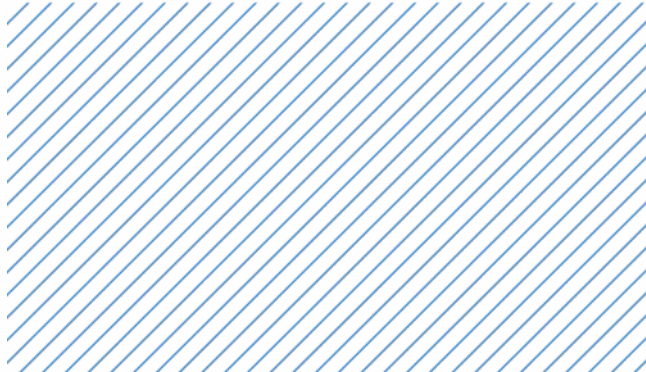
HIGHER SCHOOL OF ECONOMICS
NATIONAL RESEARCH UNIVERSITY

Лекция 5

Исключения

Программирование на языке C++

Константин Леладзе
ФКН ВШЭ



Проблема

До сих пор, обработка некоего исключительного сценария поведения программы, при котором дальнейшее выполнение ее куска должно быть прервано, а контекст возвращен в вызывающий блок, происходило “в ручном режиме”.

Например, представим, что у нас есть метод, который вычисляет площадь многоугольника, заданного набором точек, внутри которого нам необходимо проверить, нет ли повторений в этом наборе и то, не является ли многоугольник самопересекающимся. В противном же случае, следует сообщить пользователю о возникшей ошибке, вызванной некорректностью переданных данных. Для этого, получим от него специальный флаг-переменную по ссылке и изменим ее значение в том случае, если ошибка возникнет.

Посмотрим пример example-1.cpp

Очевидны несколько проблем:

- Флаг необходимо передавать, получать и обрабатывать
- Значения флага абстрактны и никак не связаны с возникшей ошибкой
- Можно пропустить какое-то значение

Исключение

Используя ключевое слово **throw**, мы можем сообщить пользователю о том, что произошло исключение. Концептуально, исключение – не есть ошибка, исключение – это способ передачи сообщений в программе.

Выбрасывание исключения из функции приводит к прерыванию ее выполнения и возвращению контекста к вызывающему блоку.

Если исключение не **обработать в вызывающем блоке**, это приведет к ошибке исполнения программы.

```cpp
#include <iostream>
#include <cmath>

bool are_close (const double first, const double second, const double eps = 1e-6) {
    return std::fabs(first - second) < eps;
}

/// calculates sqrt(x) / (x - 1)
double calculate (const double x) {
    if (are_close(x, 1.))
        throw "denominator turns zero";
    else if (x < 0.)
        throw "value under the square root should not be negative";
    return std::sqrt(x) / (x - 1);
}

int main () {
    double x;
    std::cin >> x;
    std::cout << calculate(x) << std::endl;
    return 0;
}
```

Исключение

Попробуем запустить программу и протестировать ее работу на различных входных данных:

```
13
0.300463

Process finished with exit code 0
```

```
1
libc++abi: terminating with uncaught exception of type char const*

Process finished with exit code 134 (interrupted by signal 6: SIGABRT)
```

```
-4
libc++abi: terminating with uncaught exception of type char const*

Process finished with exit code 134 (interrupted by signal 6: SIGABRT)
```

Необработанные исключения приводят к ошибке исполнения, причем сообщения-исключения не выводятся, вместо этого, выводится лишь тип: **const char*.**

Обработка исключений

Обработать исключение можно в вызывающем блоке, используя конструкцию **try / catch**.

Внутрь блока **try** необходимо поместить код, который может выкинуть исключение. Если внутри него будет выброшено исключение, его исполнение прервется, и контекст перейдет в блок **catch**.

В блоке **catch** же нужно указать тип исключения, которое мы хотим в нем поймать, так как в общем случае, исключение может быть произвольного типа, хотя и на практике рекомендуется использовать исключительно типы, наследующие класс **std::exception,** но об этом позже**.**

Если исключение обработать (поймать) в вызывающем блоке, то ошибки исполнения программы не произойдет. Ошибку вызывает только необработанная ошибка.

```cpp
int main () {
    double x;
    std::cin >> x;

    try {
        std::cout << calculate(x) << std::endl;
    } catch (const char* const str) {
        std::cout << "calculation exception: " << str << std::endl;
    }

    return 0;
}
```

```
-1
calculation exception: value under the square root should not be negative

Process finished with exit code 0
```

```
1
calculation exception: denominator turns zero

Process finished with exit code 0
```

```
13
0.300463

Process finished with exit code 0
```

