|  |
| --- |
|  |
| МИНОБРНАУКИ РОССИИ |
| *Федеральное государственное бюджетное образовательное учреждение высшего образования* ***«МИРЭА – Российский технологический университет»***  **РТУ МИРЭА** |

Институт Информационных технологий (ИТ)

Кафедра Математического обеспечения и стандартизации информационных технологий (МОСИТ)

|  |
| --- |
| **ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ № 5** |
| **по дисциплине** |
| **«Тестирование и верификация программного обеспечения»** |

|  |  |
| --- | --- |
| Выполнили студенты группы ИКБО-04-22 | Егоров Л.А. |
| Принял ассистент | Петрова А.А. |

|  |  |  |
| --- | --- | --- |
| Практическая работа выполнена | «\_\_»\_\_\_\_\_\_\_202\_\_г. | *(подпись студента)* |
|  |  |  |
| «Зачтено» | «\_\_»\_\_\_\_\_\_\_202\_\_ г. | *(подпись руководителя)* |

Москва 2024

# Содержание

[Содержание 2](#_Toc1)

[1 Статический анализ 3](#_Toc2)

[1.1 Анализ кода на языке Python 3](#_Toc3)

[1.1.1 Добавление ошибок в код 4](#_Toc4)

[1.2 Анализ кода на языке C 6](#_Toc5)

[1.2.1 Внесённые ошибки 7](#_Toc6)

[1.3 Вывод 10](#_Toc7)

[2 Динамический анализ 12](#_Toc8)

[2.1 Анализ кода на языке Python 12](#_Toc9)

[2.1.1 Внесённые ошибки 13](#_Toc10)

[2.2 Анализ кода на языке C 14](#_Toc11)

[2.2.1 Внесённые ошибки 16](#_Toc12)

[2.3 Вывод 20](#_Toc13)

[Вывод 22](#_Toc14)

[Приложение А 23](#_Toc15)

Про

# 1 Статический анализ

Статический анализ кода — это процесс анализа исходного кода программы без его исполнения, направленный на поиск ошибок, уязвимостей, нарушений стандартов и других потенциальных проблем. Он проводится с использованием специальных инструментов (статических анализаторов), которые могут анализировать код на разных уровнях: от синтаксического до логического.

## 1.1 Анализ кода на языке Python

Для анализа был использован код электромагнитного алгоритма, который представлен в Листинге А.1.

Для анализа кода на языке Python использовалось три анализатора – MyPy, Pylint, flake8.

**MyPy** — это статический анализатор кода для Python, который проверяет типы данных в программах, написанных на Python. Он позволяет разработчикам использовать аннотации типов, чтобы явно указывать ожидаемые типы переменных, аргументов функций и возвращаемых значений. MyPy анализирует код и выявляет потенциальные ошибки, связанные с несоответствием типов, что помогает улучшить качество кода и предотвратить ошибки на этапе разработки.. Результат анализа этой программой представлен на Рисунке 1.1.1.



Рисунок 1.1.1 – Анализ программой MyPy

**Pylint** — это статический анализатор кода для Python, который проверяет качество и стиль кода на соответствие стандартам PEP 8 и другим рекомендациям по написанию чистого и поддерживаемого кода. Pylint анализирует исходный код и генерирует отчеты о найденных ошибках, потенциальных проблемах, нарушениях стиля и других аспектах качества кода. Результат анализа этой программой представлен на Рисунке 1.1.2.

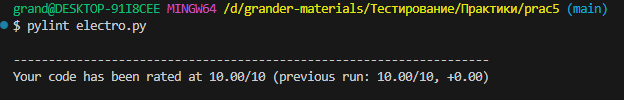


Рисунок 1.1.2 – Анализ программой pylint

**Flake8** — это инструмент статического анализа кода для Python, который объединяет в себе несколько других инструментов для проверки качества и стиля кода. Flake8 включает в себя проверки на соответствие стандартам PEP 8, поиск синтаксических ошибок и потенциальных проблем в коде. Он объединяет функциональность трех основных компонентов: PyFlakes, pycodestyle (ранее известный как pep8) и mccabe. Результат анализа этой программой представлен на Рисунке 1.1.3.



Рисунок 1.1.3 – Анализ программой flake8

### 1.1.1 Добавление ошибок в код

В код исходной программы добавлено 5 ошибок:

1. Из функций были убраны строки документации (Рисунок 1.1.4).
2. Одна строка была сделана большой длины (больше 79 символов) (Рисунок 1.1.5).
3. Добавлена неиспользуемая переменная (Рисунок 1.1.6).
4. Допущена ошибка в типизации нескольких функций (Рисунок 1.1.7).
5. Добавлены лишние отступы между функциями (Рисунок 1.1.8).

