

# Искусство отладки

Денис Адамчук



# Структура лекции

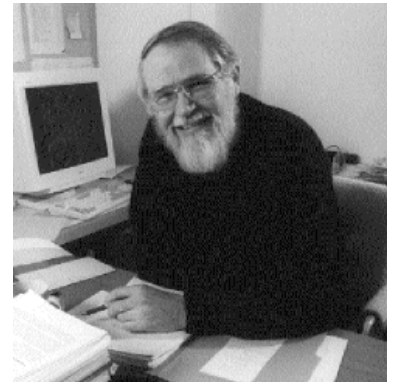
## Основные разделы и подразделы.

- ❑ Введение
- ❑ Отладка в контексте разработки
  - Неизбежность отладки
  - Процесс разработки
  - Тестирование
  - Информационное и программное обеспечение
- ❑ Инструменты отладки ПО
  - Отладчик
  - Профилировщик
  - Детектор утечек памяти
  - Анализаторы кода
- ❑ Отладчик Visual Studio Code
  - Точки останова
  - Выполнение программы
  - Стек вызовов
  - Просмотр переменных
  - Debug и Release
- ❑ Дампы памяти
- ❑ Типовые ошибки
- ❑ Q&A

# Введение

**Отладка** – это поиск причин (или изучение) и устранение ошибок в программе.

“ Отлаживать код вдвое сложнее, чем писать.  
Если Вы используете весь свой интеллект  
при написании программы, вы по определению  
недостаточно умны, чтобы её отладить. ”



**Брайан Керниган**



# Отладка в контексте разработки

- Неизбежность отладки
- Отладка и процесс разработки
- Алгоритм поиска и устранения ошибки
- Информационное и программное обеспечение



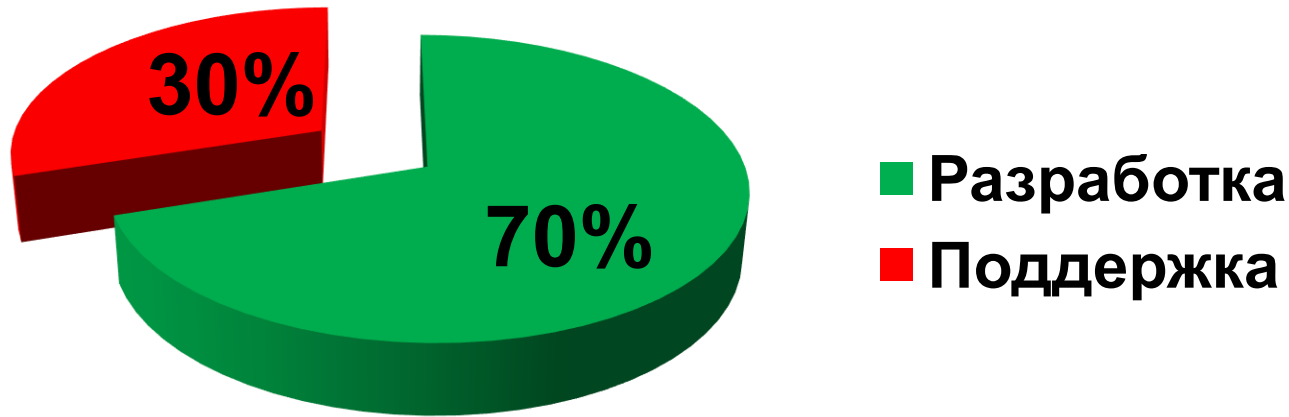


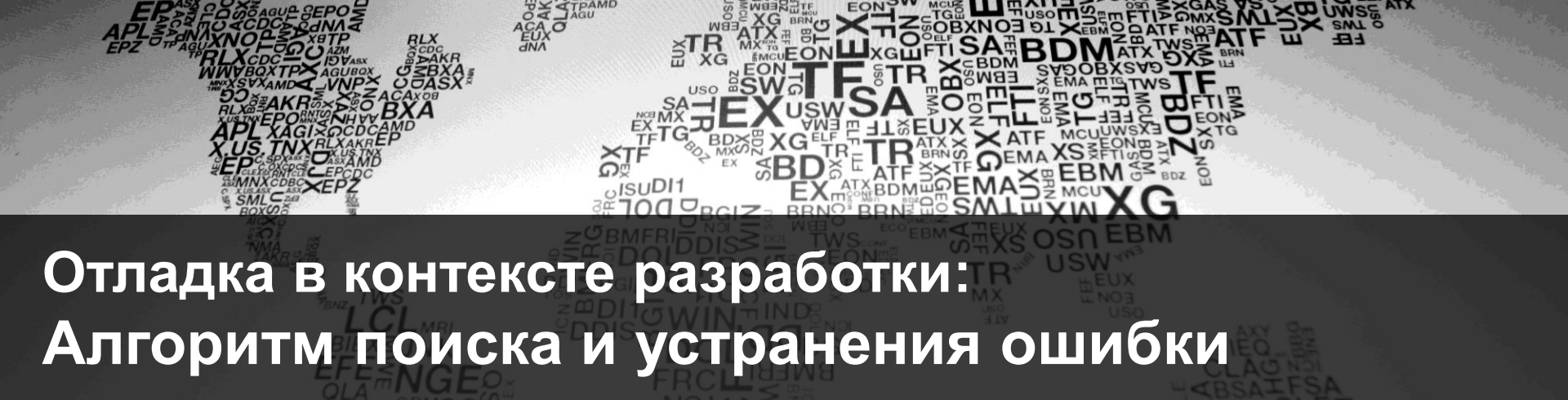
# Отладка в контексте разработки : Неизбежность отладки

- Человеческий фактор
- Недостаток знаний и навыков
- Компромисс между качеством и сроками
- Неясные и меняющиеся требования

# Отладка в контексте разработки: Процесс разработки

В среднем 30% времени разработчиков тратится на поиск и исправление ошибок в существующем ПО





# Отладка в контексте разработки: Алгоритм поиска и устранения ошибки

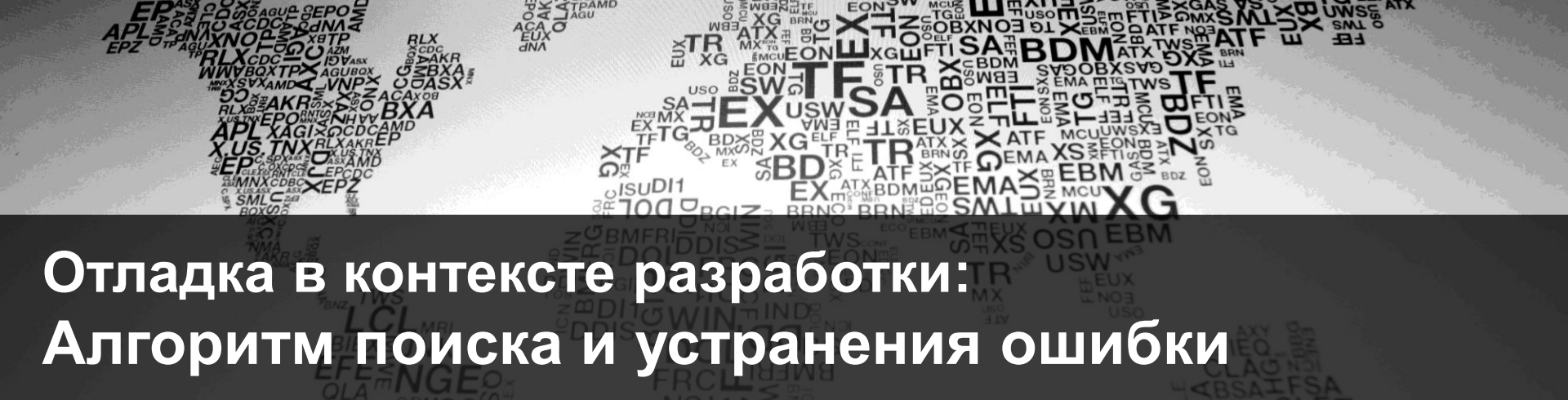
## Основные шаги:

Воспроизведение (reproducing)

Исследование (investigation)

Исправление (fixing)

Проверка (testing)

