Закон Мёрфи

Соловьёв Владимир Валерьевич
Huawei, НГУ, СУНЦ
vladimir.conwor@gmail.com
vk.com/conwor

Если какая-нибудь неприятность может произойти, она обязательно случится

Если написанная программа сработала правильно, значит, во время её работы выполнилось чётное число ошибок

Некорректные ситуации (ошибки исполнения)

- В функцию, ожидающую положительный аргумент, передали -42
- Файл, заданный пользователем, не существует или поломан
- Попытались прочитать поле по указателю, равному NULL
- Компьютер, к которому пытаемся обратиться по сети, недоступен
- Вышли за границы массива
- В текстовый калькулятор ввели строку “3/0”

Невероятные некорректные ситуации

Ошибки программиста, более сложные, чем умеет проверять компилятор

Прерывают процесс работы программы, как правильно, фатально

После отладки не должны возникать у пользователей

Невероятные некорректные ситуации

Ошибки программиста, более сложные, чем умеет проверять компилятор

Прерывают процесс работы программы, как правильно, фатально

После отладки не должны возникать у пользователей (по идее)

Вероятные некорректные ситуации

Ошибки окружения программы - входных данных или иных обстоятельств

Должны быть обработаны, после чего программа продолжает работать

Могут возникать у пользователей

Некорректные ситуации

- В функцию, ожидающую положительный аргумент, передали -42
- Файл, заданный пользователем, не существует или поломан
- Попытались прочитать поле по указателю, равному NULL
- Компьютер, к которому пытаемся обратиться по сети, недоступен
- Вышли за границы массива
- В текстовый калькулятор ввели строку “3/0”

Некорректные ситуации (вероятные)

- В функцию, ожидающую положительный аргумент, передали -42
- **Файл, заданный пользователем, не существует или поломан**
- Попытались прочитать поле по указателю, равному NULL
- **Компьютер, к которому пытаемся обратиться по сети, недоступен**
- Вышли за границы массива
- **В текстовый калькулятор ввели строку “3/0”**

Некорректные ситуации (невероятные)

- **В функцию, ожидающую положительный аргумент, передали -42**
- Файл, заданный пользователем, не существует или поломан
- **Попытались прочитать поле по указателю, равному NULL**
- Компьютер, к которому пытаемся обратиться по сети, недоступен
- **Вышли за границы массива**
- В текстовый калькулятор ввели строку “3/0”

Некорректные ситуации

Точное определение - вероятна ошибка или нет - может дать только программист, исходя из смысла, вкладываемого в программу

Невероятные некорректные ситуации

```
// X should be positive!!!
void foo(int x) {
        …
}
```

```
foo(-42)
```

Невероятные некорректные ситуации

```
// X should be positive!!!
void foo(unsigned int x) {
        …
}
```

```
foo(-42)
```

Невероятные некорректные ситуации

```
// X should be positive!!!
void foo(unsigned int x) {
        …
}
```

```
foo(-42)
```

Вообще, перевести невероятную ситуацию в ошибки компиляции - хорошая идея

Невероятные некорректные ситуации

```
// X should be positive!!!
void foo(unsigned int x) {
        …
}
```

```
foo(-42)
```

Вообще, перевести невероятную ситуацию в ошибки компиляции - хорошая идея
Но в данном случае - это даже не предупреждение

Невероятные некорректные ситуации

```
// X should be positive!!!
void foo(int x) {
        if (x <= 0) {
                std::cout << X should be positive in Foo!”;
                exit(1);
        }
}
```

Невероятные некорректные ситуации

```
// X should be positive!!!
void foo(int x) {
        if (x <= 0) {
                std::cout << X should be positive in Foo!”
                exit(1);
        }
}
```

Исполняется у пользователя!
В другом примере это может быть очень сложный код

Невероятные некорректные ситуации

```
// X should be positive!!!
void foo(int x) {
        if (x <= 0) {
                std::cout << X should be positive in Foo!”
                exit(1);
        }
}
```

Много слов, путается с логикой программы

Невероятные некорректные ситуации

```
// X should be positive!!!
void foo(int x) {
        if (x <= 0) {
                std::cout << X should be positive in Foo!”
                exit(1);
        }
        if (x == 478) {
                ...
        }
        ...
}
```