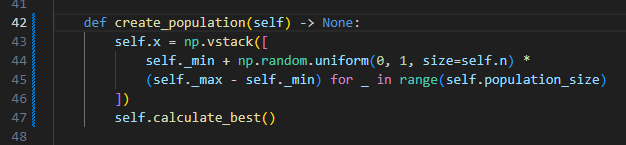


Рисунок 1.1.4 – Отсутствие документации в функции

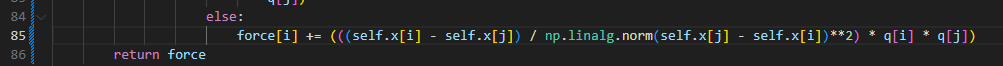


Рисунок 1.1.5 – Строка длиной больше 79 символов

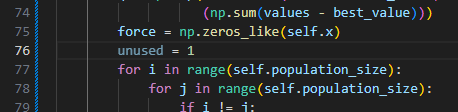


Рисунок 1.1.6 – Добавление неиспользуемой переменной

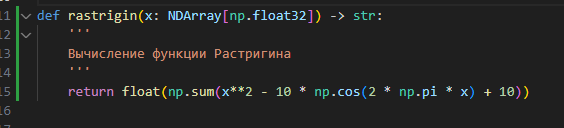


Рисунок 1.1.7 – Ошибка в типизации результата функции

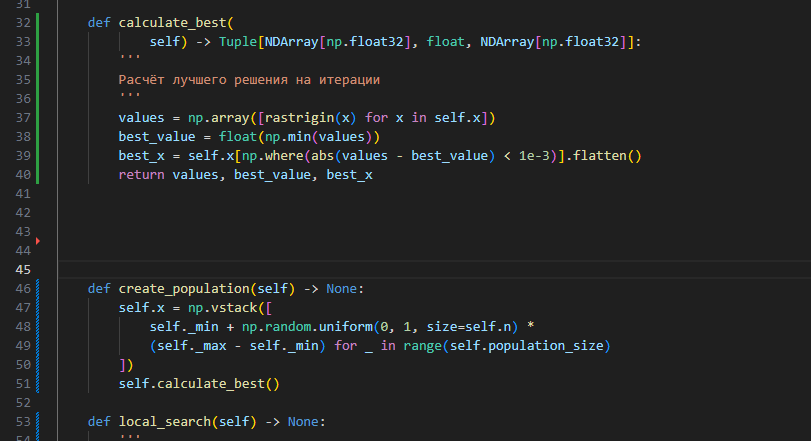


Рисунок 1.1.8 – Добавление лишних отступов

Результаты анализа представлены на Рисунках 1.1.9 - 1.1.11.

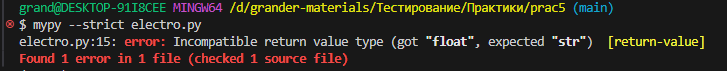


Рисунок 1.1.1.1 – Анализ программой mypy

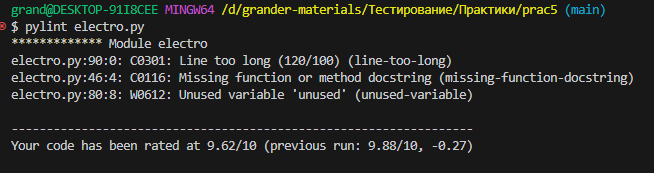


Рисунок 1.1.1.2 – Анализ программой pylint

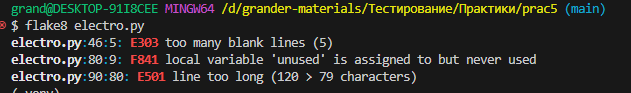


Рисунок 1.1.1.3 – Анализ программой flake8

В результате программа mypy обнаружила ошибку, связанную с типизацией. Pylint больше указала на стилистические ошибки, а flake8 указала все ошибки, которые допущены в оформлении по PEP8.

## 1.2 Анализ кода на языке C

Для анализа взята программа на языке C, выполняющая операцию добавления строки в матрицу. Её код представлен в Листинге А.2.

Для анализа кода используются статические анализаторы cppcheck, clang-tidy и gcc analyzer.

**Cppcheck** — это легковесный статический анализатор кода, который выполняет анализ исходного кода на предмет ошибок, связанных с безопасностью, производительностью и другими аспектами качества кода. Он поддерживает языки C и C++ и предназначен для выявления потенциальных проблем, которые могут быть пропущены компилятором или другими инструментами. Результат анализа представлен на Рисунке 1.2.1.

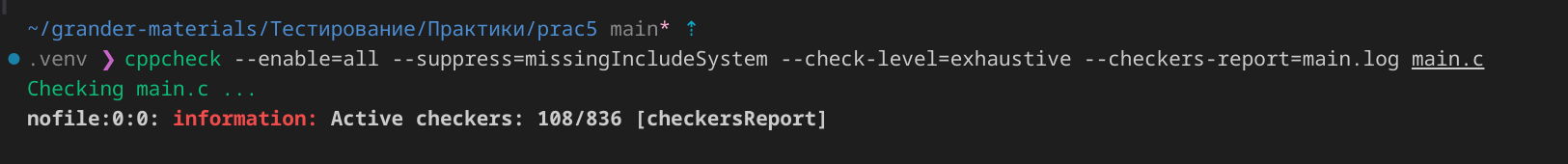


Рисунок 1.2.1 – Результат анализа cppcheck

**Clang-Tidy** — это инструмент статического анализа кода, встроенный в компилятор Clang, который выполняет анализ исходного кода на предмет ошибок, нарушений стиля, потенциальных уязвимостей и других аспектов качества кода. Clang-Tidy поддерживает языки C, C++ и Objective-C и предназначен для автоматизации процесса улучшения и очистки кода. Результат анализа представлен на Рисунке 1.2.2.

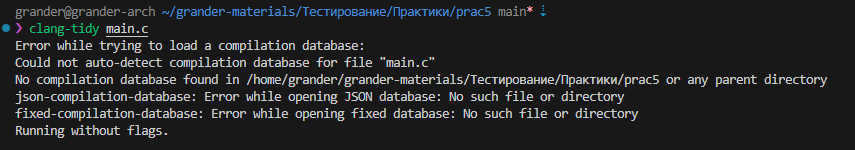


Рисунок 1.2.2 – Результат анализа clang-tidy

**GCC -fanalyzer** — это опция компилятора GCC, которая включает встроенный статический анализатор кода. Этот анализатор выполняет глубокий анализ исходного кода на предмет ошибок, потенциальных уязвимостей и других аспектов качества кода. Он работает на этапе компиляции и может выявлять проблемы, которые могут быть пропущены другими инструментами. Результат анализа представлен на Рисунке 1.2.3.