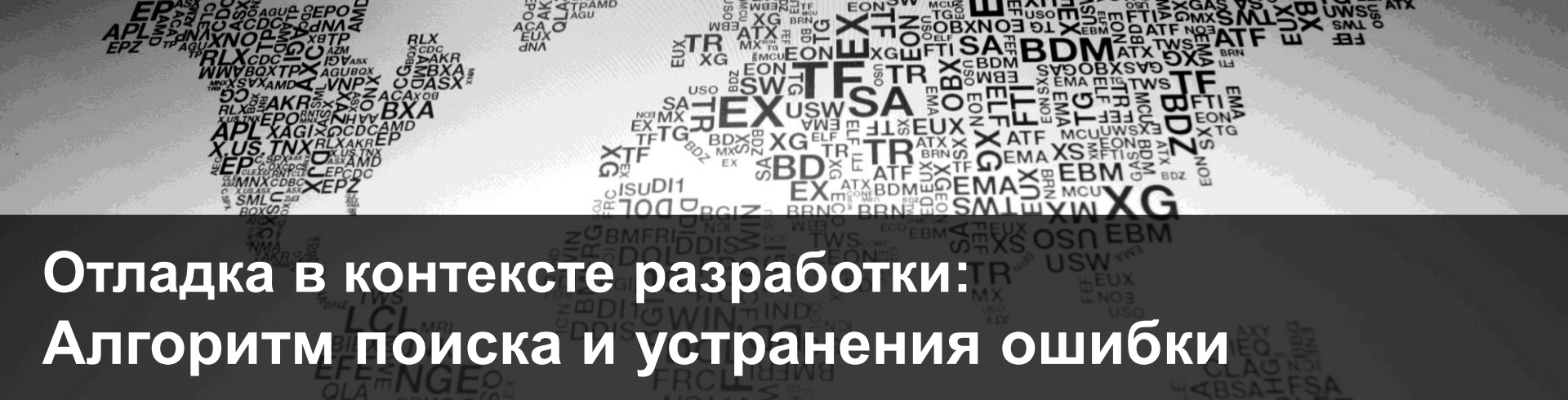


# Отладка в контексте разработки: Алгоритм поиска и устранения ошибки

## Основные шаги. Воспроизведение.

- Воспроизведенная проблема – 50% успеха
- Воспроизведение может быть долгим
- Что помогает воспроизвести проблему:
  - Инструкции или подсказки от пользователя
  - Код
  - Логи
  - Конфигурация
  - Отслеживание изменений в коде
  - Настойчивость и изобретательность
- Формальный результат: шаги или ничего

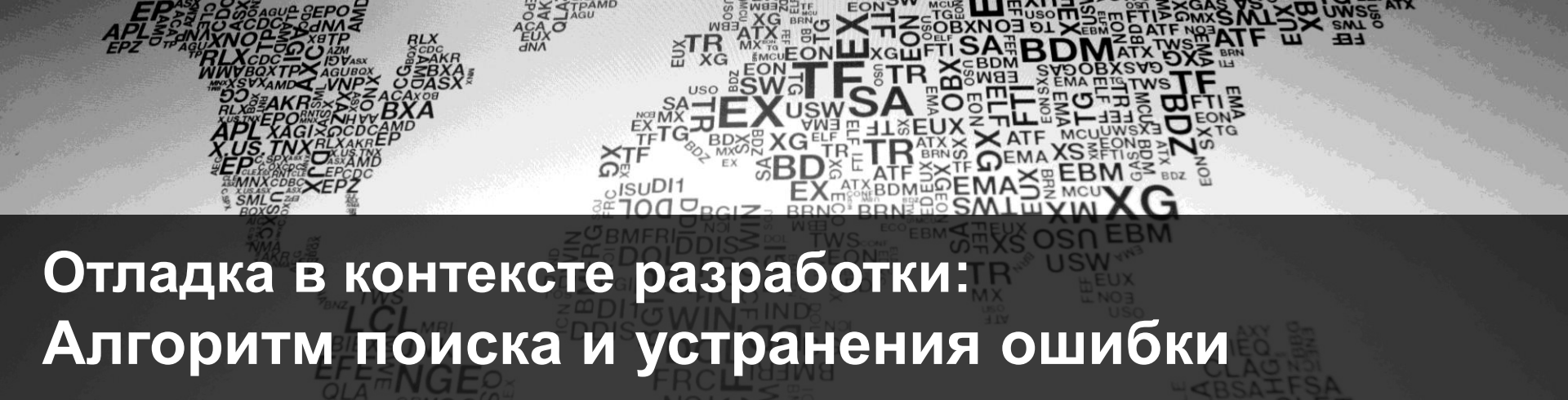




# Отладка в контексте разработки: Алгоритм поиска и устранения ошибки

## Основные шаги. Исследование.

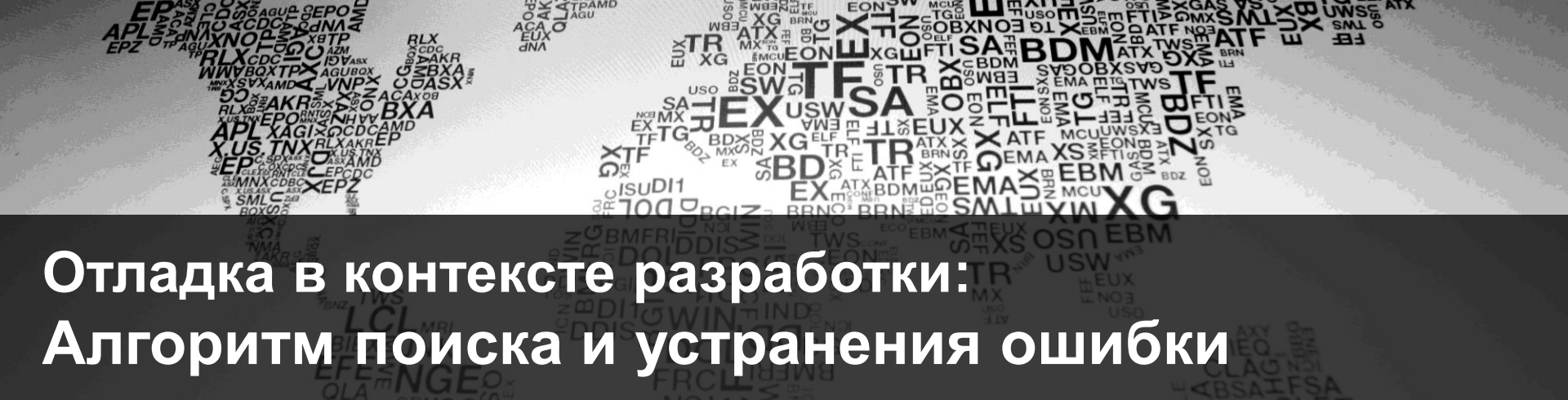
- Длительность этого этапа зависит от успешности предыдущего
- Что помогает исследованию:
  - Код
  - Отладчик и другие инструменты отладки
  - Анализ требований и документации к продукту
  - Описания коммитов
  - Общение с коллегами
- Формальный результат: ясное понимание причины ошибки или публикация описания ошибки в *Системе отслеживания ошибок*



# Отладка в контексте разработки: Алгоритм поиска и устранения ошибки

## Основные шаги. Исправление.

- Проблему в коде исправить проще, чем проблему в дизайне
- Что помогает исправлению:
  - Отладчик и другие инструменты отладки
  - Общение с коллегами
  - Ясное понимание того, какое поведение является верным
  - Общение с бизнес-экспертами
  - Наличие юнит-тестов
  - Принцип «не навреди» (особенно в преддверии релиза)
- Формальный результат: коммит в *Системе контроля версий*



# Отладка в контексте разработки: Алгоритм поиска и устранения ошибки

## Основные шаги. Проверка.

- Не стоит пренебрегать проверкой.
- Возможно, нужна проверка смежных областей.
- Что помогает проверке:
  - Ревью кода
  - Наличие плана для ручного тестирования
  - Наличие автоматических тестов
  - Возможность дать пощупать фикс опытным коллегам
- Формальный результат: баг закрыт в *Системе отслеживания ошибок*



# Отладка в контексте разработки: Тестирование

Виды тестирования:

- Ручное
- Полуавтоматическое (запуск сценариев вручную)
- Автоматическое (Continuous Integration)

Этапы тестирования:

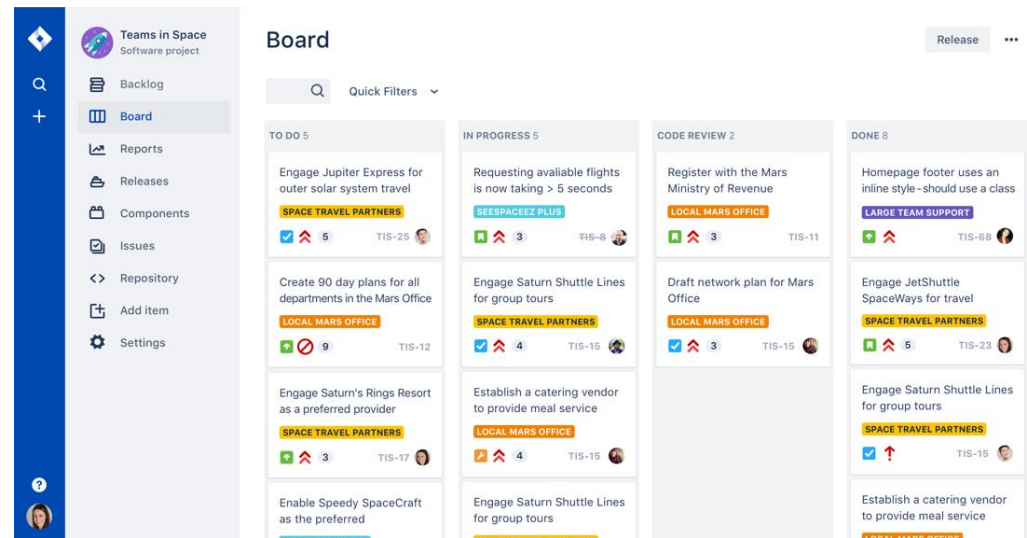
- Unit Testing – тестирование поведения **модулей** и функций в коде.
- Integration Testing – тестирование **взаимодействия** нескольких систем программы.
- System Testing – тестирование **новой** функциональности на соответствие требованиям.
- Regression Testing – проверка того, что программа **продолжает** работать в соответствии с требованиями после сделанных изменений в коде.



# Отладка в контексте разработки: Информационное и программное обеспечение

## Системы отслеживания ошибок

- Распределение задач между разработчиками
- Расстановка приоритетов
- Хранение информации о всех известных ошибках





# Отладка в контексте разработки: Информационное и программное обеспечение

## Системы отслеживания ошибок

Возможности:

- Отслеживание состояния задач по исправлению ошибок
- Хранение базы знаний по всем известным ошибкам.  
Иногда это может сохранить многие дни работы.

Состояние абстрактной проблемы:

- Проблема неизвестна (нет в *Системе отслеживания*)
- Проблема известна и находится в работе у кого-то
- Проблема известна и ждет, пока до нее дойдут руки
- Проблема известна, но не планируется к исправлению
- Проблема известна, и уже исправлена



# Отладка в контексте разработки: Информационное и программное обеспечение

## Рекомендации по системе отслеживания ошибок

Если вы обнаружили какую-то ошибку, самым первым действием должен быть поиск в системе отслеживания ошибок:

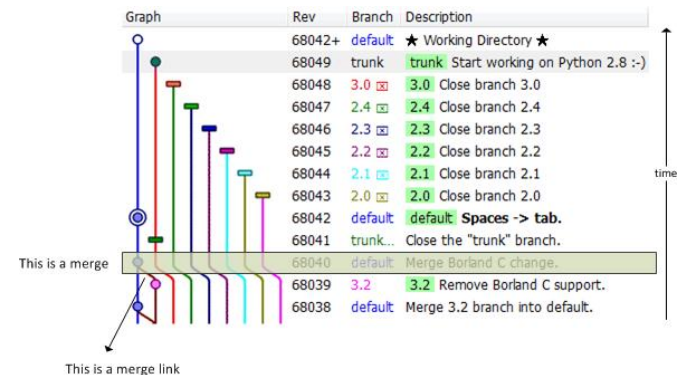
- Возможно, эта проблема уже известна и над ней работают,
- Возможно, эта проблема уже исправлена, и вам просто надо обновить свою версию программы.
- Возможно, эта проблема известна, но ее решили не исправлять по каким-то причинам, и тогда вам стоит либо учиться жить с этой проблемой, либо пересмотреть прошлое решение и начать исправлять.
- Возможно, эта проблема была исправлена, но почему-то проявилась снова, и тогда багтрекер, может быть, укажет вам, куда стоит начать смотреть.
- Возможно, вы найдете похожую проблему, а в ней – какую-то полезную информацию.



# Отладка в контексте разработки: Информационное и программное обеспечение

## Системы контроля версий

- Хранение полной истории изменений кода
- Анализ изменений в коде
- Управление версиями программы







# Отладка в контексте разработки: Информационное и программное обеспечение

## Системы контроля версий

Возможности для поиска ошибки:

- Поиск конкретной версии, если известны шаги для воспроизведения проблемы (git bisect)
- Поиск изменения (коммита), если известна версия, в которой появилась проблема
- Поиск изменений в коде, если известен файл, в котором проблема
- Описания коммитов могут раскрыть причины, почему подозрительный код написан так, а не иначе



# Отладка в контексте разработки: Информационное и программное обеспечение

## Системы контроля версий

Рекомендации:

- Одно небольшое изменение соответствует одному коммиту
- В каждом коммите должно быть описание причин его появления
- Изменение большого количества кода в разных файлах делает систему контроля бесполезной



# Отладка в контексте разработки: Информационное и программное обеспечение

## Логирование

Логи – файлы, содержащие диагностическую информацию:

- Действия пользователя перед появлением проблемы
- Изменения внутреннего состояния программы, предшествующие проблеме
- Переменные среды окружения
- Приложения, выполняющиеся на компьютере пользователя
- Информация о компьютере пользователя  
(модель процессора, размер монитора, объем памяти и т.д.)



# Отладка в контексте разработки: Информационное и программное обеспечение

## Логирование

Возможности хорошей системы логирования:

- Несколько уровней логирования
- Ротация лог-файлов
- Возможность записи сообщений не только в файлы
- Поток-безопасность
- Асинхронное логирование
- Настройка формата записей





# Отладка в контексте разработки: Информационное и программное обеспечение

## Уровни логирования

Trace	Полная информация обо всех изменениях состояния
Debug	Подробная внутренняя информация о работе программы.
Information	Краткая информация об изменении состояния программы.
Warning	Программа находится в неожиданном или нестандартном состоянии. Лучше не игнорировать.
Error	Явная ошибка в работе программы. Программа в целом продолжает работу.
Fatal	Ошибка, приводящая к неработоспособности всей программы или подсистемы.

# Отладка в контексте разработки: Информационное и программное обеспечение

## Логирование

Пример лог-файла:

```
app_2022_09_09_14_47_00.log (C:\Users\adenis) - GVIM1
1 2022-09-09 14:47:16.345+03:00 [INFO] [0x00006c04] [core] App.EXE/JETS: creating context for session 75
2 2022-09-09 14:47:29.071+03:00 [WARN] [0x000075b4] [net] 172.26.130.45: Destination host unreachable.$
3 2022-09-09 14:47:30.309+03:00 [TRACE] [0x000023ab] [net] 35 bytes sent to 172.26.130.111$
4 2022-09-09 14:47:30.317+03:00 [INFO] [0x000075b4] [util] Image conversion from Image.JPG to Image.PNG succeeded$
```



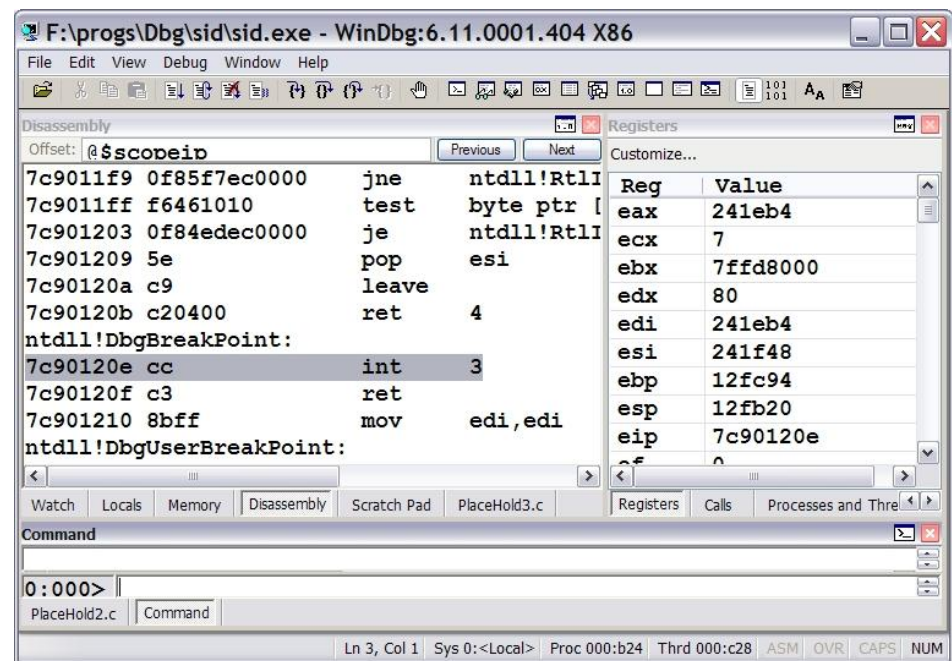
# Инструменты отладки ПО

- Отладчик (Debugger)
- Профилировщик (Profiler)
- Детектор утечек памяти (Leak Detector)
- Статический анализатор кода (Static Analyzer)
- Динамический анализатор кода (Dynamic Analyzer)