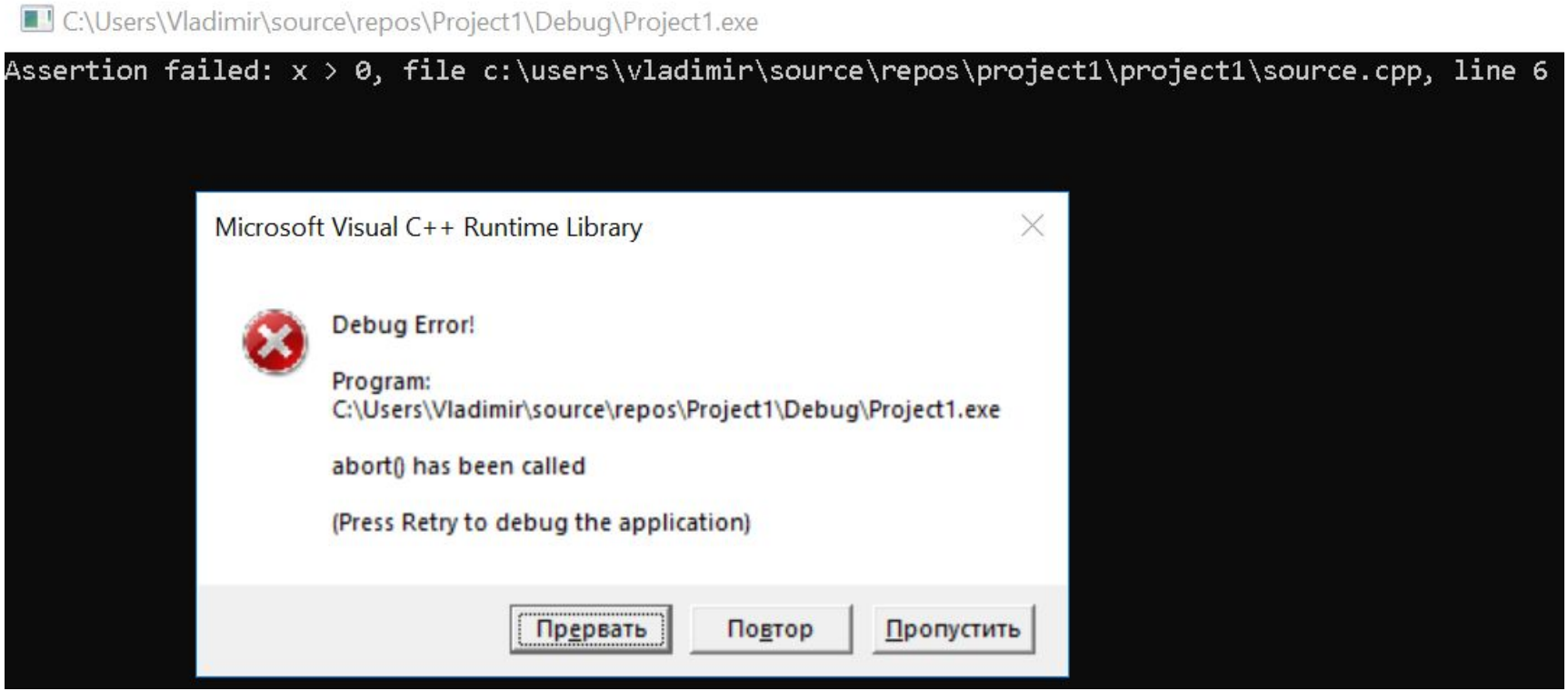
Много слов, путается с логикой программы

Ассёрты

```
#include <cassert>

// X should be positive!!!
void foo(int x) {
        assert(x > 0);
        ...
}
```

Ассёрты


C:\Users\Vladimir\source\repos\Project1\Debug\Project1.exe
Assertion failed: x > 0, file c:\users\vladimir\source\repos\project1\project1\source.cpp, line 6
Microsoft Visual C++ Runtime Library
Debug Error!
Program:
C:\Users\Vladimir\source\repos\Project1\Debug\Project1.exe
abort() has been called
(Press Retry to debug the application)
Прервать
Повтор
Пропустить

Ассёрты

Визуально отличаются от нормальной логики программы

При развале выдают строку с местом развала

Отключаются определением константы NDEBUG (GCC, MSVC, ...)

Отключение ассёртов

```
#define NDEBUG
#include <cassert>

// X should be positive!!!
void foo(int x) {
        assert(x > 0);
        ...
}
```

Отключение ассёртов

Можно указать компилятору, какие константы определены

в Visual Studio, например, Проект -> Свойства -> C/C++ -> Препроцессор -> Определения препроцессора

Таким образом - включены assert’ы или нет, контролируется конфигурацией сборки (например, для тестирования или для пользователя)

Отключение ассёртов

Не пишите в ассёртах код с побочными эффектами!

Отключение ассёртов

```
int globalVar;

int foo() {
        ...
        globalVar = 42;
        …
        if (something_good)
                return 0;
        else
                return -1;
}
```

```
assert(foo() == 0);
```

Отключение ассёртов

```
int globalVar;

int foo() {
        ...
        globalVar = 42;
        …
        if (something_good)
                return 0;
        else
                return -1;
}
```

~~assert(foo() == 0);~~

```
int r = foo();
assert(r == 0);
```

Вероятные некорректные ситуации

```
int getFileSize(char* name) {
        FILE* f = fopen(name, “r”);
        … fstat(_fileno(f), ...) ...
}
```

