Исключения II

Исключения в конструкторах и деструкторах

Исключения в конструкторах

В чем здесь проблема?

```
struct A {
  std::vector<int> v;
  int* ptr;

  A() : v(100), ptr(new int) {
    f();  // потенциально бросает исключение
  }

  ~A() {
    delete ptr;
  }
};
```

Исключения в конструкторах

```
struct A {
  std::vector<int> v;
  int* ptr;

  A() : v(100), ptr(new int) {
    f();  // потенциально бросает исключение
  }

  ~A() {
    delete ptr;
  }
};
```

Так как конструктор не завершил работу, объект не считается созданным, а это значит, что деструктор вызван не будет! Утечка памяти (ptr)

(И не выполнено правило пяти)

Исключения в конструкторах

Решение: перехватить исключение, очистить память и бросить исключение дальше.

```
struct A {
  std::vector<int> v;
  int* ptr;

  A() : v(100), ptr(new int) {
    try {
      f();  // потенциально бросает исключение
    } catch (...) {
      delete ptr;
      throw;
    }
  }
  ~A() {
    delete ptr;
  }
};
```

Исключения в конструкторах

А что будет с вектором?

```
struct A {
  std::vector<int> v;
  int* ptr;

  A() : v(100), ptr(new int) {
    try {
      f();  // потенциально бросает исключение
    } catch (...) {
      delete ptr;
      throw;
    }
  }
  ~A() {
    delete ptr;
  }
};
```

Исключения в конструкторах

С вектором все будет нормально - объект вектора был создан, а значит для него будет вызван деструктор.

```
struct A {
  std::vector<int> v;
  int* ptr;

  A() : v(100), ptr(new int) {
    try {
      f();  // потенциально бросает исключение
    } catch (...) {
      delete ptr;
      throw;
    }
  }
  ~A() {
    delete ptr;
  }
};
```

Исключения в конструкторах

Если вы используете RAII, то проблем совсем нет.

```
struct A {
  std::vector<int> v;
  std::unique_ptr<int> ptr;

    A() : v(100), ptr(std::make_unique<int>(0)) {
      f();  // потенциально бросает исключение
    }
};
```

Правило нуля в действии.

Исключения в деструкторах

Очевидная проблема - утечка ресурсов, которую решить довольно просто:

```
struct A {
  std::vector<int> v;
  int* ptr;

    // ...

  ~A() {
    try {
      f(ptr);  // потенциально бросает исключение
    } catch (...) {
      delete ptr;
      throw;
    }
    delete ptr;
  }
};
```

При этом для вектора в любом случае будет вызван деструктор.

Исключения в деструкторах

Допустим, мы позволяем исключениям покидать деструкторы.

Видите ли вы проблему?

```
void h() {
  A a;  // Деструктор может бросить исключение
  g();  // Может бросить исключение
    // ...
}
```