Рисунок 1.2.3 – Результат анализа gcc -fanalyzer

### 1.2.1 Внесённые ошибки

В код исходной программы добавлено 5 ошибок:

1. Во внутренний цикл добавлена переменная, которая перекрывает переменную внешнего цикла (Рисунок 1.2.4).
2. Для считывания данных использована небезопасная функция scanf (Рисунок 1.2.5).
3. Добавлена бесконечный цикл for (Рисунок 1.2.6).
4. Добавлено обращение к неинициализированному участку памяти (Рисунок 1.2.7).
5. Убрана очистка памяти после завершения программы (Рисунок 1.2.8).

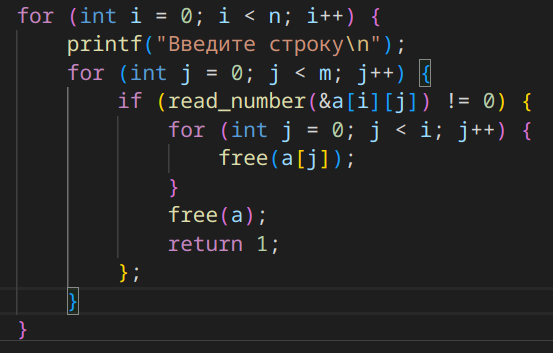


Рисунок 1.2.4 – Перекрытие переменных

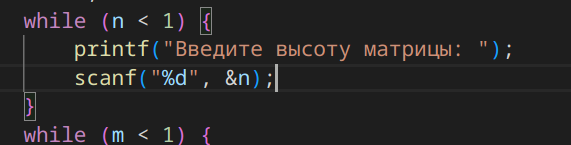


Рисунок 1.2.5 – Использование небезопасной функции scanf

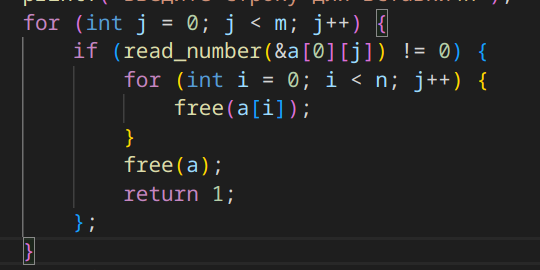


Рисунок 1.2.6 – Бесконечный цикл for

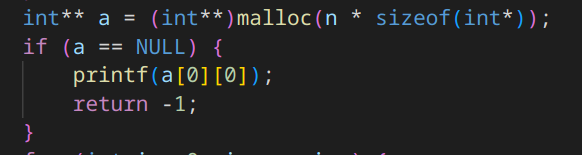


Рисунок 1.2.7 – Обращение к неинициализированному участку памяти

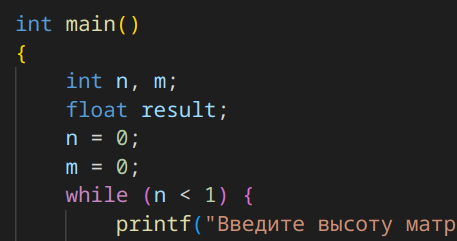


Рисунок 1.2.8 – Добавление неиспользуемой переменной

На Рисунках 1.2.9 - 1.2.11 представлены результаты анализа получившегося кода тремя анализаторами.

