

# Язык C++

## Error Handling

# Ошибки

---

1. Выход за границу массива
2. Деление на ноль
3. Невозможность выделить память
4. Отсутствие прав на открытие файла
5. Недоступность внешнего сервера
6. ....

# Assert

---

```
#include <cassert>
```

```
int main() {  
    assert(2+2 == 4);  
    assert(2+2 == 5);  
    return 0;  
}
```

```
int main(): Assertion `2+2 == 5' failed.
```

# static\_assert

---

```
static_assert(sizeof(int) == 4, "int must be 4 bytes");
```

```
template <typename T>
struct data_structure {
    static_assert(
        std::is_default_constructible<T>::value,
        "Data Structure requires default-constructible elements"
    );
};

struct no_default {
    no_default () = delete;
};

int main() {
    data_structure<no_default> ds_error;
    return 0;
}
```

# Код возврата

---

```
// количество успешно записанных
size_t fwrite( const void *buffer, size_t size, size_t count, FILE *stream );

// errno
FILE *fopen( const char *filename, const char *mode );

// ошибка в качестве кода возврата
errno_t fopen_s(FILE *restrict *restrict streamptr, const char *restrict filename, const char *restrict mode);
```

ERRNO

# Обработка в месте возврата

---

```
int main() {  
    FILE* file = fopen("test.tmp", "w");  
    if(!file) {  
        // do something  
    }  
    if(fprintf(file, "Hello") < 0 || printf(file, "World") < 0) {  
        // do something  
    }  
    if(fclose(file) ==EOF) {  
        // do something  
    }  
    return 0;  
}
```

# Exception. throw + try + catch

---

```
int foo() {  
    throw std::runtime_error("error");  
}
```

```
void boo() {  
    throw 2;  
}
```

```
void coo() {  
    throw std::string("Hello world");  
}
```

```
int main(int, char**) {  
    try{  
        foo();  
    }  
    catch(...) {  
  
    }  
}
```

# Stack unwinding

---

1. Сконструированный объект пробрасывается обратно по стэку
2. До встречи подходящего блока try\catch
3. “Раскручивая” стэк обратно уничтожаются все объекты с automatic storage duration  
**(!NB Если исключение не перехватывается, то stack unwinding зависит от реализации )**
4. std::terminate если в процессе возникает еще одно исключение
5. Деструктор поехсепт
6. Сам объект хранится в неопределенном участке памяти



# Stack unwinding

---

```
struct Foo {
    Foo() { std::cout << "Foo()\n"; }
    ~Foo() {
        std::cout << "~Foo()\n";
    }
};

void internalFunc() {
    Foo f;
    throw std::runtime_error("Some error");
}

void externalFunc() {
    try {
        internalFunc();
    }
    catch (std::exception& e) {
        std::cout << e.what() << std::endl;
    }
}
```

# Exception

---

```
int main() {  
    try {  
        foo();  
    } catch (const std::overflow_error& e) {  
        // do somethisg  
    } catch (const std::runtime_error& e) {  
        // do somethisg  
    } catch (const std::exception& e) {  
        // do somethisg  
    } catch (...) {  
        // do somethisg  
    }  
}
```

# Гарантии безопасности исключений

---

- No guarantee
- Basic guarantee
  - Сохраняется инвариант
  - Нет утечек
- Strong guarantee
  - Сохраняется инвариант
  - Нет утечек
  - Состояние возвращается к состоянию до исключения
- Nothrow guarantee
  - Не может быть выкинуто исключение

# Exception guarantee

---

```
struct Foo {  
    int value;  
  
    Foo(int v)  
        : value(v)  
    {}  
  
    Foo(const Foo& other)  
        : value(other.value)  
    {  
        if(something)  
            throw std::runtime_error("KEKW");  
    }  
};
```

Конструктор может кинуть исключение

# Exception guarantee

---

```
class Boo {  
private:  
    Foo* foo_ = nullptr;  
    int value_ = 0;  
  
public:  
    Boo(int value = 0) : value_(value) {}  
  
    Boo(int value, int foo_value)  
        : foo_(new Foo{foo_value})  
        , value_(value)  
    {}  
  
    ~Boo() { delete foo_; }  
  
    friend std::ostream& operator<< (std::ostream& stream, const Boo& value);  
};
```