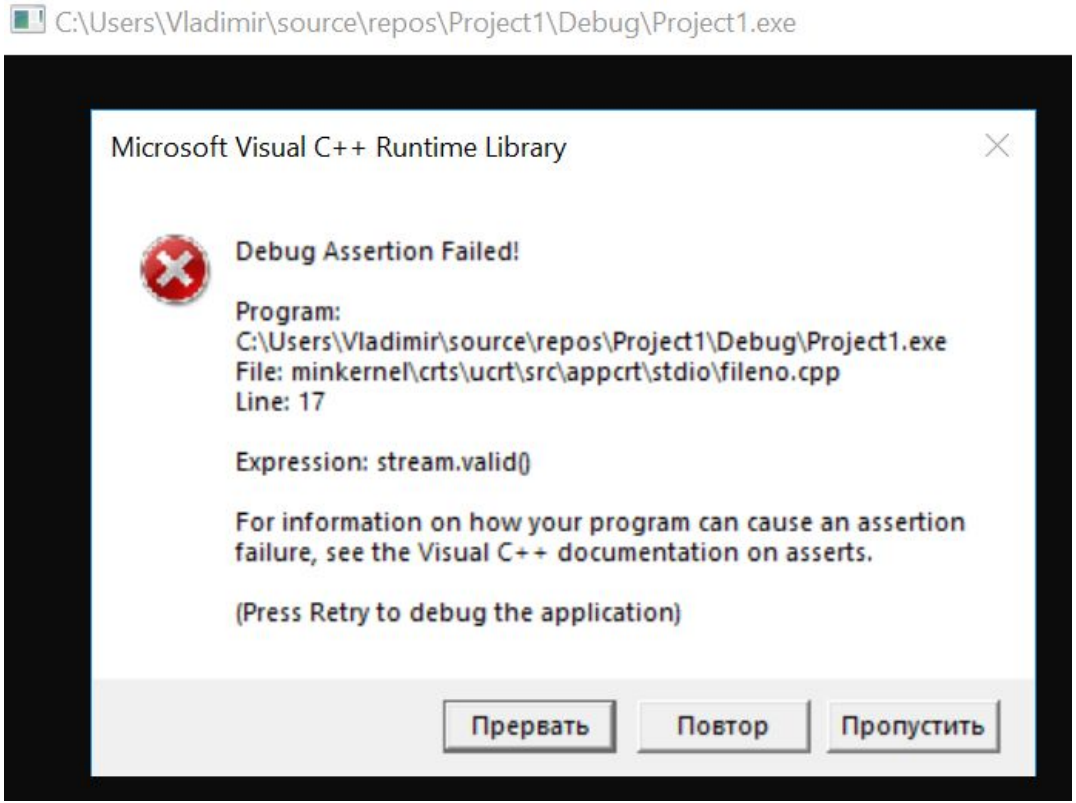
Вероятные некорректные ситуации

```
int getFileSize(char* name) {
        FILE* f = fopen(name, “r”); // f может быть NULL
        … fstat(_fileno(f), ...) ...
}
```

Вероятные некорректные ситуации

```
int getFileSize(char* name) {
        FILE* f = fopen(name, “r”); // f может быть NULL
        … fstat(_fileno(f), ...) ...
}
```

Вероятные некорректные ситуации



Вероятные некорректные ситуации

```
int getFileSize(char* name) {
        FILE* f = fopen(name, “r”); // f может быть NULL
        … fstat(_fileno(f), ...) ...
}
```

Вероятные некорректные ситуации

```
int getFileSize(char* name) {
        FILE* f = fopen(name, “r”); // f может быть NULL
        … fstat(_fileno(f), ...) ...
}
```

Вероятные некорректные ситуации

```
int getFileSize(char* name) {
        FILE* f = fopen(name, “r”); // f может быть NULL
        … fstat(_fileno(f), ...) ...
}
```

Это событие может быть вызвано тем, что пользователь ввёл неправильное имя файла

Вероятные некорректные ситуации

```
int getFileSize(char* name) {
        FILE* f = fopen(name, “r”); // f может быть NULL
        … fstat(_fileno(f), ...) ...
}
```

Это событие может быть вызвано тем, что пользователь ввёл неправильное имя файла

Вместо фатального развала нужно сообщить об этом пользователю и продолжить работу

Вероятные некорректные ситуации

```
int getFileSize(char* name) {
        FILE* f = fopen(name, “r”); // f может быть NULL
        … fstat(_fileno(f), ...) ...
}
```

Это событие может быть вызвано тем, что пользователь ввёл неправильное имя файла

Вместо фатального развала нужно сообщить об этом пользователю и продолжить работу

Ассёрт не подходит

Вероятные некорректные ситуации

```
int getFileSize(char* name) {
        FILE* f = fopen(name, “r”);
        while (f == NULL) {
                std::cout << “File do not exist, try again!\n”;
                std::cin >> name;
                f = fopen(name, “r”);
        }
        … fstat(_fileno(f), ...) ...
}
```

Вероятные некорректные ситуации

```
int getFileSize(char* name) {
        FILE* f = fopen(name, “r”);
        while (f == NULL) {
                std::cout << “File do not exist, try again!\n”;
                std::cin >> name;
                f = fopen(name, “r”);
        }
        … fstat(_fileno(f), ...) ...
}
```

Что эта функция себе позволяет?

Вероятные некорректные ситуации

getFileSize была простой (библиотечной) функций и вдруг оказалась нагружена логикой приложения

Представьте себе, что тем же самым вдруг бы занялась функция fopen. Было бы удобно её использовать?