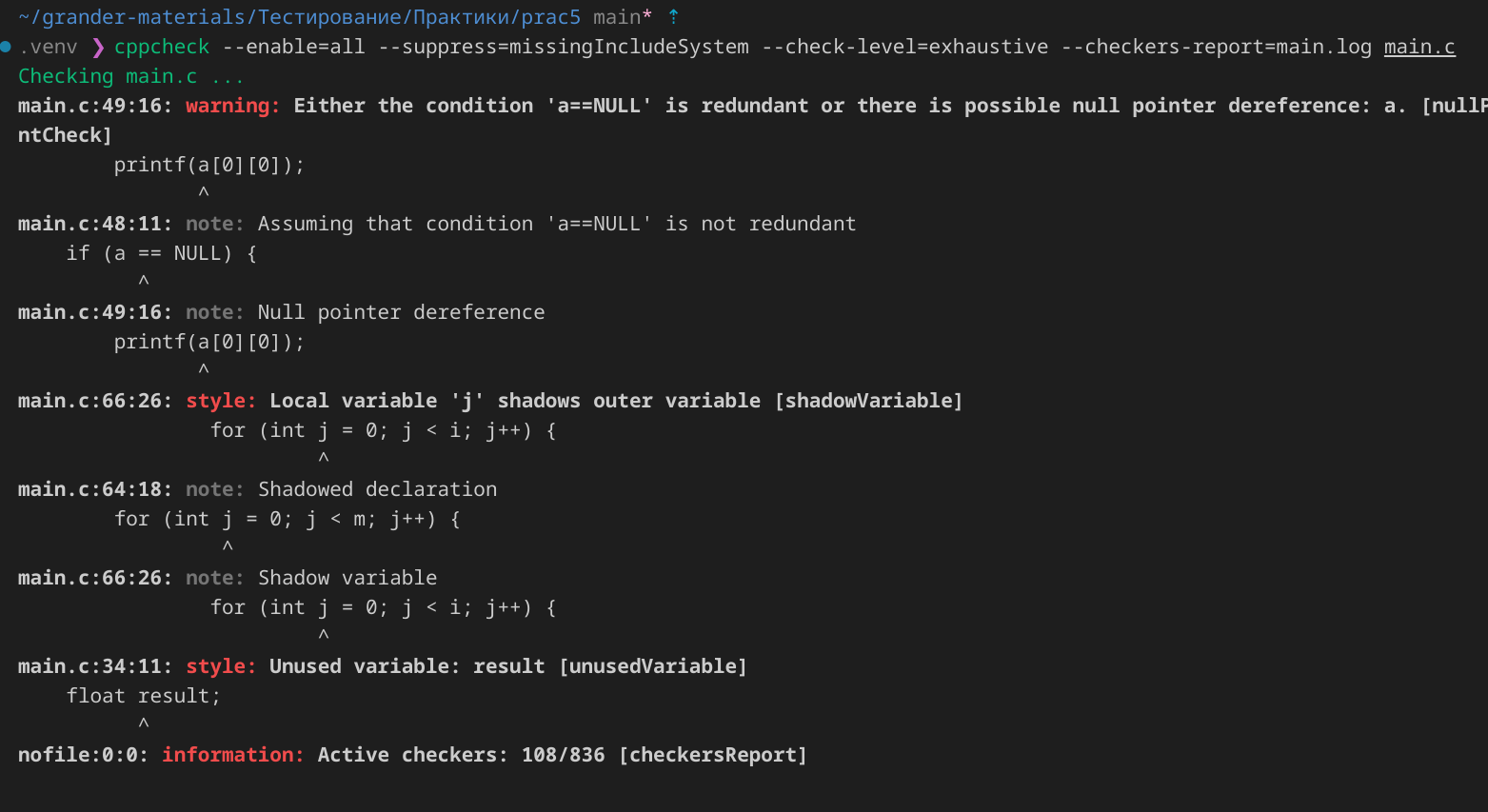


Рисунок 1.2.9 – Результат анализа cppcheck

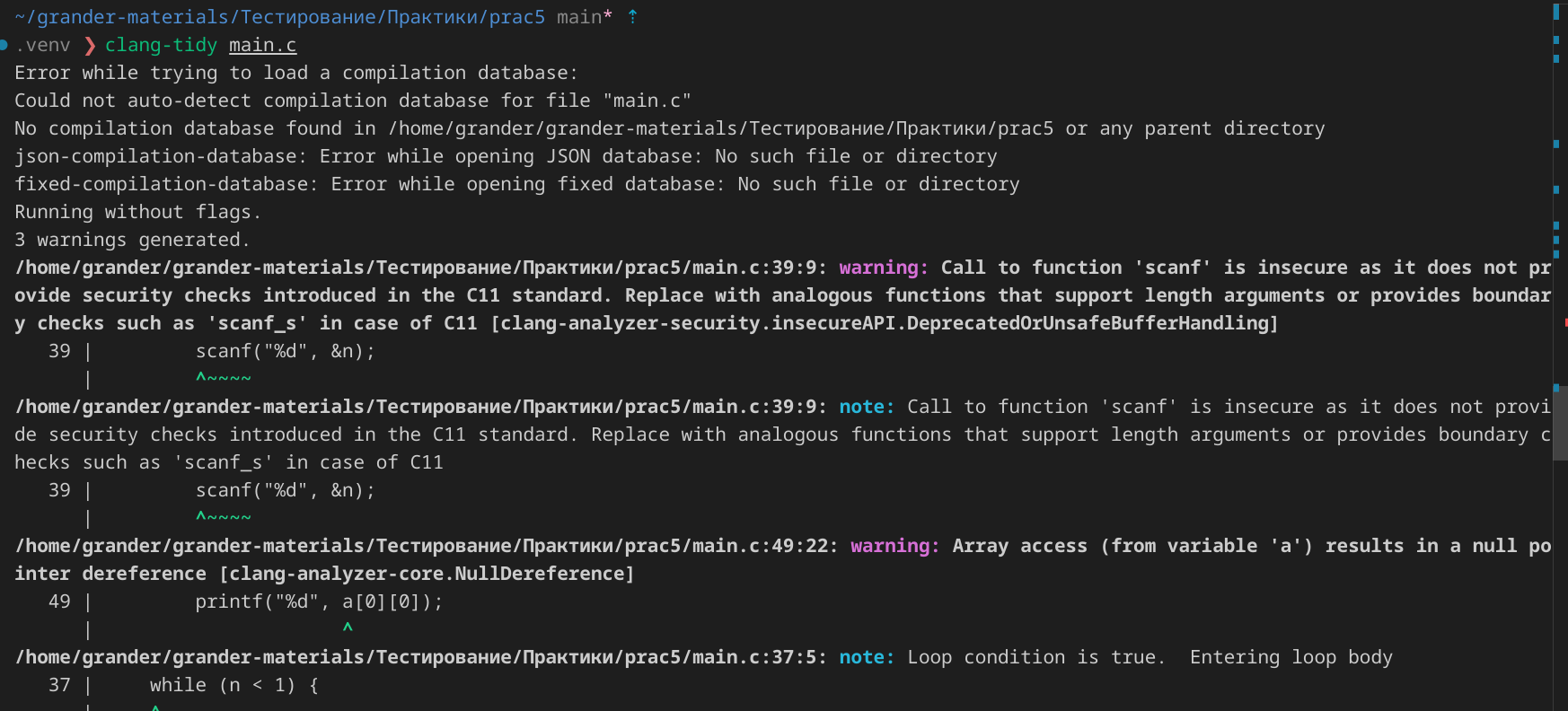


Рисунок 1.2.10 – Результат анализа clang-tidy

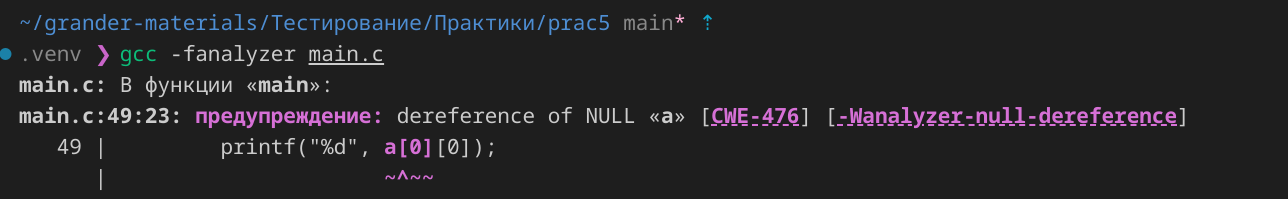


Рисунок 1.2.11 – Результат анализа gcc –fanalyzer

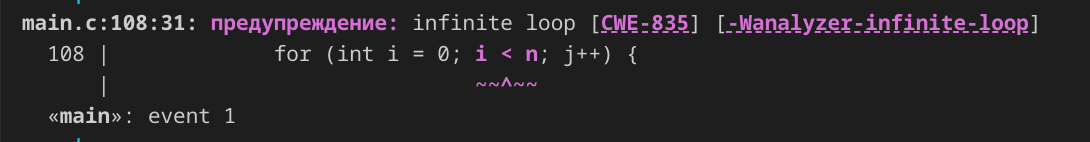


Рисунок 1.2.12 – Результат анализа gcc –fanalyzer

Все программы смогли обнаружить обращение к неинициализированному участку памяти; cppcheck также обнаружил перекрытие переменной во внутреннем цикле и объявление неиспользуемой переменной; clang-tidy указал на использование небезопасной функции scanf; gcc –fanalyzer указал на бесконечный цикл.

## 1.3 Вывод

1. Раннее обнаружение ошибок: Статический анализ позволяет обнаруживать потенциальные ошибки и несоответствия типов на ранних этапах разработки, до запуска программы. Это сокращает время на отладку и улучшает качество кода.

2. Улучшение читаемости и поддерживаемости: Аннотации типов делают код более понятным для разработчиков, особенно в больших проектах. Это облегчает навигацию по коду и его поддержку.

3. Автоматизация проверок: Инструменты статического анализа, такие как mypy, могут быть интегрированы в системы непрерывной интеграции (CI/CD), что позволяет автоматизировать процесс проверки кода на соответствие стандартам и правилам.