# No guarantee

---

```
Boo(const Boo& other)
    : value_(other.value_)
    , foo_(new Foo(*other.foo_))
{
}
```

```
Boo& operator=(const Boo& other) {
    value_ = other.value_;
    delete foo_;
    foo_ = new Foo(*other.foo_);
    return *this;
}
```

Возможно ***other.foo\_ == nullptr***

# No guarantee

---

```
Boo& operator=(const Boo& other)
{
    if(this == &other)
        return *this;

    value_ = other.value_;
    delete foo_;
    if(other.foo_)
        foo_ = new Foo(*other.foo_);
    return *this;
}
```

После перехвата исключение в *foo\_* адрес который мы удалили

# Basic guarantee

---

```
Boo& operator=(const Boo& other) {  
    if(this == &other)  
        return *this;  
  
    value_ = other.value_;  
    delete foo_;  
    foo_ = nullptr;  
  
    if(other.foo_)  
        foo_ = new Foo(*other.foo_);  
  
    return *this;  
}
```

Даже если конструктор бросит  
исключение **Boo** останется в  
инвариантном состоянии



# Basic guarantee

---

```
Boo(const Boo& other)
    : value_(other.value_)
{
    if(other.foo_)
        foo_ = std::make_unique<Foo>(*other.foo_);
}
```

```
Boo& operator=(const Boo& other) {
    if(this == &other)
        return *this;

    value_ = other.value_;
    foo_.release()
    if(other.foo_)
        foo_ = std::make_unique<Foo>(*other.foo_);

    return *this;
}
```



**RAII**

# Strong guarantee

---

```
Boo& operator=(const Boo& other) {  
    if(this == &other)  
        return *this;  
  
    Boo tmp(other);  
    swap(tmp);  
  
    return *this;  
}
```

```
void swap(Boo& other) noexcept {  
    std::swap(value_, other.value_);  
    std::swap(foo_, other.foo_);  
}
```

Исключение в  
конструкторе **Boo** не  
поменяет состояние **this**

**swap** не кидает  
исключение

**Copy And Swap Idiom**

# Strong guarantee

---

```
Boo& operator=(const Boo& other)
```

```
{
```

```
    if(this == &other)
```

```
        return *this;
```

```
    Boo tmp(other);
```

```
    *this = std::move(tmp);
```

```
    return *this;
```

```
}
```

```
Boo& operator=(Boo&& ) noexcept = default;
```

# noexcept

---

1. Гарантирует что функция не будет бросать исключения
2. Не сворачивает стэк
3. Позволяет компилятору лучше оптимизировать код
4. `std::terminate`
5. Деструктор `noexcept` по умолчанию

# std::exception

---

1. Кидать стандартные типы в качестве исключений - малоинформативно
2. Исключение должно нести информацию о случившемся событии
3. std::exception - базовый класс для исключений стандартной библиотеки
4. Тип исключения также является полезной информацией