Вероятные некорректные ситуации

Нужна механика возврата, но не значения, а сообщения об ошибке

Если функция способна обработать ошибку, она это делает

Если не способна - возвращает ошибку тому, кто её вызвал

Вероятные некорректные ситуации

```
int getFileSize(char* name) {
        FILE* f = fopen(name, “r”);
        if (f == NULL) {
                return -1;
        }
        … fstat(_fileno(f), ...) ...
}
```

Вероятные некорректные ситуации

```
int getFileSize(char* name) {
        FILE* f = fopen(name, “r”);
        if (f == NULL) {
                return -1;
        }
        … fstat(_fileno(f), ...) ...
}
```

Если не всё множество значений возвращаемого типа корректно, остаток можно использовать для детектирования ошибки

Вероятные некорректные ситуации

```
int readFirstDigitFromFile(char* name) {
        FILE* f = fopen(name, “r”);
        if (f == NULL) {
                return -1;
        }
        ...
}
```

Вероятные некорректные ситуации

```
int readFirstDigitFromFile(char* name) {
        FILE* f = fopen(name, “r”);
        if (f == NULL) {
                return -1;
        }
        ...
}
```

Часто невозможно переиспользовать возвращаемое значение

Вероятные некорректные ситуации

```
int readFirstDigitFromFile(char* name, int* errCode) {
        *errCode = 0;
        FILE* f = fopen(name, “r”);
        if (f == NULL) {
                *errCode = 1;
                return 0;
        }
        ...
}
```

Часто невозможно переиспользовать возвращаемое значение

Добавим функции результатов!

Вероятные некорректные ситуации

```
int readFirstDigitFromFile(char* name, int* errCode) {
        *errCode = 0;
        FILE* f = fopen(name, “r”);
        if (f == NULL) {
                *errCode = 1;
                return 0;
        }
        ...
}
```

Возвращаемое значение всё равно приходится выдумывать

Если бы из функции возвращался объект по значению, пришлось бы создавать какой-то фейк

Вероятные некорректные ситуации

```
int readFirstDigitFromFile(char* name, int* errCode) {
        *errCode = 0;
        FILE* f = fopen(name, “r”);
        if (f == NULL) {
                *errCode = 1;
                return 0;
        }
        ...
}
```

Если захотим вместе с ошибкой нести больше информации, чем просто её код, придётся увеличивать этот хвост параметров

Вероятные некорректные ситуации

```
int readFirstDigitFromFile(char* name, int* errCode) {
        *errCode = 0;
        FILE* f = fopen(name, “r”);
        if (f == NULL) {
                *errCode = 1;
                return 0;
        }
        ...
}
```

Плохо читается, смешиваясь с логикой нормальной работы функции

Вероятные некорректные ситуации

```
int readFirstDigitFromFile(char* name, int* errCode) {
        *errCode = 0;
        FILE* f = fopen(name, “r”);
        if (f == NULL) {
                *errCode = 1;
                return 0;
        }
        ...
}
```

Давайте ещё посмотрим, во что превратился вызов

Вероятные некорректные ситуации

```
void foo(char* name) {
        …
        int errCode;
        int digit = readFirstDigitFromFile(name, &errCode);
        if (errCode != 0) {
                ...
        }
        ...
}
```

Вероятные некорректные ситуации

```
void foo(char* name) {
        …
        int errCode;
        int digit = readFirstDigitFromFile(name, &errCode);
        if (errCode != 0) {
                ...
        }
        ...
}
```

Также плохо читается, смешиваясь с логикой работы функции

Вероятные некорректные ситуации

```
void foo(char* name) {
        …
        int errCode;
        int digit = readFirstDigitFromFile(name, &errCode);
        if (errCode != 0) {
                ...
        }
        ...
}
```

Если эта функция неспособна обработать ошибку, она должна проделать то же самое!

Вероятные некорректные ситуации

```
void foo(char* name, int* errCode) {
        …
        int digit = readFirstDigitFromFile(name, &errCode);
        if (errCode != 0) {
                return;
        }
        ...
}
```

Вероятные некорректные ситуации

```
void foo(char* name, int* errCode) {
        …
        int digit = readFirstDigitFromFile(name, &errCode);
        if (errCode != 0) {
                return;
        }
        ...
}
```

А ещё функция может какие-то типы ошибок уметь обрабатывать, а какие-то нет

Вероятные некорректные ситуации

```
void foo(char* name, int* errCode) {
        …
        int digit = readFirstDigitFromFile(name, &errCode);
        if (errCode == 37) {
                ...
        }
        if (errCode != 0) {
                return;
        }
        ...
}
```