# Инструменты отладки ПО: Отладчик (Debugger)

Возможности:

- Приостановка программы
- Пошаговое исполнение
- Отслеживание и изменение значений переменных и произвольных участков памяти
- Настройка точек останова



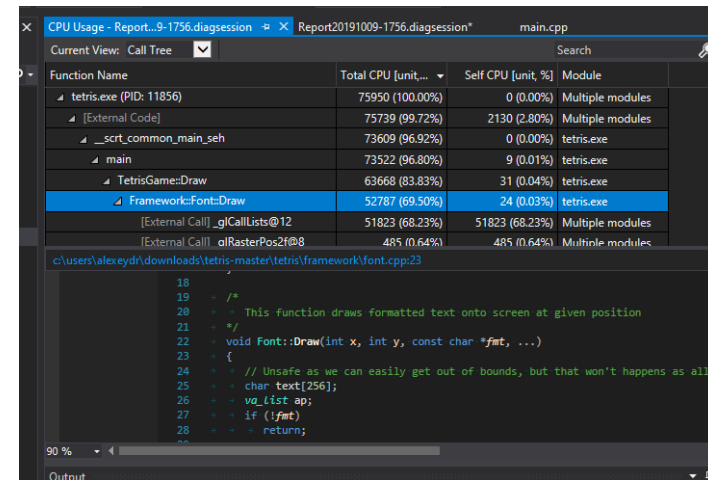


# Инструменты отладки ПО: Профилировщик (Profiler)

## Возможности:

- Анализ отдельных функций и строк кода.
- Анализ конкретного временного диапазона.
- Анализ конкретного потока программы.
- Определяет количество вызовов и время, которое проводится внутри функции.

Только профилировщик способен подсказать, где скрываются узкие места вашей программы.



## Примеры:

- Visual Studio
- Windows Performance Analyzer

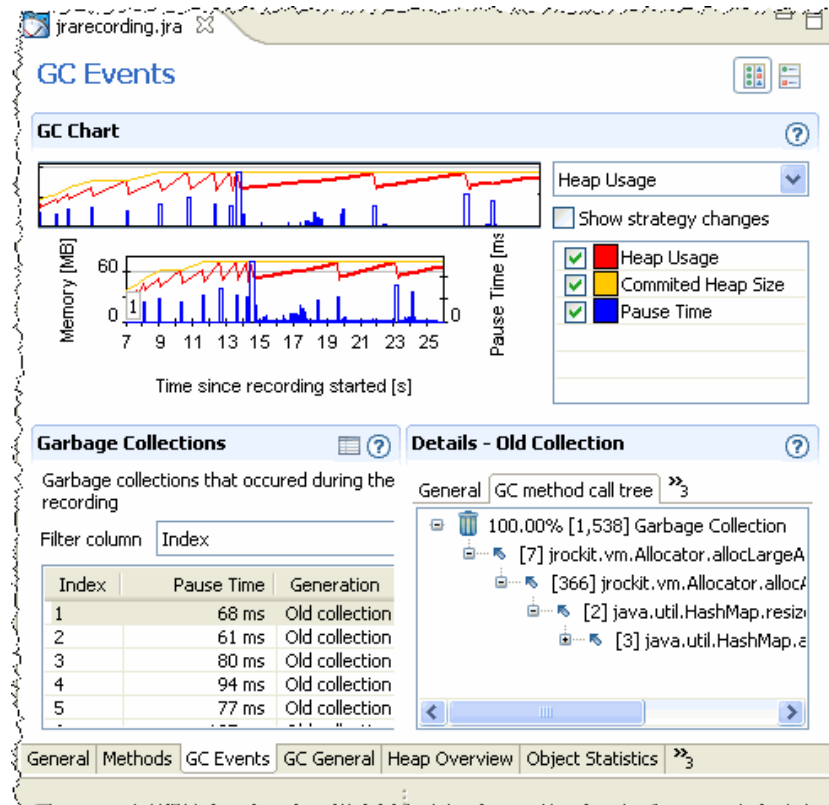
# Инструменты отладки ПО: Детектор утечек памяти

Возможности:

- Выявление “утечки” памяти (в языках с ручным управлением памятью)
- Выявление неоптимального использования памяти

Примеры:

- Visual Studio
- Visual Leak Detector
- Intel Inspector
- Valgrind



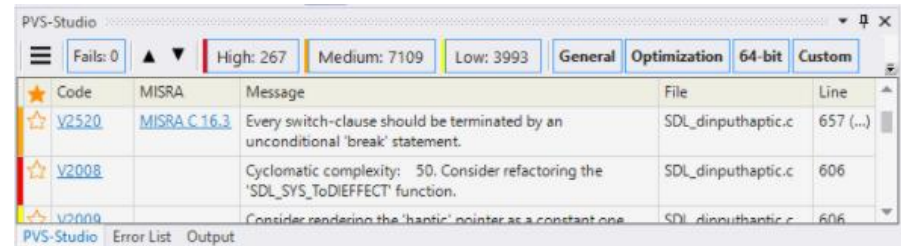
# Инструменты отладки ПО: Статический анализатор кода

Статический анализатор:

- Анализирует исходный код на предмет логических ошибок и подозрительных конструкций.

Примеры:

- PVS-Studio
- Visual Studio Static Analysis
- clang, gcc
- CPPCheck





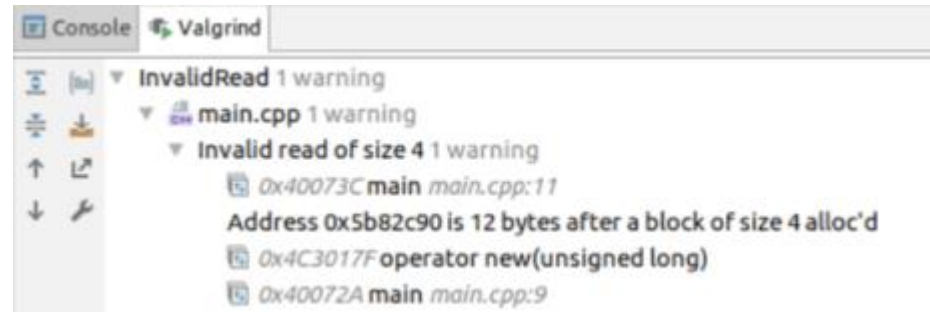
# Инструменты отладки ПО: Динамический анализатор кода

Динамический анализатор:

- Анализирует работающую программу на предмет корректной работы с памятью и с ресурсами системы.

Примеры:

- Application Verifier
- Address Sanitizer
- Valgrind







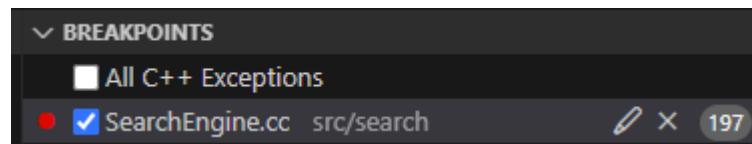
# Отладчик Visual Studio Code

- Точки остановки (Breakpoints)
- Выполнение программы
- Стек вызовов (Call Stack)
- Просмотр переменных (Watch window)
- Debug и Release конфигурации

# Отладчик Visual Studio Code: Точки останова: типы

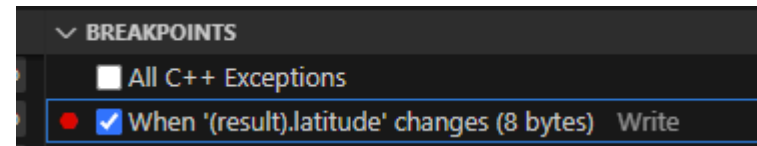
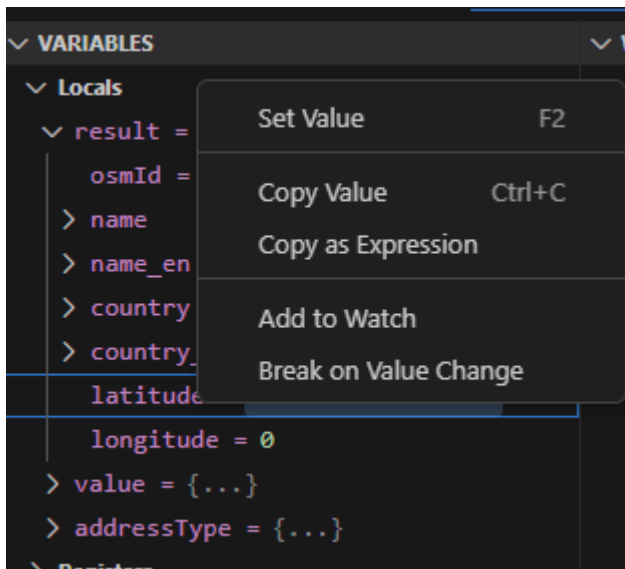
Simple Breakpoint – остановка программы при исполнении конкретной строки кода.

```
193  GeoProtoPlaces·SearchEngine::FindCitiesByPosition
194  {
195      ··· // First, find ids of "relation" entities by a
196      ··· const overpass::OsmIds relationIds = overpass:
197      ··· return findCities(relationIds, nominatim::Matc
198  }
```