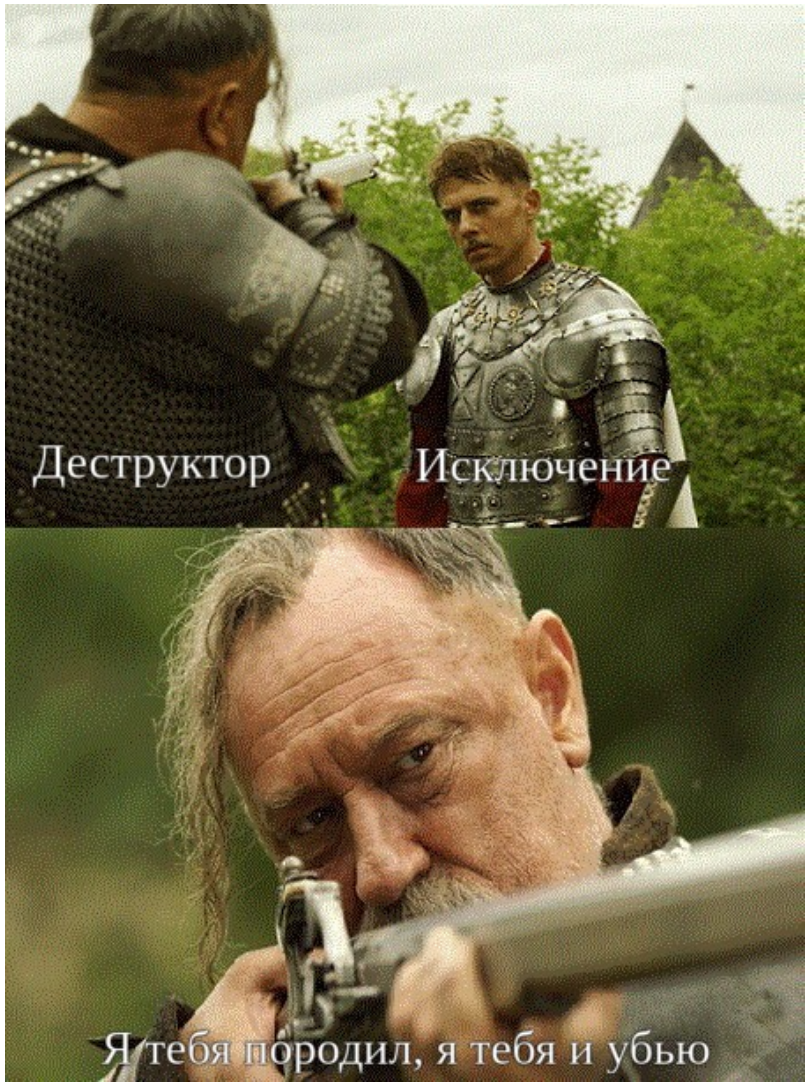
Исключения в деструкторах

Сценарий такой: допустим `g()` бросает исключение, начинается раскручивание стека, вызывается деструктор `A` и в нем снова бросается исключение.
Итог - из одной функции летит 2 исключения (UB).

```
void h() {
  A a;  // Деструктор может бросить исключение
  g();  // UB, если исключение будет брошено
    // ...
}
```

Исключения в деструкторах

Мораль: *не позволяйте исключениям покидать деструкторы*



Исключения в деструкторах

В современном C++ все деструкторы по умолчанию помечены как `noexcept`, поэтому вылет исключения из деструктора приводит к аварийному завершению программы.

.

.

.

.

.

.

.

.

.

(но, если очень хочется, то `noexcept(false)` и https://en.cppreference.com/w/cpp/error/uncaught_exception)

Гарантии безопасности исключений

Гарантии безопасности исключений

Давайте формализуем понятие "безопасный код" с помощью так называемых *гарантий безопасности*.

Гарантия безопасности отвечает на вопрос, в каком состоянии находится система после возникновения ошибки.

Гарантии безопасности:

1. Гарантия отсутствия исключений
2. Базовая гарантия безопасности
3. Строгая гарантия безопасности

Гарантия отсутствия исключений

Самая простая (для понимания) гарантия. Функция удовлетворяет гарантии отсутствия исключений, если она никогда не бросает исключений.

Пример.

```
template <class T>
size_t Stack<T>::Size() const noexcept {  // удовлетворяет гарантии
  return size_;
}
```

```
template <class T>
Stack<T>::Stack(const Stack<T>&);  // не удовлетворяет гарантии
// может бросить, например, при нехватке памяти
```

Базовая гарантия безопасности

Функция удовлетворяет *базовой гарантии безопасности*, если после возникновения исключения все компоненты программы находятся в согласованном (валидном состоянии), утечки ресурсов не произошло.

Строгая гарантия безопасности

Функция удовлетворяет *строгой гарантии безопасности*, если после возникновения исключения все компоненты программы находятся в **том же состоянии**, что и до вызова, утечки ресурсов не произошло.

Внезапно: для чего нужен `noexcept`?

К сожалению, "эффективный код" и "безопасный код" далеко не всегда совместимы друг с другом.

Часто для достижения безопасности приходится обходиться без наиболее эффективных функций (в них могут возникать неожиданные ошибки).

`noexcept` позволяет дать понять другим функциям, что она безопасна, и что ее можно спокойно использовать, не переживая за возможные ошибки.

Иерархия исключений C++

Вопросы

Хорошая ли идея бросать `int`?