А ещё функция может какие-то типы ошибок уметь обрабатывать, а какие-то нет

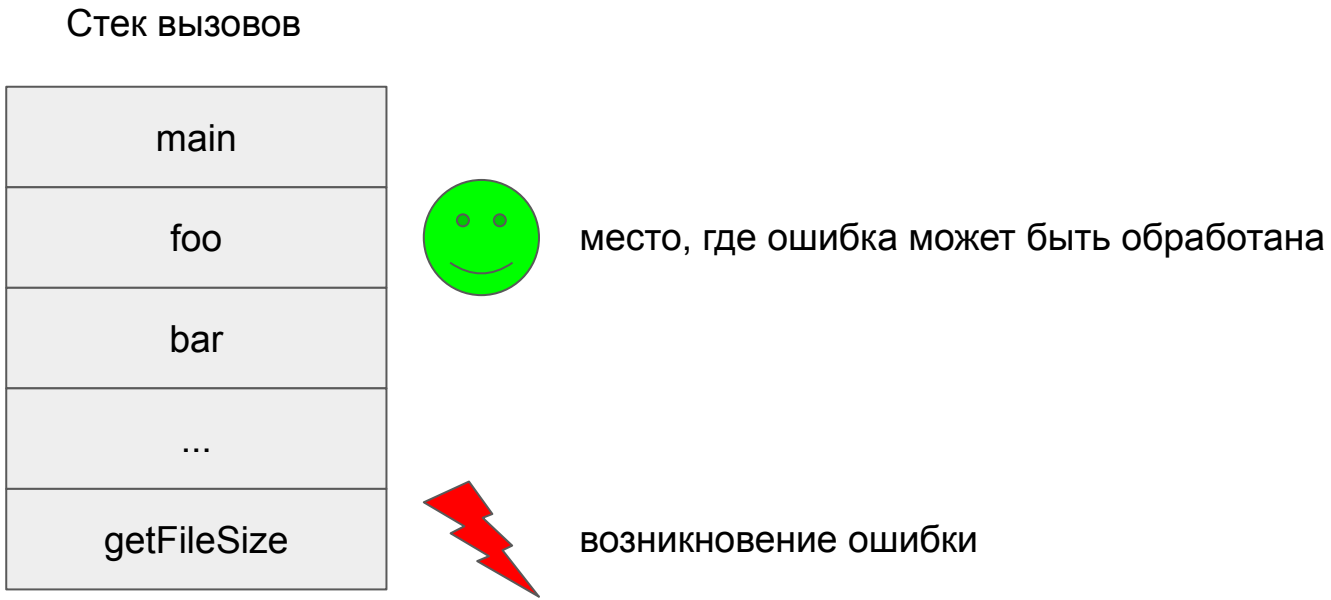
Ручная передача ошибок

Очень сильно засоряет весь код, включая тот, который никакого отношения к ошибкам не имеет