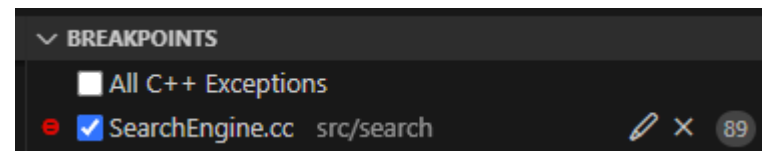
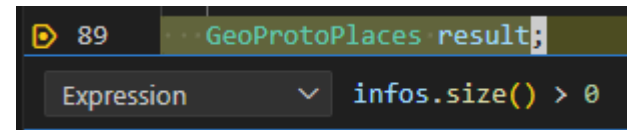
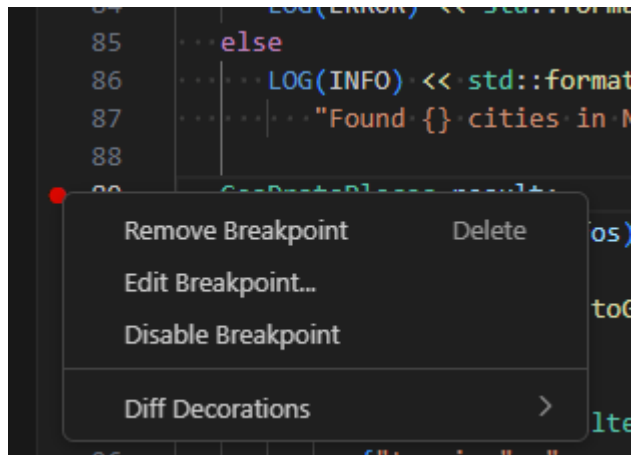
# Отладчик Visual Studio Code: Точки останова: типы

Data Breakpoint – остановка программы при изменении памяти по указанному адресу. Обычно ограничено размером указателя (4 или 8 байт).



# Отладчик Visual Studio Code: Точки останова: настройка

Expression: остановка программы только при выполнении условия  
Синтаксис условий – упрощенный C++.





# Отладчик Visual Studio Code: Точки останова: настройка

Expression: остановка программы только при выполнении условия  
Синтаксис условий – упрощенный C++.

```
190 if (json::GetString(json::Get(item, "address_type")) == type)
191 {
192     auto newObject = jsonToObject<RelationInfo>(item, type);
193     bool needAdd = true;
```

Expression    newObject.name\_en.compare("Yerevan") == 0

# Отладчик Visual Studio Code: Точки останова: настройка

Log Message – напечатать сообщение в окно Debug Console

```
36 if (itID != itBegin)
37     request += ",";
38     request += "R" + std::to_string(*itID);
39 }
40 return request + (language ? std::format("&accept-1
```

```
36 if (itID != itBegin)
37     request += ",";
38     request += "R" + std::to_string(*itID);
39 }
40 return request + (language ? std::format("&accept-1
```

Log Message    Request string = "{request.c\_str()}"

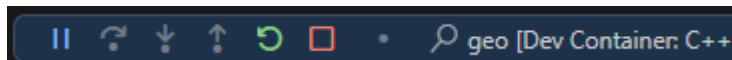
```
Thread 17 "nexting_thread" hit Breakpoint 2, (anonymous namespace)::formatRelationLookupR
equest (itBegin=..., itEnd=..., language=0x15f1b1e "en-US") at /workspaces/geo/src/searc
h/NominatimApiUtils.cc:36
36     if (itID != itBegin)
Adding OsmId 364066 to request string

Thread 17 "nexting_thread" hit Breakpoint 2, (anonymous namespace)::formatRelationLookupR
equest (itBegin=..., itEnd=..., language=0x15f1b1e "en-US") at /workspaces/geo/src/searc
h/NominatimApiUtils.cc:36
36     if (itID != itBegin)
Adding OsmId 364087 to request string

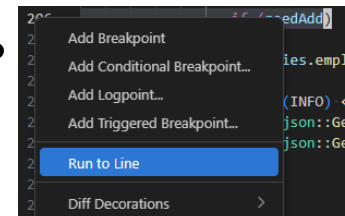
Thread 17 "nexting_thread" hit Breakpoint 2, (anonymous namespace)::formatRelationLookupR
equest (itBegin=..., itEnd=..., language=0x15f1b1e "en-US") at /workspaces/geo/src/searc
h/NominatimApiUtils.cc:36
36     if (itID != itBegin)
Adding OsmId 13404218 to request string

Thread 17 "nexting_thread" hit Breakpoint 4, (anonymous namespace)::formatRelationLookupR
equest (itBegin=..., itEnd=..., language=0x15f1b1e "en-US") at /workspaces/geo/src/searc
h/NominatimApiUtils.cc:40
40     return request + (language ? std::format("&accept-language={}", language) :
""");
Request string = "0x7fb4dc01f120 "format=json&osm_ids=R364066,R364087,R13404218"
```

# Отладчик Visual Studio Code: Выполнение программы



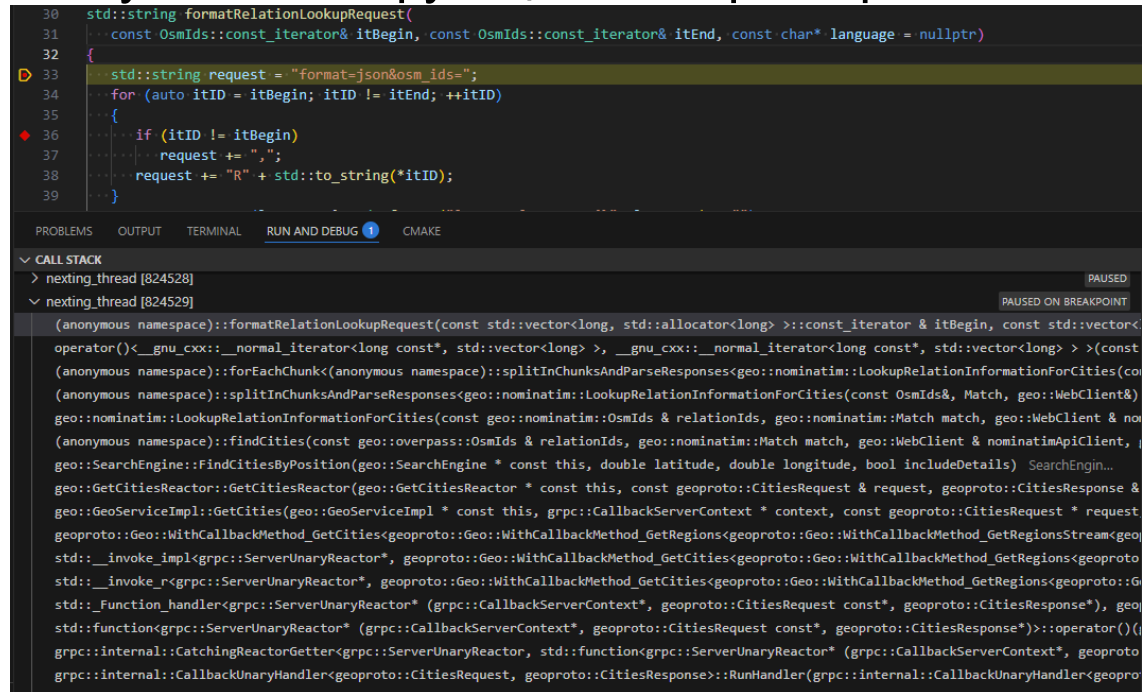
- Pause (F6) – приостановить выполнение программы
- Continue (F5) – продолжить работу программы до следующей остановки
- Step Over (F10) – выполнить следующую строку программы целиком
- Step Into (F11) – выполнить следующую операцию, ничего не пропуская
- Step Out (Shift+F11) – выполнить текущую функцию до конца
- Restart (Ctrl+Shift+F5) – перезапустить выполнение программы
- Stop (Shift+F5) – остановить все потоки программы прямо сейчас
- Run to Line (контекстное меню) – выполнить программу до выбранной строчки





