C++

Fall 2020

compscicenter.ru

Башарин Егор, t.me/egorbasharin

Лекция XIII

Error handling. Exceptions. Exception safety.

Section 1

Errors

Что является ошибкой?

Что считать ошибкой во время выполнения некоторой функции:

- Нарушение постусловий функции
- Нарушения предусловий других вызываемых функций
- Неудачное восстановление инварианта класса (*для неприватных методов*)

Пример I

```
// pre-condition: h >= 0, w >= 0
// post-condition: area >= 0
int area(int h, int w) {
    return h * w;
}

int use_area() {
    int h = get_h();
    int w = get_w();
    return area(h, w);
}
```

Пример II

```
class GeoPoint {
    void move(double lat_direction, double lon_direction
    {
        lat_ += lat_direction;
        lon_ += lon_direction;
    }

private:
    double lat_; // [-90; 90]
    double lon_; // (-180; 180]
};
```

wide contract

```
#include <vector>

int func() {
    std::vector<int> v{1,2,3};
    int i = v.at(4);
    return i;
}
```

- Есть ли ошибки в функции `func`?
- Нарушаются ли предусловия у метода `at`?

Section 2

Handling errors

Способы обработки ошибок

- использование глобальной переменной
- возврат кода ошибки
- завершение программы
- исключения
- использование `std::optional`

Проверка условий в коде

`static_assert` — проверка во время компиляции

`assert` — проверка во время исполнения (в дебажной сборке)

Useful things

- Логирование
- Debugger
- Crash reporting: stacktraces, memory dump

Section 3

Exceptions

Exceptions

+ Компактный код

+ Механизм распростренения поддержан рантаймом языка

- Non-zero overhead

- Проблемы с ABI

Throwing exceptions

```
throw expression

throw
```

- можно выбросить объект любого типа
- пользовательский класс исключения наследуется от `std::exception` (*idiomatic way*)
- исключения из стандартной библиотеки: <stdexcept>

Stack unwinding

После выброса иссключения происходит уничтожение объектов в обратном порядке до тех пор, пока не достигнуто начало try-блока

```
struct T {
    int i;
    ~T() { std::cout << "~T" << i << "\n"; }
};

void f() {
    T t{1}; throw std::runtime_error("error");
}

void g() {
    try {
        T t{2}; f();
    } catch (...) {
        throw;
    }
}

int main() {
    T t{3};
    g();
}
```

Catching exceptions syntax

try-block

```
try {
    f();
} catch (const std::overflow_error& e) {
    // this executes if f() throws std::overflow_error (
} catch (const std::runtime_error& e) {
    // this executes if f() throws std::underflow_error
} catch (const std::exception& e) {
    // this executes if f() throws std::logic_error (bas
} catch (...) {
    // this executes if f() throws std::string or
    // int or any other unrelated type
}
```

Catching exceptions syntax

function-try-block

```
struct S {
    std::string m;
    S(const std::string& arg) try : m(arg, 100) {
        std::cout << "constructed, mn = " << m << '\n';
    } catch(const std::exception& e) {
        std::cerr << "arg=" << arg
            << " failed: " << e.what() << '\n';
    } // implicit throw; here
};
```

Catching exceptions

- если не перехватить, то выполнение программы закончится
- перехватывать лучше по константной ссылке

Section 4

`noexcept` operator & `noexcept` specifier

`noexcept` specifier

- оптимизации компилятора
- ускорение алгоритмов
- помощь в обеспечении гарантий безопасности исключений

noexcept specifier

```
template <class T>
struct UniquePtr {
    UniquePtr(UniquePtr&& other) noexcept
        : resource_(other.release())
    {}
    T* release() { /* implementation */ }
private:
    T* resource_;
};

void nothrow_func() noexcept { /*body*/ }

int main() {
    auto f = [](int i) noexcept { return i; };
    f(1);
}
```

noexcept specifier

noexcept(const_expression)

```
void f() noexcept(true) { }
void g() noexcept(false) { }
```

noexcept specifier

```
#include <type_traits>
template <class T, class U>
void assign(T& dest, const U& src)
    noexcept(std::is_nothrow_assignable_v<T, U>)
{
    dest = src;
}
```