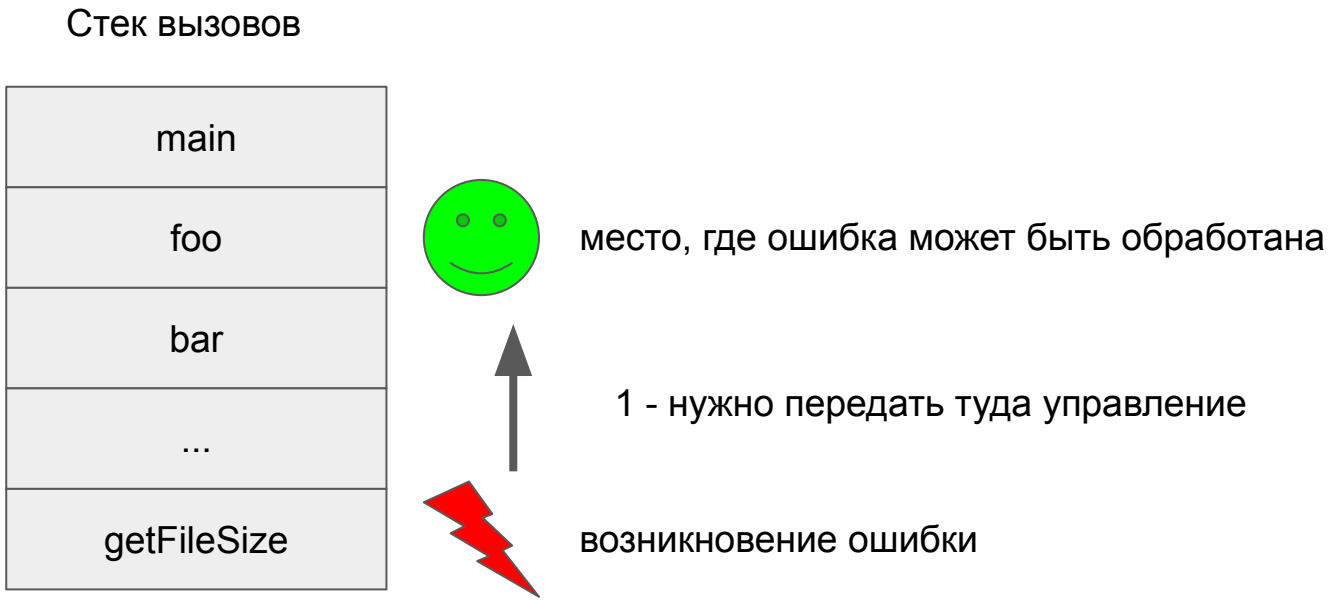
Осмыслим проблему

Стек вызовов

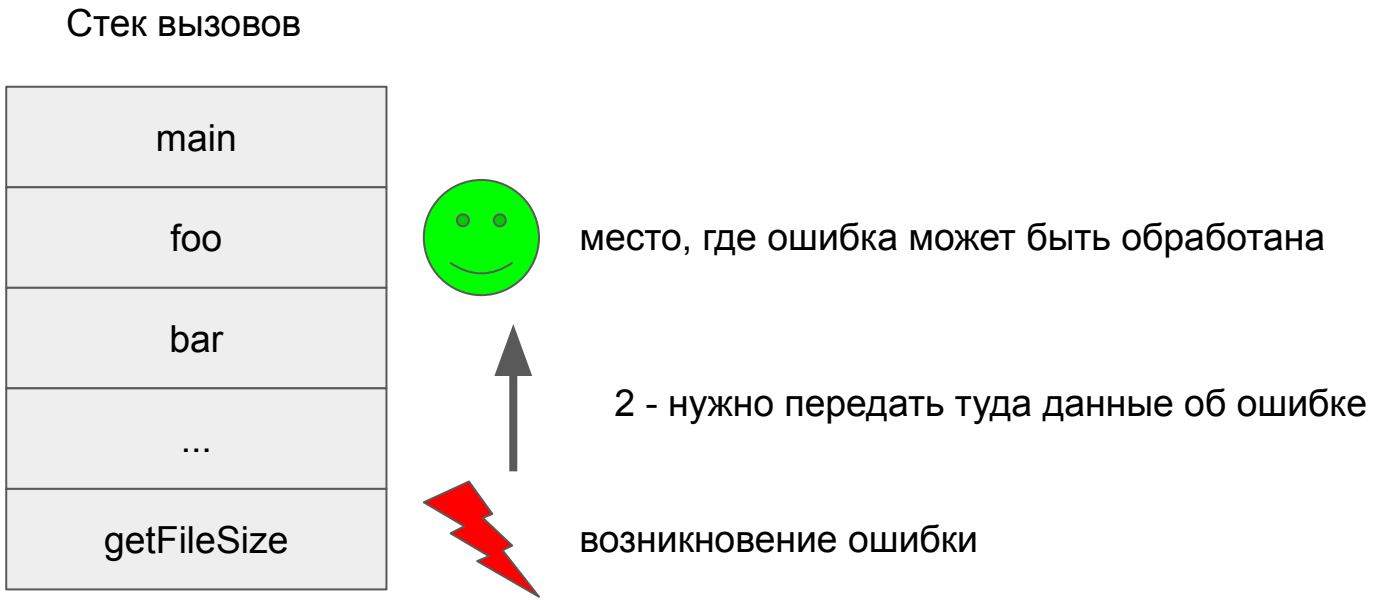
Осмыслим проблему



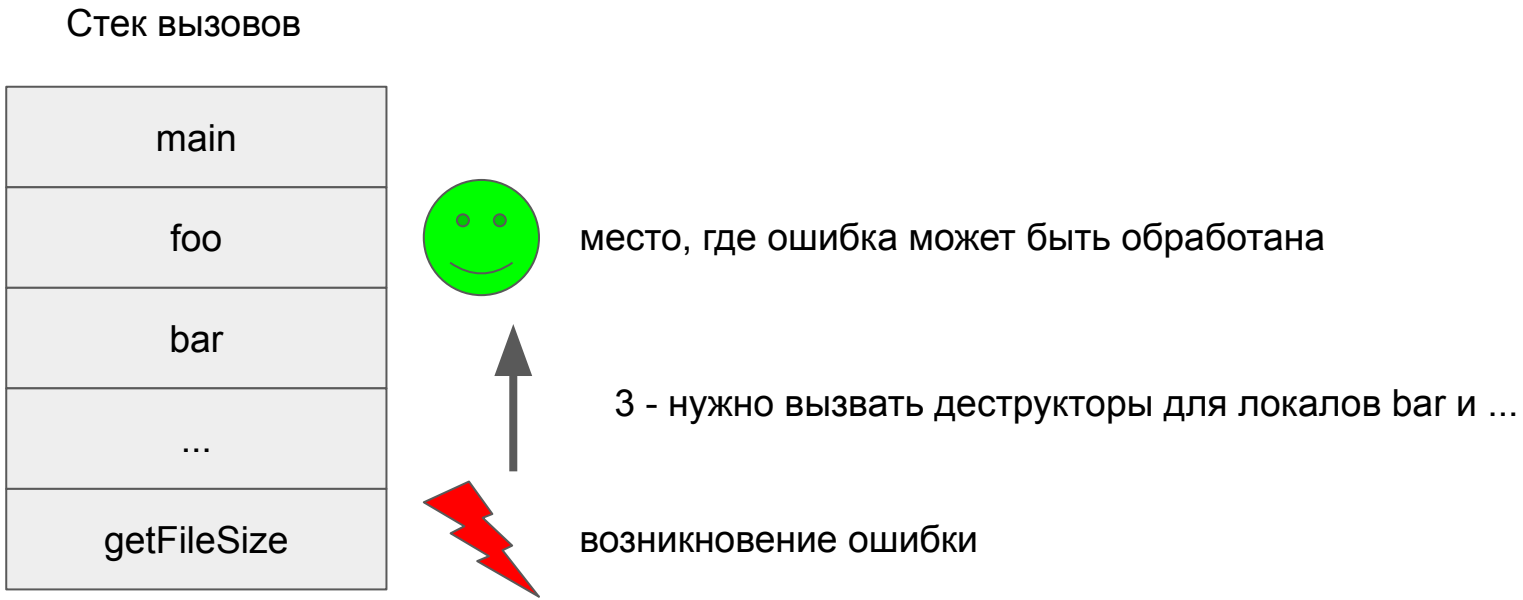
Осмыслим проблему



Осмыслим проблему



Осмыслим проблему



setjmp/longjmp

setjmp вызывается в точке обработчика, запоминая указатель на стек

longjmp переходит в эту точку, подправляя стек и перенося один int

setjmp/longjmp

```
jmp_buf barCall;

void foo() {
        …
        int errCode = setjmp(barCall);
        if (errCode == 0) {
                bar();
                ...
        } else {
                // обработка ошибки
        }
}
```

```
void getFileSize(char* name) {
        FILE* f = fopen(...);
        if (f == NULL) {
                longjmp(barCall, 37);
        }
        ...
}
```

setjmp/longjmp

Передаёт всё равно мало данных (хотя можно воспользоваться глобалами)

При наличии множества точек обработки и множества точек потенциальных ошибок превращается в малопонятную кашу

setjmp/longjmp

Передаёт всё равно мало данных (хотя можно воспользоваться глобалами)

При наличии множества точек обработки и множества точек потенциальных ошибок превращается в малопонятную кашу

Не вызывает деструкторы для локальных объектов

Исключения

setjmp/longjmp с человеческим лицом, передачей произвольного объекта и вызовом деструкторов для всего счищаемого стека

Исключения

```
void foo() {
        …
        try {
                bar();
        }
        catch (int errCode) {
                // обработка ошибки
        }
}
```

```
void getFileSize(char* name) {
        FILE* f = fopen(...);
        if (f == NULL) {
                throw 37;
        }
        ...
}
```

Семантика

Аргумент оператора throw - **исключение**. В C++ это может быть что угодно.

Если точка, где написан оператор throw, находится внутри try-блока, происходит попытка обработки исключения

Исключение сравнивается по типу с тем, что описано в обработчике

Семантика

Если типы подходят, управление передаётся в обработчик, и исключение считается **обработанным**

Иначе (или если throw не написан внутри try-блока), вызываются деструкторы для локальных объектов, и исключение передаётся (**выбрасывается**) следующему методу по стеку

Семантика

Для следующего метода происходит всё то же самое, только вместо точки throw используется точка вызова метода, который выбросил исключение

Если исключение долетает до начала стека, оно ловится системным обработчиком

Обработчиков может быть несколько

```
void foo() {
        …
        try {
                bar();
        }
        catch (int errCode) {
                ...