4. Также статический анализ позволяет обнаружить участки кода, которые очень сложны для чтения и которые нужно разбить на функции, а также ошибки в оформлении кода.

# 2 Динамический анализ

## 2.1 Анализ кода на языке Python

Для динамического анализа кода на языке Python использованы два анализатора кода – memory\_profiler и line\_profiler.

**memory\_profiler** — это инструмент для профилирования использования памяти в Python-программах. Он позволяет отслеживать, сколько памяти используется на каждой строке кода, что полезно для выявления утечек памяти, оптимизации использования памяти и понимания, какие части кода потребляют больше всего ресурсов. Результат анализа этой программой представлен на Рисунке 2.1.1.

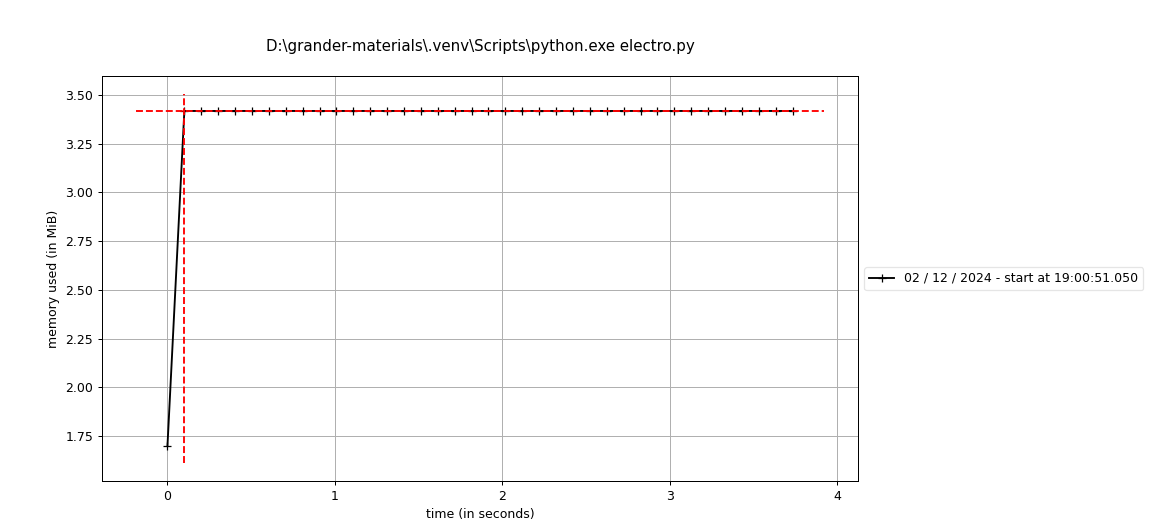


Рисунок 2.1.1 – Результат анализа memory\_profiler

**line\_profiler** — это инструмент для профилирования производительности кода на уровне строк в Python. Он позволяет измерять время выполнения каждой строки кода, что полезно для выявления узких мест в производительности и оптимизации кода. Результат анализа представлен на Рисунке 2.1.2.