# Отладчик Visual Studio Code: Стек вызовов

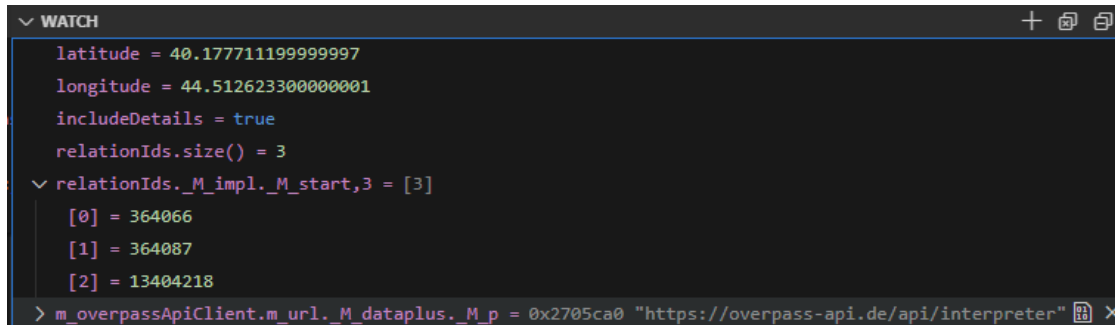
- Показывает полную цепочку вызовов функций, которая привела к текущему состоянию программы
- Сверху – текущая точка исполнения программы





# Отладчик Visual Studio Code: Просмотр переменных

- Просмотр значений глобальных и локальных переменных
- Просмотр значений простых выражений
- Изменение значений переменных

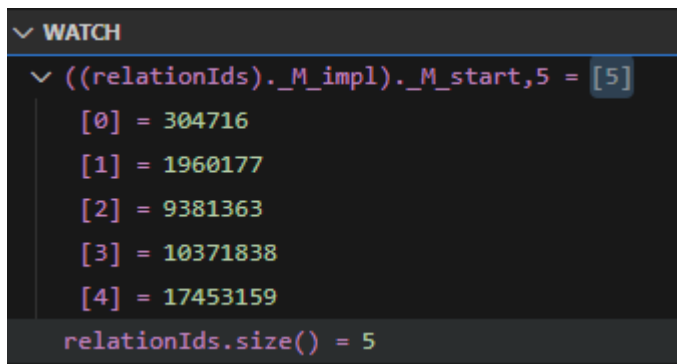
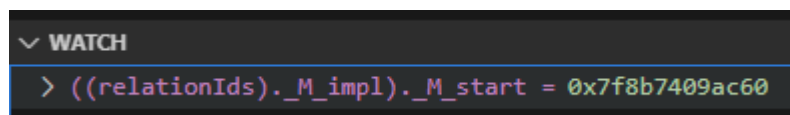
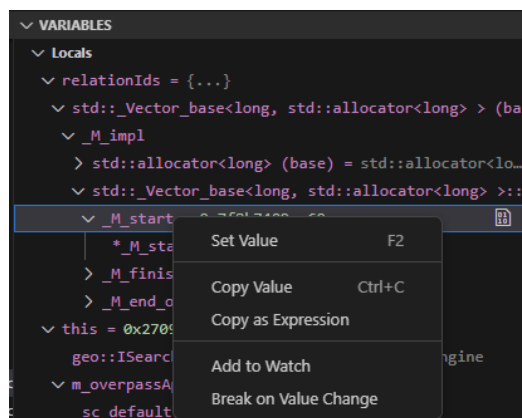


The screenshot shows the 'WATCH' window in Visual Studio Code. It displays the following variables and their values:

- `latitude` = 40.177711199999997
- `longitude` = 44.512623300000001
- `includeDetails` = `true`
- `relationIds.size()` = 3
- Expanded view of `relationIds._M_impl._M_start,3` = [3]:
  - [0] = 364066
  - [1] = 364087
  - [2] = 13404218
- Bottom line: `> m_overpassApiClient.m_url._M_dataplus._M_p = 0x2705ca0 "https://overpass-api.de/api/interpreter"`

# Отладчик Visual Studio Code: Просмотр переменных

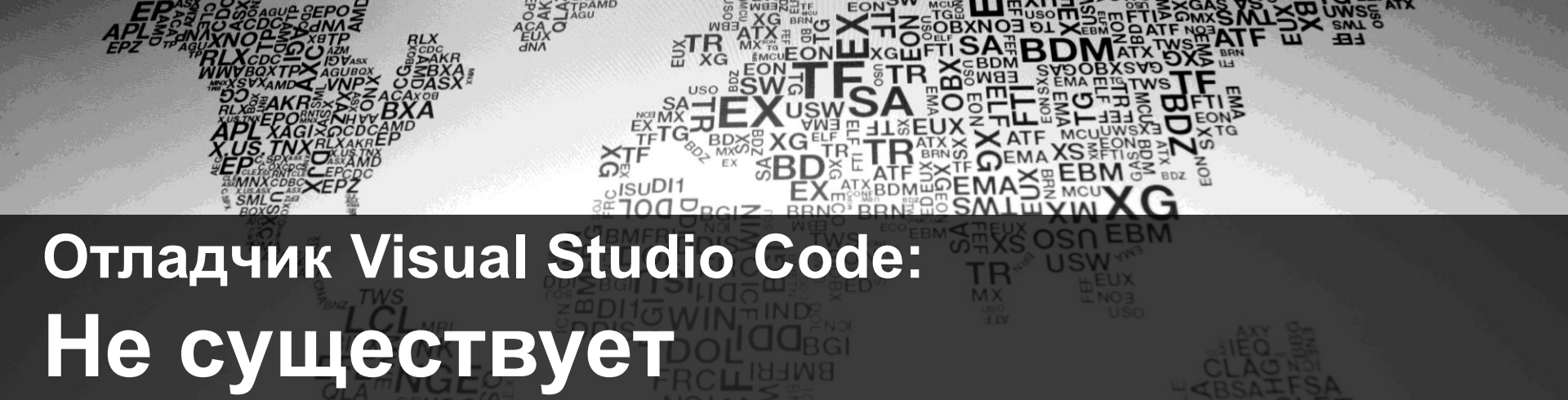
- Форматирование типов из STL





# Отладчик Visual Studio Code: Debug и Release

- В Debug конфигурации код не оптимизируется и содержит избыточную информацию для максимального удобства отладки.
- В Release конфигурации код оптимизирован, а отладка при этом может быть затруднена или вовсе невозможна.
- DevContainer Geo-сервиса по умолчанию собирает Debug сборку



# Отладчик Visual Studio Code: Не существует

## 1. Visual Studio Code не содержит собственного отладчика

- IDE лишь предоставляет интерфейс и общее управление отладкой.

## 2. Интеграция реализована через расширения и Debug Adapter Protocol (DAP)

- VSCode не обращается к отладчику напрямую, а использует унифицированный протокол общения с debug adapter'ами.