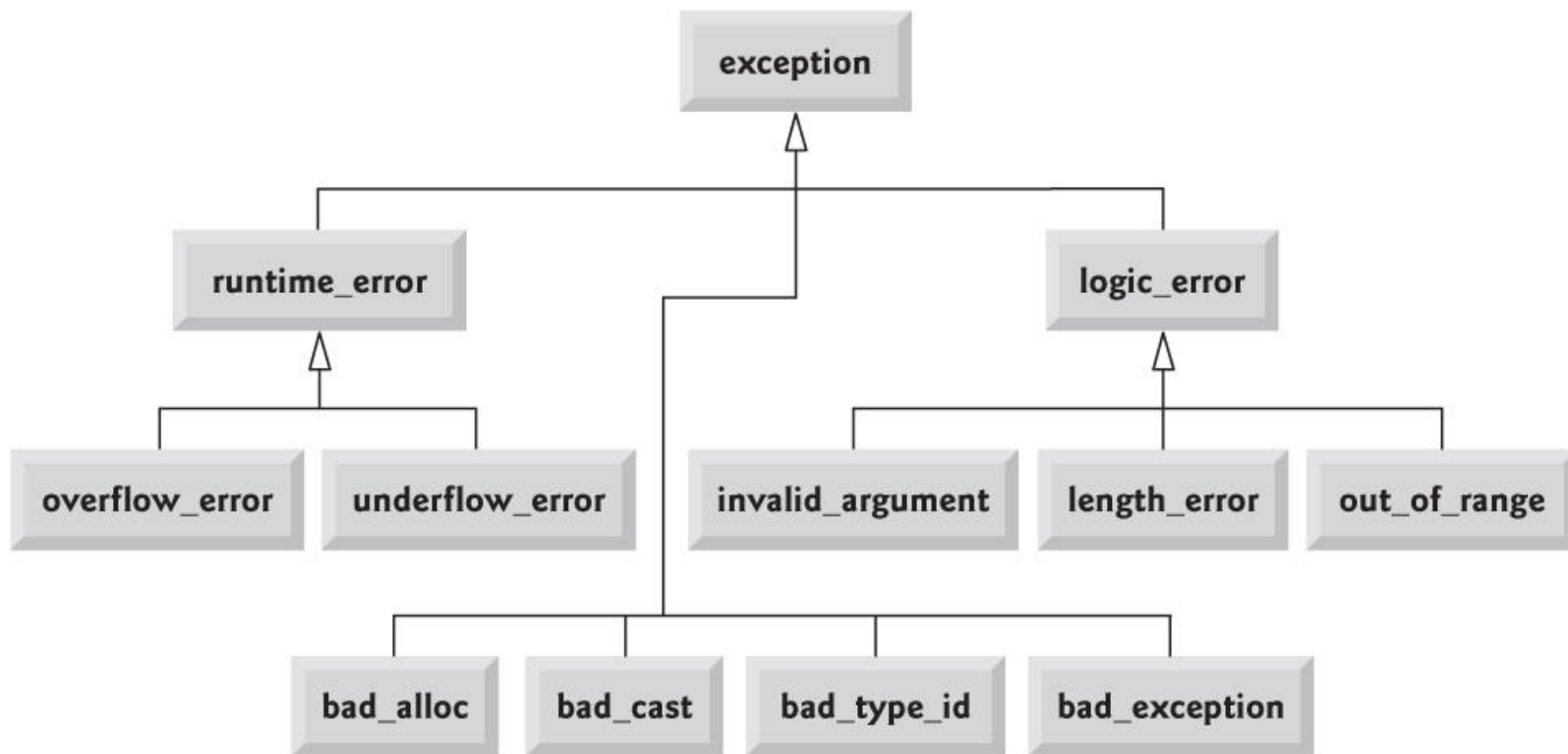
# std::exception

---

```
class exception {  
public:  
    exception() noexcept;  
    exception(const exception&) noexcept;  
    exception& operator=(const exception&) noexcept;  
    virtual ~exception();  
    virtual const char* what() const noexcept;  
};
```

# std::exception

---



# std::exception

---

```
class my_exception : public std::exception { // derived from std::exception
public:
    my_exception(const std::string& what)
        : what_(what) {
    }
    const char* what() const noexcept override {
        return what_.c_str();
    }
private:
    std::string what_;
};
```



# std::exception

---

```
int foo() {  
    throw my_exception("error"); // by rvalue  
}  
  
int main(int, char**) {  
    try{  
        foo();  
    }  
    catch(const my_exception& e) { // by const reference  
        std::cerr << e.what();  
        std::runtime_error  
    }  
}
```

# Exception

---

1. Исключения предназначены исключительно для обработки ошибок
2. Обработки ошибок должна строиться вокруг инварианта объекта
3. Исключения принято кидать по-значению, а ловить по-ссылку

# Exception cost

---

```
struct invalid_value {};  
  
void do_sqrt(std::span<double> values) {  
    for (auto& v : values) {  
        if (v < 0) throw invalid_value{};  
        v = std::sqrt(v);  
    }  
}
```

Threads	1	2	4	8	12
0.0% failure	19ms	19ms	19ms	19ms	19ms
0.1% failure	19ms	19ms	19ms	19ms	20ms
1.0% failure	19ms	19ms	20ms	20ms	23ms
10% failure	23ms	34ms	59ms	168ms	247ms

[Proposal P2544R0](#)

# std::expected

---

```
enum class EDivError {  
    DevisionByZero = 0,  
};  
  
std::expected<int, EDivError> my_div(int a, int b) {  
    if(b == 0)  
        return std::unexpected{EDivError::DevisionByZero};  
  
    return a/b;  
}
```