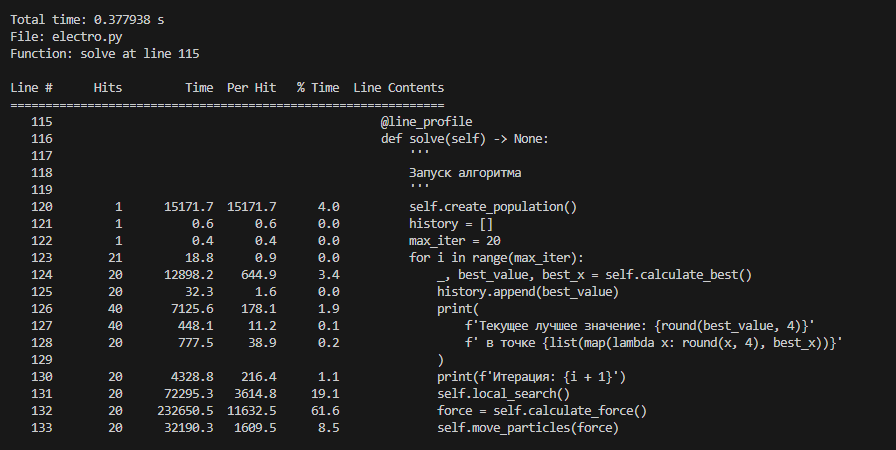


Рисунок 2.1.2 – Результат анализа line\_profiler

### 2.1.1 Внесённые ошибки

В код программы внесены три ошибки:

* в функции вычисления функции Растригина добавлена задержка на 0.3 секунды (Рисунок 2.1.3);
* в функцию вычисления лучшего результата добавлено лишнее заполнение массива (Рисунок 2.1.4);
* что-то там (Рисунок 2.1.5).

