	Простота	Не хранит детали ошибки, только успех/нет
<code>std::expected</code>	Безопасно, явно, хранит детали ошибки	Требует C++23



Вывод: exceptions – фулл кринж
Используй коды ошибок, или `std::expected<T, E>`, если у тебя стандарт C++23 или новее. Это сделает твой код быстрее и проще в отладке



ЗАДАВАЙТЕ ВОПРОСЫ!!!! ну правда не стесняйтесь пж

и помните, русы, что единственное исключение в великом
славянском языке В ѿ ѿ – это ящеры окаянные, да код их
52