        }
        catch (char* message) {
                ...
        }
}
```

Универсальный обработчик

```
void foo() {
        …
        try {
                bar();
        }
        catch (int errCode) {
                ...
        }
        catch (...) {
                ...
        }
}
```

Полиморфизм подтипов

```
class A { … };
class B : public A { … };
```

```
void foo() {
        if (...) {
                B b;
                throw b;
        }
}
```

```
void bar() {
        try {
                foo();
        }
        catch (A a) {
                ...
        }
}
```

Полиморфизм подтипов

```
class A { … };
class B : public A { … };
```

```
void foo() {
        if (...) {
                B b;
                throw b;
        }
}
```

Но при приёме объект “огрубится” до A

```
void bar() {
        try {
                foo();
        }
        catch (A a) {
                ...
        }
}
```

Полиморфизм подтипов

```
class A { … };
class B : public A { … };
```

```
void foo() {
        if (...) {
                throw new B();
        }
}
```

На указателях полиморфзим работает полностью, то есть от указателя будут работать виртуальные функции с реализациями из класса B

```
void bar() {
        try {
                foo();
        }
        catch (A* a) {
                ...
        }
}
```

Полиморфизм подтипов

Выброс по значению приводит к копированию объекта, а обработка по значению - к ещё одному копированию

Вместе с полуработающим полиморфизмом чаще всего лучше бросать исключения по ссылкам (один конструктор копий) или указателям (нет конструкторов копий, но нужно удалить объект вручную в обработчике)

Всё хорошо

1 - передача управления в обработчик без мешанины с логикой программы

2 - выбросить можно произвольный объект, то есть, любые данные

3 - деструкторы вызываются, как при ручной передаче ошибки

Сюрпризы

```
class Foo {
public:
        ~Foo() {
                if (...) {
                        throw 37;
                }
        }
};
```

```
void bar() {
        Foo f;
        throw 42;
}
```

```
void baz() {
        try {
                bar();
        }
        catch (int x) {
                std::cout << x;
        }
}
```

Сюрпризы

В процессе выброса исключения работают деструкторы

Если они выбросят своё исключение, это фатально завершит всю программу

Параллельно два исключения лететь не могут

Лучше не бросать исключений из деструкторов

Сюрпризы

```
class Foo {
public:
        Foo() {
                if (...)
                        throw 37;
        }
};
```

```
void bar() {
        Foo f;
}
```