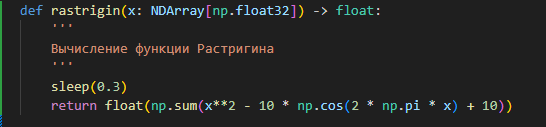


Рисунок 2.1.3 – Задержка в функции Растригина

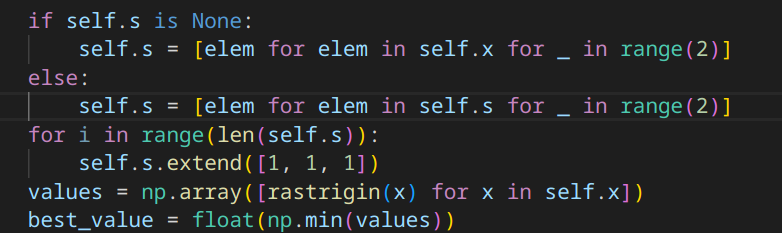


Рисунок 2.1.4 – Лишние заполнения массива

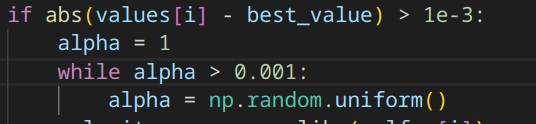


Рисунок 2.1.5 – Неэффективное вычисление случайного числа

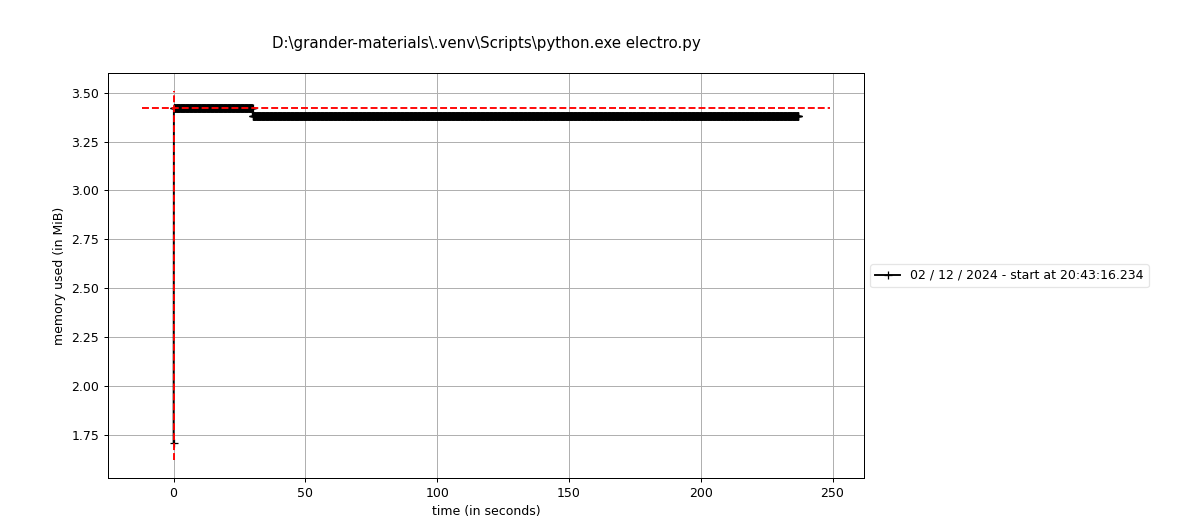


Рисунок 2.1.6 – Результат анализа memory\_profiler

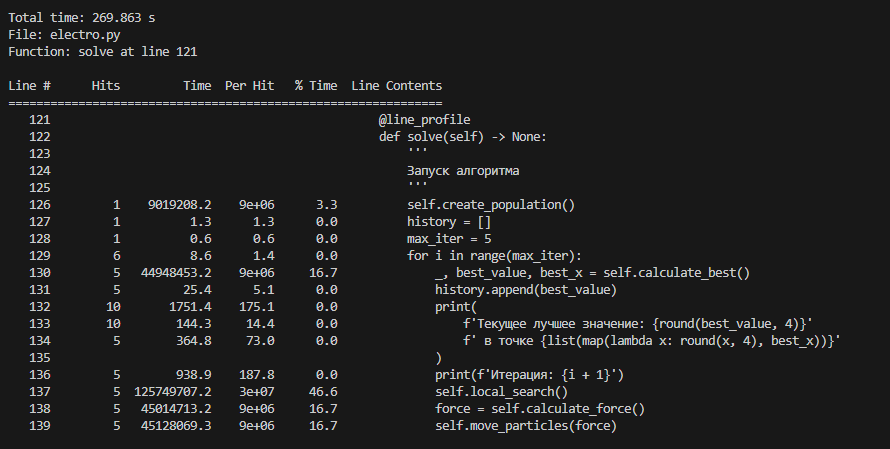


Рисунок 2.1.7 – Результат анализа line\_profiler

## 2.2 Анализ кода на языке C

Для динамического анализа кода на языке C использованы два динамических анализатора – Valgrind и DynamoRIO.

**Valgrind** — это мощный инструмент для динамического анализа программ, написанных на языках C, C++ и других языках, которые могут быть скомпилированы в исполняемые файлы для архитектуры x86. Valgrind предоставляет несколько инструментов для анализа использования памяти, поиска утечек памяти, профилирования производительности и других аспектов качества кода. Результат анализа представлен на Рисунке 2.2.1.

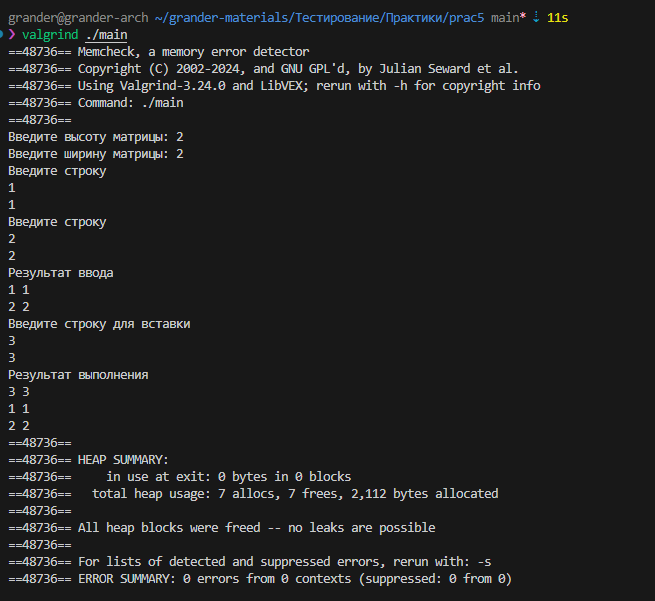


Рисунок 2.2.1 – Результат анализа Valgrind

**DynamoRIO** — это мощная платформа для динамического инструментирования программ, которая позволяет вставлять инструкции в исполняемый код во время выполнения программы. DynamoRIO поддерживает программы на языках C, C++ и других, которые могут быть скомпилированы в исполняемые файлы для архитектуры x86 и x86-64. Она предоставляет гибкий и мощный механизм для анализа и модификации кода на лету, что делает её полезной для различных задач, таких как профилирование, отладка, оптимизация и анализ безопасности. Результат анализа представлен на Рисунке 2.2.2.

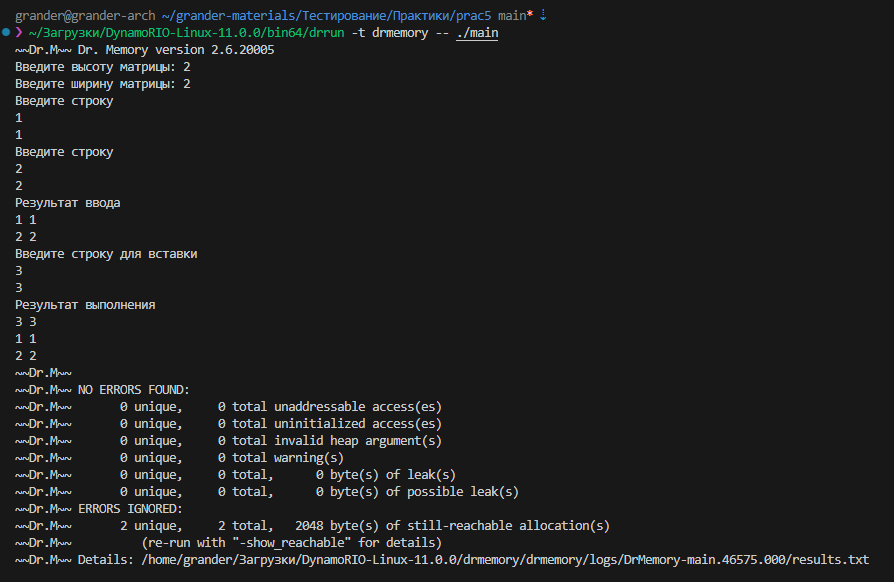


Рисунок 2.2.2 – Результат анализа DynamoRIO

### 2.2.1 Внесённые ошибки

В код программы внесены следующие ошибки:

1. Убрана очистка памяти после завершения программы (Рисунок 2.2.3).
2. Добавлено обращение к неинициализированному участку памяти (Рисунок 2.2.4).
3. Сделана очистка незанятой памяти (Рисунок 2.2.5).