## 3. Для C/C++ отладка чаще всего выполняется с помощью GDB или LLDB

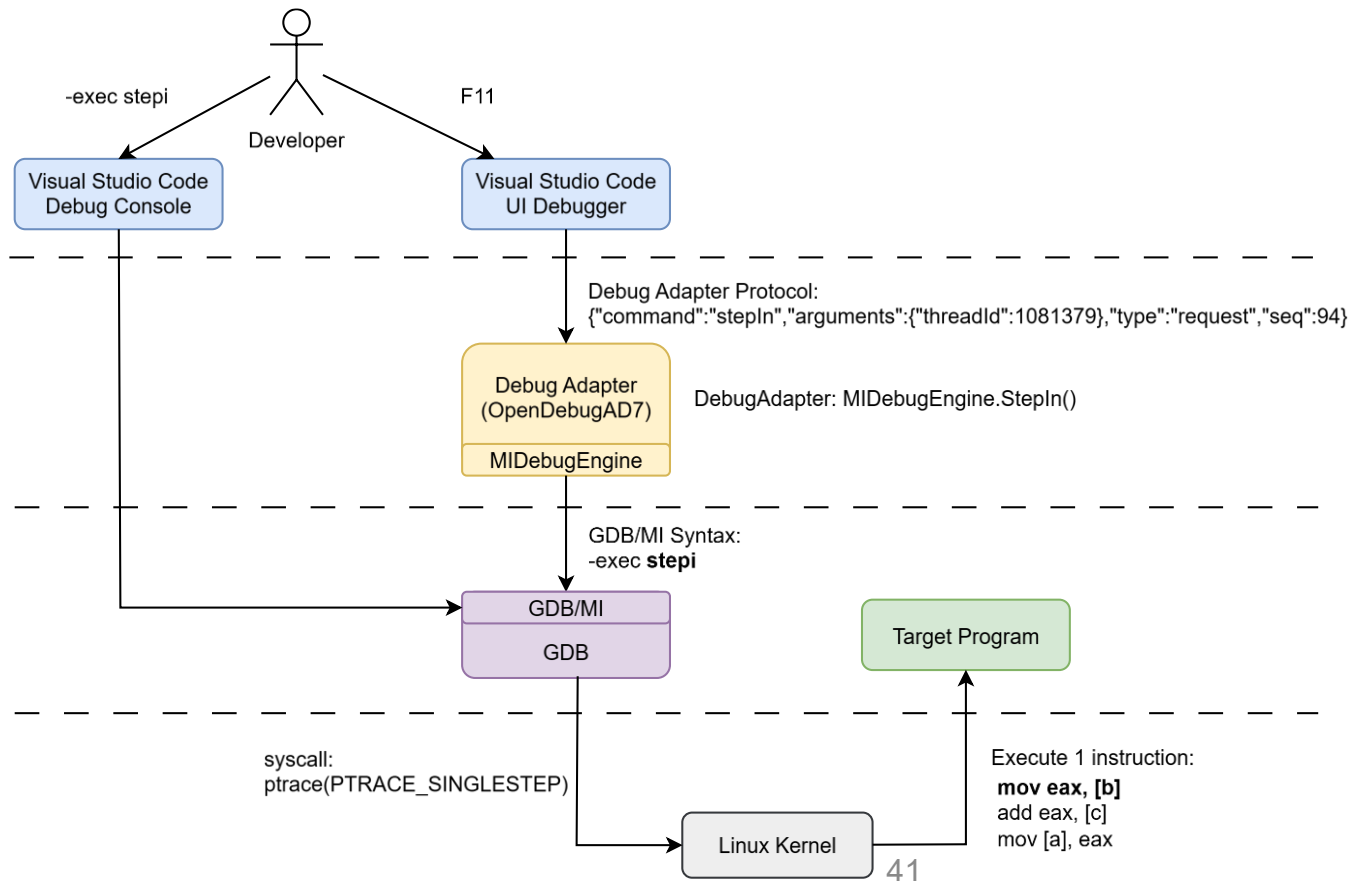
- Выбор зависит от платформы, по умолчанию LLDB используется на macOS.

## 4. Расширение C/C++ запускает debug adapter (OpenDebugAD7), который управляет GDB через GDB/MI

- VSCode взаимодействует с adapter'ом по DAP, а adapter — с GDB по протоколу GDB Machine Interface.



# Отладчик Visual Studio Code: Не существует





# Креш дампы (core dumps)


- Назначение
- Создание
- Анализ (с отладочными символами и без)

# Креш дампы (core dumps): Назначение

Содержат информацию о состоянии программы в определённый момент времени:

- Полный/частичный снимок памяти
- Потоки приложения
  - стек вызовов
  - значения регистров процессора
  - значения локальных переменных
- Информация об ошибке или исключении





# Креш дампы (core dumps): Создание

- В Linux автоматическое создание core-дампов при сбоях можно разрешить или запретить через системные настройки.
- Дампы могут создаваться:
  - Автоматически при сбоях (если это разрешено в системе):  
\$ ulimit -c unlimited  
\$ sudo sysctl -w kernel.core\_pattern=core
  - Вручную — с помощью специальных инструментов, например gcore, для сохранения состояния "живого" процесса:  
\$ gcore -o ./core {PID}
- В некоторых дистрибутивах используется система systemd-coredump, которая собирает и хранит дампы централизованно.



# Креш дампы (core dumps): Анализ

- Анализ core-дампа похож на отладку программы в отладчике
- Основной инструмент — отладчик gdb, в котором можно:
  - Посмотреть стек вызовов (backtrace)
  - Исследовать переменные и память
  - Перейти к строкам исходного кода
- Для корректного анализа нужны:
  - Исполняемый файл
  - Символы отладки — они позволяют отображать имена функций, переменных и исходные строки:
    - В процессе разработки они обычно **встроены** в бинарник (через флаг -g)
    - В релизных сборках символы могут быть **вынесены** в отдельные .debug файлы, которые могут храниться на специальном символ-сервере
  - Желательно — исходный код соответствующей версии

# Креш дампы (core dumps): Анализ (gdb)

```
$ g++ -g -O0 main.cc -o crash_demo
$ ./crash_demo
$ gdb ./crash_demo ./core
```

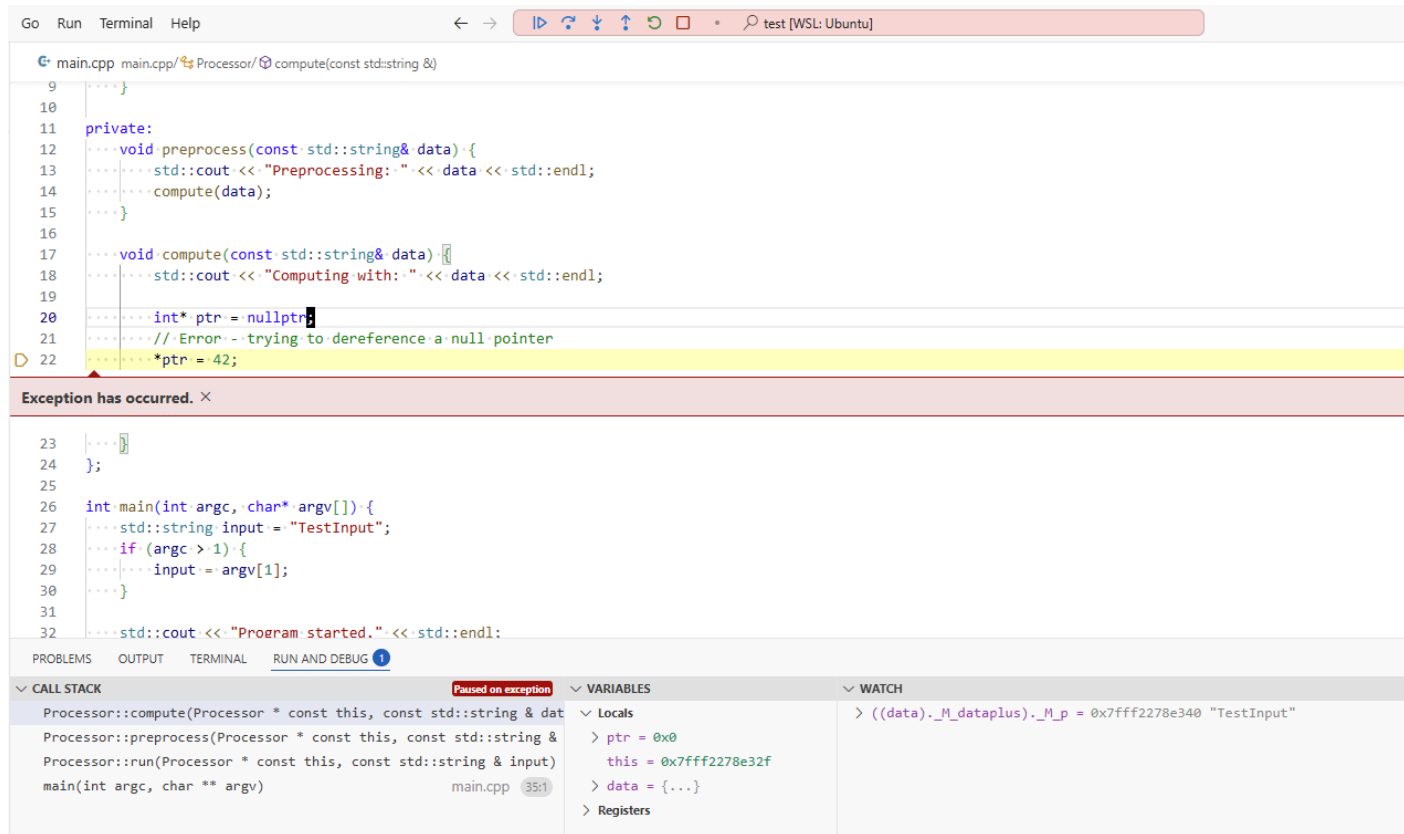
```
(gdb) bt
#0  Processor::compute (this=0x7fffffffddc70, data="TestInput") at main.cc:24
#1  0x00005555555551af in Processor::preprocess (this=0x7fffffffddc70, data="TestInput") at main.cc:17
#2  0x0000555555555141 in Processor::run (this=0x7fffffffddc70, input="TestInput") at main.cc:11
#3  0x00005555555550d9 in main (argc=1, argv=0x7fffffffe098) at main.cc:33
```