Обработка нескольких типов исключений

```
int main () {
    try {
        bool throw_int_or_double;
        std::cin >> throw_int_or_double;

        if (throw_int_or_double)
            throw 1; /// int
        else
            throw 3.14; /// double
    } catch (const int x) {
        std::cout << "int caught" << std::endl;
    } catch (const double x) {
        std::cout << "double caught" << std::endl;
    }

    return 0;
}
```

Если из какого-то блока **try** предполагается выбрасывание нескольких типов исключений, то после него можно написать несколько блоков **catch**, внутри каждого из которых принять соответствующий тип исключения и обработать его. При исполнении, интерпретатор сам определит, какой блок **catch** вызвать в зависимости от типа исключения.

```
0
double caught

Process finished with exit code 0
```

```
1
int caught

Process finished with exit code 0
```

Обработка нескольких типов исключений

```
struct granny {};
struct mommy : granny {};
struct son : mommy {};

int main () {
    try {
        throw son{};
    } catch (const granny& g) {
        std::cout << "granny caught" << std::endl;
    } catch (const mommy& m) {
        std::cout << "mommy caught" << std::endl;
    } catch (const son& s) {
        std::cout << "son caught" << std::endl;
    }

    return 0;
}
```

Выбран будет первый подходящий блок **catch**. Исполнен всегда будет только один блок, даже если подходит несколько, например, в случае наследования.

В примере слева, исключение типа **son** обрабатывается блоком **granny**, так как он подошел первым (*преобразование типов от наследника к родителю – единственное допустимое преобразование в **catch***), несмотря на то, что дальше есть блок **son**, который является более подходящим.

```
granny caught

Process finished with exit code 0
```

Обработка нескольких типов исключений

Вывод: послушаемся подсказки IDE, и будем располагать блоки catch в порядке, обратном иерархии наследования:

```
int main () {
    try {
        throw son{};
    } catch (const son& g) {
        std::cout << "son caught" << std::endl;
    } catch (const mommy& m) {
        std::cout << "mommy caught" << std::endl;
    } catch (const granny& s) {
        std::cout << "granny caught" << std::endl;
    }

    return 0;
}
```

→

```
son caught

Process finished with exit code 0
```

Блок **catch-all**

```
void f () {...}

int main () {
    try {
        f();
    } catch (...) {
        std::cout << "caught an unknown exception" << std::endl;
    }

    return 0;
}
```

Существует возможность сделать блок catch, который будет ловить все исключения вне зависимости от типа, однако получить само исключение внутри него не выйдет ввиду неопределенности его типа. Использовать подобный блок в enterprise-решении не стоит, однако он полезен для имплементации мокового функционала, во время дебага, при использовании плохо задокументированной библиотеки, либо же в экзотическом случае, когда исключение необходимо “проглотить” вовсе.

***Замечание**: блок **catch-all** должен быть единственным и следовать строго после всех остальных блоков **catch**.*

Повторное выбрасывание исключений

Если по какой-то причине есть необходимость обработать исключение, и после этого прокинуть его дальше, есть возможность сделать throw из блока try без указания выбрасываемого объекта (он автоматически определится как исключение, полученное в “аргументе” блока try):

```
int main () {
    try {
        try {
            throw 123;
        } catch (...) {
            std::cout << "caught exception (1)" << std::endl;
            throw;
        }
    } catch (...) {
        std::cout << "caught exception (2)" << std::endl;
    }

    return 0;
}
```



```
caught exception (1)
caught exception (2)

Process finished with exit code 0
```

Аналогичный синтаксис сработает и в случае с не **catch-all** блоком, то есть тип исключения, которое повторно выбрасывается не имеет значения.

Копирование исключения при выбрасывании

До сих пор, мы выбрасывали лишь примитивные типы (такие как **char***, **int**, и так далее).
Изучим же, что происходит при попытке выбросить локально инстанцированный именованный объект.
Оказывается, что в этом случае, оператор **throw** копирует его даже в том случае, если мы получаем его по константной ссылке в блоке catch:

```
struct logger {
    logger() = default;
    logger (const logger&) { std::cout << "copy" << std::endl; }
};

int main () {
    try {
        logger l;
        std::cout << &l << std::endl;
        throw l;
    } catch (const logger& l) {
        std::cout << &l << std::endl;
    }

    return 0;
}
```



```
0x7ff7b70d4818
copy
0x7fdb52704490
```

Без копирования, пока что непонятно как переместить объект из одного scope’е (порождаемого **try**) в scope блока **catch**.