# std::expected

---

```
int main() {  
    auto r = my_div(8, 0);  
    if(r)  
        std::cout << *r << std::endl;  
  
    try {  
        std::cout << r.value() << std::endl;  
    } catch (std::bad_expected_access<EDivError>& err) {  
        std::cout << err.what() << std::endl;  
    }  
  
    return 0;  
}
```

# std::expected (C++ 23)

---

- Позволяет возвращать либо ожидаемое значение либо ошибку
  - Накладные расходы сравнимы с кодом возврата
  - Передает ответственность за обработку вызывающему коду
- 
- `std::expected<T, E>`
  - `std::unexpected<E>`
  - `std::bad_expected_access`

# Исключения и код возврата

---

1. Исключения позволяют обрабатывать ошибки единообразно, но не в месте возникновения
2. Коды возврата позволяют обработать ошибку сразу при возникновении но не единообразно
3. `std::expected` позволяет иметь комбинированный подход

# Исключения и код возврата

---

```
uint32_t to_uint(std::string_view str) {  
    uint32_t result = 0;  
    for(char c : str) {  
        result *= 10;  
        result += c - '0';  
    }  
  
    return result;  
}  
  
int main(int, char**){  
    std::cout << to_uint("100500") << std::endl;  
}
```



# Исключения и код возврата

---

```
uint32_t to_uint(std::string_view str) {  
    uint32_t result = 0;  
    for(char c : str) {  
        result *= 10;  
        result += c - '0';  
    }  
  
    return result;  
}  
  
int main(int, char**){  
    std::cout << to_uint("abc") << std::endl;  
    std::cout << to_uint("abc100500") << std::endl;  
    std::cout << to_uint("100500abc") << std::endl;  
}
```

Требуется обрабатывать  
исключительные ситуации

# Исключения и код возврата

---

```
uint32_t to_uint(std::string_view str) {  
    uint32_t result = 0;  
  
    if(str.empty())  
        return 0;  
  
    for(char c : str) {  
        if(c < '0' || c > '9')  
            return result;  
        result *= 10;  
        result += c - '0';  
    }  
  
    return result;  
}
```

- Не сообщает нам об ошибке
- Не решает проблему строк начинающихся с цифр
- Не решает проблему пустых строк

# Исключения и код возврата

---

```
bool to_uint(std::string_view str, uint32_t& result) {  
    if(str.empty())  
        return false;  
  
    for(char c : str) {  
        if(c < '0' || c > '9')  
            <stdlib.h>  
            result *= 10;  
            result += c - '0';  
    }  
  
    return true;  
}
```

Лучше, но не отвечает на вопрос  
что случилось

# Исключения и код возврата

---

```
uint32_t to_uint(std::string_view str) {  
    uint32_t result = 0;  
    if(str.empty()) {  
        errno = EINVAL;  
        return result;  
    }  
  
    for(char c : str) {  
        if(c < '0' || c > '9') {  
            errno = EDOM;  
            return result;  
        }  
        result *= 10;  
        result += c - '0';  
    }  
  
    return result;  
}
```

Решает проблему в стиле C  
через [errno](#)

# Исключения и код возврата

---

```
uint32_t to_uint(std::string_view str) {  
    uint32_t result = 0;  
    if(str.empty())  
        throw std::invalid_argument("String is empty");  
  
    for(char c : str) {  
        if(c < '0' || c > '9')  
            throw std::invalid_argument{std::format("Argument {} is not a number", str)};  
  
        result *= 10;  
        result += c - '0';  
    }  
  
    return result;  
}
```

Проброс исключения

# Исключения и код возврата

---

```
std::optional<uint32_t> to_uint(std::string_view str) {  
    if(str.empty())  
        return {};  
    uint32_t result = 0;  
  
    for(char c : str) {  
        if(c < '0' || c > '9')  
            return {};  
  
        result *= 10;  
        result += c - '0';  
    }  
  
    return result;  
}
```

Похож на вариант с bool, но бросит исключение если не проверить наличие результата

# Исключения и код возврата

---

```
std::expected<uint32_t, std::invalid_argument> to_uint(std::string_view str) {  
    if(str.empty())  
        return std::unexpected{std::invalid_argument("String is empty")};  
  
    uint32_t result;  
  
    for(char c : str) {  
        if(c < '0' || c > '9')  
            return std::unexpected{std::invalid_argument{std::format("Argument {} is not a number", str)}};  
  
        result *= 10;  
        result += c - '0';  
    }  
  
    return result;  
}
```

Использует std::expected