---

```
$ g++ main.cc -o crash_demo
$ ./crash_demo
$ gdb ./crash_demo ./core
```

```
(gdb) bt
#0  0x000055555555516a in ?? ()
#1  0x0000555555555120 in ?? ()
#2  0x00005555555550b5 in ?? ()
#3  0x0000555555555065 in ?? ()
#4  0x00007ffff7d6d90 in __libc_start_call_main (main=0x555555555040, argc=1, argv=0x7fffffffe098) at
../sysdeps/nptl/libc_start_call_main.h:58
```

# Креш дампы (core dumps): Анализ (VSCode)



The screenshot displays the Visual Studio Code interface with a C++ project. The editor shows a file named `main.cpp` with the following code:

```
9  ...  
10  
11 private:  
12 void preprocess(const std::string& data){  
13     std::cout << "Preprocessing: " << data << std::endl;  
14     compute(data);  
15 }  
16  
17 void compute(const std::string& data){  
18     std::cout << "Computing with: " << data << std::endl;  
19  
20     int* ptr = nullptr;  
21     // Error - trying to dereference a null pointer  
22     *ptr = 42;  
23  
24 }  
25  
26 int main(int argc, char* argv[]){  
27     std::string input = "TestInput";  
28     if (argc > 1){  
29         input = argv[1];  
30     }  
31  
32     std::cout << "Program started." << std::endl;
```

An exception has occurred, as indicated by the red bar and the message "Exception has occurred. X". The program is paused at line 22, where it attempts to dereference a null pointer.

The bottom of the interface shows the "RUN AND DEBUG" panel with the following sections:

- CALL STACK:** Shows the call stack with the following frames:
  - Processor::compute(Processor \* const this, const std::string & data)
  - Processor::preprocess(Processor \* const this, const std::string & data)
  - Processor::run(Processor \* const this, const std::string & input)
  - main(int argc, char \*\* argv) main.cpp 35:1
- VARIABLES:** Shows the current state of variables:
  - Locals:
    - `ptr = 0x0`
    - `this = 0x7fff2278e32f`
    - `data = {...}`
  - Registers
- WATCH:** Shows the value of the expression `((data)._M_dataplus)._M_p = 0x7fff2278e340 "TestInput"`.



# Типовые ошибки

- Общие закономерности
- Фатальные ошибки времени исполнения
  - Segmentation Fault
  - Heap Corruption
  - Stack Overflow
- Другие ошибки управления памятью
  - Memory Leaks





# Типовые ошибки: Общие закономерности

- Ошибку исправить раньше легче чем позже.
- Ошибка чаще всего в вашем коде.
- Изредка, ошибка бывает в сторонней библиотеке или фреймворке.
- Чем популярнее сторонняя библиотека, там меньше в ней ошибок.
- Ошибки *случаются* в компиляторе, в ОС и в драйверах.



# Типовые ошибки: Segmentation Fault

Segmentation Fault – сигнал ОС процессу о доступе к запрещенной памяти.

## 1. “Мусорный” указатель:

```
int* ptr;           // неинициализирован (мусор в памяти)
*ptr = 10;          // попытка записи по неизвестному адресу
```

## 2. Запись в read-only память:

```
char* str = "hello"; // строка в read-only сегменте
str[0] = 'H';         // попытка записи
```

## 3. Выход за пределы массива (buffer overrun):

```
int arr[3] = {1, 2, 3};
arr[10] = 42; // выход за границы массива → undefined behavior
```



# Типовые ошибки: Heap Corruption

Heap Corruption – ошибка на логическом уровне, которая может иметь разные последствия.

## 1. Двойное освобождение (double delete):

```
int* data = new int(42);  
delete data;  
delete data;
```

## 2. Переполнение области памяти в куче (heap overflow):

```
char* buf = new char[8];  
strcpy(buf, "This is too long");
```

## 3. Использование освобожденной памяти (use-after-free):

```
int* data = new int(42);  
delete data;  
*data = 99;
```



# Типовые ошибки: Коварный сценарий use-after-free

```
#include <iostream>
#include <thread>

struct A { int x = 1; };
struct B { int y = 2; };

A* shared;

void thread1() {
    shared = new A();
    delete shared; // Освобождаем память
    shared->x = 99; // Use-after-free: память могла быть переиспользована
}

void thread2() {
    B* b = new B(); // Возможно, попадаем в ту же область памяти, где был A
    std::cout << "B->y = " << b->y << std::endl; // Может напечатать "B->y = 99"
}

int main() {
    std::thread t1(thread1);
    std::thread t2(thread2);
    t1.join();
    t2.join();
}
```





# Типовые ошибки: Stack Overflow

## 1. Бесконечная рекурсия:

```
void infiniteRecursion() {  
    infiniteRecursion(); // Бесконечный вызов → переполнение стека  
}
```

## 2. Большие объекты на стеке:

```
void hugeStackAllocation() {  
    int massiveArray[10000000]; // Стек по умолчанию 1-8МБ  
}
```

## 3. Глубокая рекурсия:

```
int factorial(int n) {  
    if (n == 0) return 1;  
    return n * factorial(n - 1); // При больших n - переполнение  
}
```



# Типовые ошибки: Stack Overflow (пояснение)

Стек – область хранения текущих активных вызовов.

Максимальный размер по умолчанию: 1-8МБ

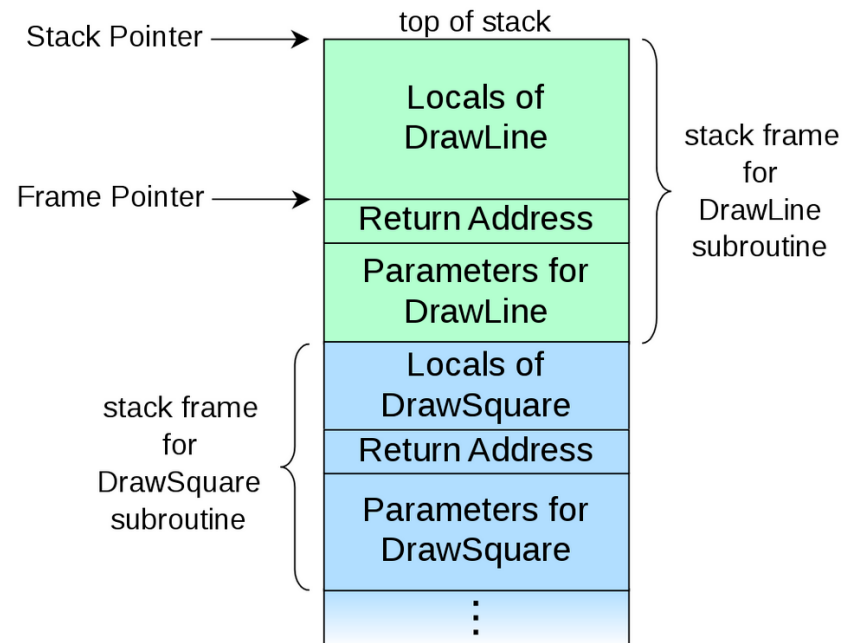
Каждый вызов функции добавляет кадр стека (stack frame), который содержит:


- Параметры функции
- Адрес возврата
- Локальные переменные функции

# Типовые ошибки: Stack Overflow (причина)

```
void DrawLine(int x1, int y1, int x2, int y2)
{
    // ... рисуем линию ...
}
```

```
void DrawSquare(int x, int y, int size)
{
    int left = x - size;
    int right = x + size;
    DrawLine(left, y, right, y);
    // ... другие линии ...
}
```





# Типовые ошибки: Memory Leaks

## 1. Забытый delete:

```
int* data = new int[100]; // выделили память
// забыли delete[] data; - утечка памяти
```

## 2. Циклические ссылки с std::shared\_ptr:

```
struct Node { std::shared_ptr<Node> next; };
auto a = std::make_shared<Node>();
auto b = std::make_shared<Node>();
a->next = b;
b->next = a; // цикл - память никогда не освободится
```

## 3. Невиртуальный деструктор в базовом классе:

```
struct Base { ~Base() {} }; // деструктор НЕ виртуальный
struct Derived : Base { ~Derived() { /* освобождение */ } };
```

```
Base* obj = new Derived();
delete obj; // вызовется только ~Base - утечка в Derived
```





# Q&A