Хорошая ли идея ловить "что угодно"?

Исключения-классы

В C++ есть договоренность - бросать исключения классовых типов (тип дает информацию о природе исключений)

```
class DivisionByZero {
  std::string info_;

 public:
  explicit DivisionByZero(const char* info) noexcept : info_(info) {
  }
  const char* What() const noexcept {
    return info_.c_str();
  }
};

template <class T>
T Divide(T x, T y) {
  if (y == 0) { throw DivisionByZero("some info..."); }
  return x / y;
}
```

Исключения-классы

```
class DivisionByZero {
  std::string info_;

 public:
  explicit DivisionByZero(const char* info) noexcept : info_(info) {
  }
  const char* What() const noexcept {
        return info_.c_str();
  }
};

int main() {
  try {
    Divide(1, 0);
  } catch (const DivisionByZero& error) {
    std::cerr << error.What() << '\n';
  } catch (...) {
    // а что произошло здесь?
  }
}
```

Исключения-классы

Давайте сделаем следующее: заведем общий базовый класс `Exception`, а все остальные исключения будем наследовать от него.

```
class Exception {
 public:
  virtual const char* What() const noexcept { return "Exception"; }
  virtual ~Exception() = default;
};
class DivisionByZero : public Exception {
  // ...
  const char* What() const noexcept override { return "DivisionByZero"; }
};

int main() {
  try {
    Divide(1, 0);
  } catch (const Exception& ex) {
    std::cerr << ex.What() << '\n';  // DivisionByZero
  }
}
```

Стандартная библиотека исключений C++

В C++ есть готовый базовый класс исключений - `std::exception`.

Он содержит единственный виртуальный метод - `what()`.

Все стандартные классы исключений унаследованы от него:

`std::logic_error`, `std::runtime_error`, `std::bad_cast` (бросает `dynamic_cast`), `std::bad_alloc` (бросает `new` при неудаче), `std::bad_weak_ptr` и много других

От них, в свою очередь, могут быть унаследованы (уточнены) другие исключения. Например, `std::out_of_range` унаследован от `std::logic_error`, а `std::bad_any_cast` от `std:bad_cast`

Стандартная библиотека исключений C++

Таким образом, можем группировать исключения по степени общности и по смыслу:

```
int main() {
  try {
    f();  // потенциально бросает исключения
  }
  catch (std::out_of_range& oor) { /* обрабатываем выход за границы */ }
  catch (std::logic_error& le) { /* обрабатываем logic_error'ы */ }
  catch (std::runtime_error& re) { /* обрабатываем runtime_error'ы */ }
  catch (std::exception& ex) { /* обрабатываем все остальное */ }
}
```

Для тех, кто не спит: для чего принимаем по ссылке?

Стандартная библиотека исключений C++

Свои классы исключений также стоит наследовать от одного из стандартных классов ошибок

```
class DivisionByZero : public std::runtime_error {
public:
    using std::runtime_error::runtime_error;
};
```

Бонус 1: function try блок

Неприятность

Что, если хочется навесить блок `try` на все тело функции?

```
void f() {
  try {
    // ...
  } catch (std::exception& exception) {
    // ...
  }
}
```

Выглядит не очень

Проблема

```
class B {
  int* ptr_;
  A a_;
 public:
  B() : ptr_(new int(11)), a_(0) {  // <--!
  }
  // ...
};
```

Что, если при конструировании `a_` вылетит исключение?

Решение: function try block

Блок `try-catch` можно навесить на функцию целиком (вместе со списком инициализации)

```
void f() try {
  // ...
} catch (std::exception& exception) { /* ... */ }
```

```
class B {
  int* ptr_;
  A a_;
 public:
  B() try : ptr_(new int(11)), a_(0) {  // <--!
    // ...
  } catch (...) {
    delete ptr_; throw;
  }
  // ...
};
```

Упражнение: найти баг в предложенном решении

Бонус 2: Метаинформация о функции

Макросы `__func__`, `__LINE__`, `__FILE__`:
https://en.cppreference.com/w/c/language/function_definition

`std::source_location` (C++20):
https://en.cppreference.com/w/cpp/utility/source_location