***Примечание**: в будущем мы познакомимся с другим способом делать подобное перемещение объекта более эффективно, ведь полное копирование объекта – дорогая операция, и это проблема.*

Копирование исключения при выбрасывании

Рассмотрим также другой пример, в котором выбросим наследника, обрезанного до базового класса с использованием ссылки:

```
struct base {
    base () = default;
    base (const base&) { std::cout << "base copied" << std::endl; }
};

struct derived : base {
    derived () = default;
    derived (const derived&) { std::cout << "derived copied" << std::endl; }
};

int main () {
    try {
        derived d;
        base& b = d;
        throw b;
    } catch (const base& b) {
        std::cout << "base caught" << std::endl;
    } catch (const derived& d) {
        std::cout << "derived caught" << std::endl;
    }

    return 0;
}
```

```
base copied
base caught

Process finished with exit code 0
```

В этом случае объект скопируется исходя из типа выброшенного выражения, то есть скопируется как **base**, а не как **derived**.

***Вывод**: оператор **throw** – не полиморфен.*

Копирование исключения при выбрасывании

Вывод из прошлых двух слайдов прост: будем стараться всегда выбрасывать временный (неименованный) объект. В этом случае копирования не произойдет, так как сработает простая оптимизация компилятора, благодаря которой временный объект будет будто бы “перемещен” сразу же в **scope** блока **catch**:

```
struct logger {
    logger() = default;
    logger (const logger&) { std::cout << "copy" << std::endl; }
};

int main () {
    try {
        throw logger();
    } catch (const logger& l) {
        std::cout << &l << std::endl;
    }

    return 0;
}
```



```
0x7fc2dc704490


Process finished with exit code 0
```

Исключения в конструкторах

```
struct ugly {
    ugly (): _arr(new int[100]) {
        std::cout << "memory allocated" << std::endl;
        throw 123;
    }

    ~ugly () {
        delete[] _arr;
        std::cout << "memory deallocated" << std::endl;
    }

private:
    int* const _arr;
};

int main () {
    try {
        ugly u;
        /// ...
    } catch (...) {
        std::cout << "exception caught" << std::endl;
    }

    return 0;
}
```

Несмотря на наличие деструктора, вызван он не будет, ведь исключение возникло до завершения процесса конструирования объекта **u**, а значит деструкторировать его не надо.

```
memory allocated
exception caught

Process finished with exit code 0
```

Ввиду этого, возникнет утечка памяти. Таким образом, использовать исключения в конструкторах **не рекомендуется**.

Исключения в деструкторах

```cpp
struct ugly {
    ugly () = default;

    ~ugly () {
        throw 123;
    }
};

int main () {
    try {
        ugly u;
        throw 456;
    } catch (...) {
        std::cout << "exception caught" << std::endl;
    }

    return 0;
}
```



```
libc++abi: terminating with uncaught exception of type int

Process finished with exit code 134 (interrupted by signal 6: SIGABRT)
```

Использование исключений в деструкторах приводит к еще большим проблемам.

Например, в примере слева, в процессе выбрасывания исключения **456**, мы натыкаемся на исключение **123** во время деструктурирования объекта u, который находится в scope блока try, который в свою очередь нужно очистить при выбрасывании исключения.

Ввиду этого, программа падает с ошибкой. Таким образом, использовать исключения в деструкторах **не рекомендуется**.

Спецификация исключений до и после C++11

Примеры исключений

Рассмотрим примеры исключений на сайте C++ reference

TODO: обработка исключения от dynamic_cast ссылки

Создание собственного исключения

Исключения могут иметь произвольный тип. Классы-исключения можно создавать и самостоятельно, ведь мы уже выбрасывали объекты.

Для этого, необходимо унаследовать класс **std::exception** и переопределить **const**- и **noexcept**-метод **what**, который возвращает сообщение-ошибку. Теперь сообщение-ошибка попадает в консоль:

```
struct my_exception : std::exception {
    my_exception (const char* description): _description(description) {}

    const char* what () const noexcept override {
        return _description;
    }

private:
    const char* _description;
};

int main () {
    throw my_exception("Hello from my exception");
}
```

```
libc++abi: terminating with uncaught exception of type my_exception: Hello from my exception

Process finished with exit code 134 (interrupted by signal 6: SIGABRT)
```

Полезное свойство

Напоследок, рассмотрим пример с сайта C++ reference

The throw-expression is classified as prvalue expression of type `void`. Like any other expression, it may be a sub-expression in another expression, most commonly in the conditional operator:

```
double f(double d)
{
    return d > 1e7 ? throw std::overflow_error("too big") : d;
}

int main()
{
    try
    {
        std::cout << f(1e10) << '\n';
    }
    catch (const std::overflow_error& e)
    {
        std::cout << e.what() << '\n';
    }
}
```

TODO: исправление примера с sqrt(x)/(x-1)

HIGHER SCHOOL OF ECONOMICS
NATIONAL RESEARCH UNIVERSITY

Лекция 5

Исключения

Программирование на языке C++

Константин Леладзе
ФКН ВШЭ