`noexcept` operator

```
noexcept( expression )
```

Во время компиляции проверяет может ли `expression` выбросить исключение. Возвращает `bool`.

```
void f() noexcept(true) {}
void g() noexcept(false) {}

int main() {
    static_assert(noexcept(f()) == true);
    static_assert(noexcept(g()) == false);
}
```

noexcept operator

```
#include <type_traits>
template <class T, class U>
void assign(T& dest, const U& src)
    noexcept(std::is_nothrow_assignable_v<T, U>)
{
    dest = src;
}

template <class T, class U>
void assign2(T& dest, const U& src)
    noexcept(noexcept(std::declval<T>() = std::declval<U
{
    dest = src;
}
```

non-noexcept & stl

```
size_t cost = 0;

struct T {
    T() = default;
    T(const T&) { cost += 100; }
    T(T&&) { cost += 2; }

    T& operator=(const T&) { cost += 150; return *this; }
    T& operator=(T&&) { cost += 3; return *this; }
};

int main() {
    std::vector<T> ts(32); // 32 items
    assert(ts.size() == 32);

    cost = 0;
    ts.push_back(T());
    std::cout << cost; // bad
}
```

noexcept & stl

```cpp
size_t cost = 0;

struct T {
    T() = default;
    T(const T&) { cost += 100; }
    T(T&&) noexcept { cost += 2; }

    T& operator=(const T&) { cost += 150; return *this; }
    T& operator=(T&&) noexcept { cost += 3; return *this; }
};

int main() {
    std::vector<T> ts(32); // 32 items
    assert(ts.size() == 32);

    cost = 0;
    ts.push_back(T());
    std::cout << cost; // good
}
```

Section 5

Exception-safety

Exception-safety

- Nothrow exception guarantee
- Strong exception guarantee
- Basic exception guarantee
- No exception guarantee

Exception-safety

Nothrow exception guarantee

Функции всегда выполняются успешно (не выбрасывают исключения)

Такая гарантия ожидается от всех функций, вызывающихся при "размотке" стека. Деструкторы в том числе.

Обычно такая гарантия ожидается у move-конструкторов/операторов, swap-функций.

Exception-safety

Strong exception guarantee

Исключение в функции приведет программу в состояние до вызова этой функции.

Выполнение функции можно рассматривать как транзакцию.

Basic exception guarantee

Выброс исключение оставляет программу в валидном состоянии:

- инварианты сохранены
- утечки отсутствуют

STL: Все контейнеры реализуют по крайней мере эту гарантию

Exception-safety

No exception guarantee

Все плохо:

- утечки ресурсов
- нарушены инварианты

Дальшейшее выполнение программы неопределено.

Section 6

live-coding: stack

checklist

- [x] `default` ctor
- [x] `copy-ctor`
- [ ] `move-ctor`
- [x] `dtor`
- [ ] `push`
- [x] `pop`

checklist

[x] Не требует конструктора по умолчанию от T

[x] Общая нешаблонная база

[x] Учтено выравнивание

[_] move_if_noexcept при реаллокации

[x] `top`

Section 7

exception_ptr

Example

```
void handle_eptr(std::exception_ptr eptr) // passing by value is ok
{
    try {
        if (eptr) {
            std::rethrow_exception(eptr);
        }
    } catch(const std::exception& e) {
        std::cout << "Caught exception \"" << e.what() << "\"\n";
    }
}

int main()
{
    std::exception_ptr eptr;
    try {
        std::string().at(1); // this generates an std::out_of_range
    } catch(...) {
        eptr = std::current_exception(); // capture
    }
    handle_eptr(eptr);
} // destructor for std::out_of_range called here, when the eptr is destructed
```