Сюрпризы

```
class Foo {
        int* buff;
public:
        Foo() {
                buff = new int[...];
                if (...)
                        throw 37;
        }
        ~Foo() {
                delete[] buff;
        }
};
```

```
void bar() {
        Foo f;
}
```

Сюрпризы

```
class Foo {
        int* buff;
public:
        Foo() {
                if (...)
                        throw 37;
                buff = new int[...];
        }
        ~Foo() {
                delete[] buff;
        }
};
```

```
void bar() {
        Foo f;
}
```

Сюрпризы

Если исключение вылетит из конструктора, деструктор для него не вызовется

Если в конструкторе до исключения произошёл захват ресурсов (память, файлы, сетевые сокеты, ...), то они не освободятся

Сюрпризы

```
void foo() {
        int* arr = new int[...];
        bar();
        delete[] arr;
}
```

```
void bar() {
        if (...)
                throw 42;
}
```

Сюрпризы

```
void foo() {
        int* arr = new int[...];
        try {
                bar();
        }
        catch (int x) {
                delete[] arr;
                throw x;
        }
        delete[] arr;
}
```

```
void bar() {
        if (...)
                throw 42;
}
```

Сюрпризы

```
void foo() {
        int* arr = new int[...];
        try {
                bar();
        }
        catch (int x) {
                delete[] arr;
                throw x;
        }
        delete[] arr;
}
```

```
void bar() {
        if (...)
                throw 42;
}
```

Возвращаемся к аду из начала лекции - код, не имеющий отношения к ошибкам, ими занимается

Сюрпризы

```
void foo() {
        int* arr = new int[...];
        try {
                bar();
        }
        catch (int x) {
                delete[] arr;
                throw x;
        }
        delete[] arr;
}
```

```
void bar() {
        if (...)
                throw 42;
}
```

А если bar начнёт бросать другие типы?

Сюрпризы

```
void foo() {
        int* arr = new int[...];
        try {
                bar();
        }
        catch (int x) {
                delete[] arr;
                throw x;
        }
        delete[] arr;
}
```

```
void bar() {
        if (...)
                throw 42;
}
```

Копипаста

Сюрпризы

```
void foo() {
        int* arr = new int[...];
        try {
                bar();
        }
        catch (int x) {
                delete[] arr;
                throw x;
        }
        delete[] arr;
}
```

```
void bar() {
        if (...)
                throw 42;
}
```

А если в foo появятся новые захваты ресурсов?

Сюрпризы

```
void foo() {
        int* arr = new int[...];
        try {
                bar();
        }
        catch (int x) {
                delete[] arr;
                throw x;
        }
        delete[] arr;
}
```

```
void bar() {
        if (...)
                throw 42;
}
```

А если в foo появятся новые захваты ресурсов и новые вызовы вперемешку друг с другом?

RAII идиома

Resource Acquisition Is Initialization (получение ресурса есть инициализация)

Захватываемые ресурсы нужно связывать с объектом, удаление которого гарантируется независимо от процесса исполнения

RAII идиома

Resource Acquisition Is Initialization (получение ресурса есть инициализация)

Захватываемые ресурсы нужно связывать с объектом, удаление которого гарантируется независимо от процесса исполнения (локальным)

RAII идиома

```
class IntArr {
	int* arr;
public:
	IntArr(int size) {
		arr = new int[size];
	}

	~IntArr() {
		delete[] arr;
	}
};
```

```
void foo() {
	IntArr x(1024);
	bar();
}
```

RAII идиома

```
class IntArr {
        int* arr;
public:
        IntArr(int size) {
                arr = new int[size];
        }

        ~IntArr() {
                delete[] arr;
        }
};
```

```
void foo() {
        IntArr x(1024);
        bar();
}
```

x удалится (а вместе с ним и освободится ресурс arr) независимо от того, как закончится функция foo

RAII идиома

```
class Foo {
        IntArr buff;
public:
        Foo(int size): buff(size) {
                if (...)
                        throw 37;
        }
};
```

```
void bar() {
        Foo f;
}
```

Несмотря на то, что деструктор для f не будет вызываться, его поля будут уничтожены (с вызовом их деструкторов)

Рекомендации

1) Бросайте объекты классов, наследующих std::exception

2) Не используйте исключения для регулярной логики программы

3) Полюбите RAII идиому

Убедитесь, что вынесли с этой лекции

Вероятные и невероятные некорректные ситуации

Ассёрты, способ их отключения

Ручная передача ошибок

setjmp/longjmp

Исключения

RAII идиома

Проверочные вопросы

1) Какие плюсы у ассёртов по сравнению с ручными проверками?
2) В чём минусы ручной передачи ошибок?
3) Почему setjmp/longjmp техника плохо подходит для C++?
4) Какие объекты можно кидать, как исключения?
5) В чём суть RAII идиомы?
6) Чем чреват выброс исключения в деструкторе

   * - продумайте, как применить RAII идиому в случае, если вам нужно захватить ресурс и передать его назад по стеку

Q & A