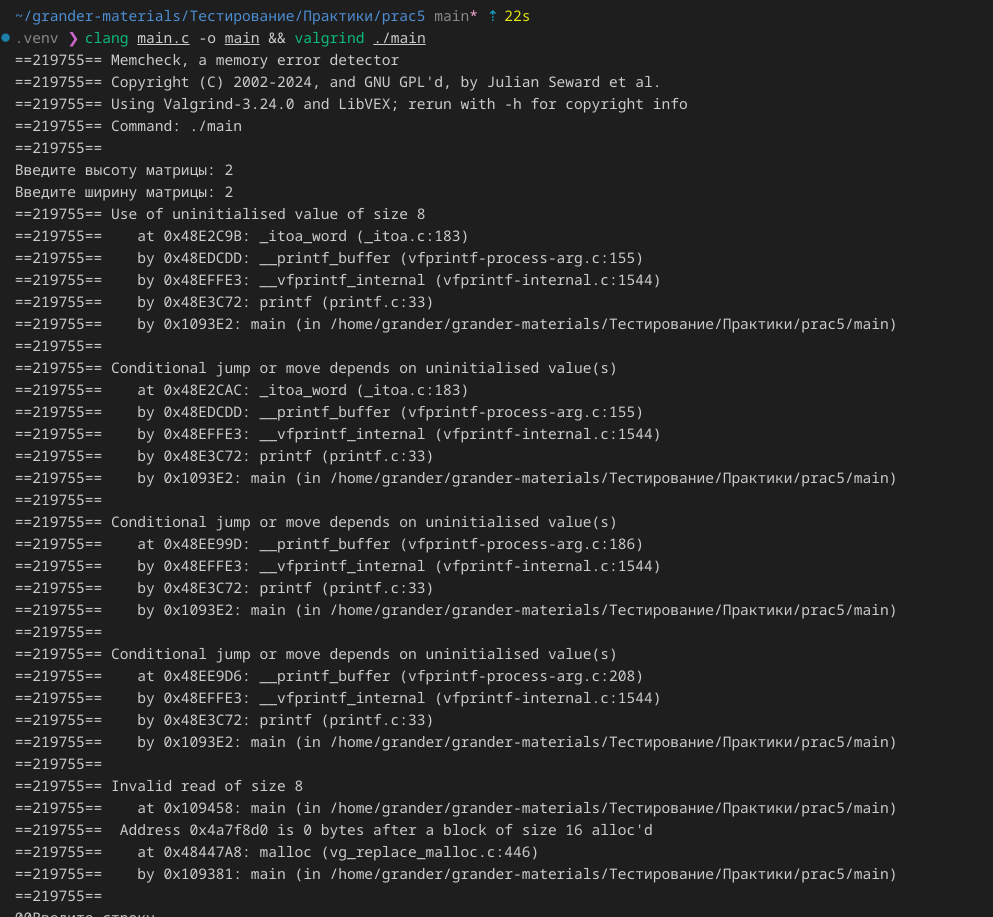


Рисунок 2.2.6 – Результат анализа Valgrind

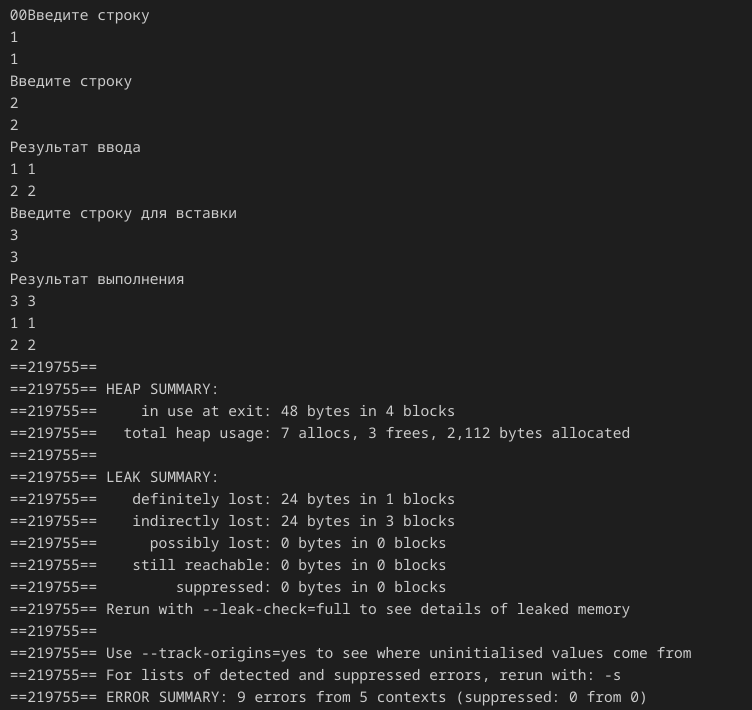


Рисунок 2.2.7 – Результат анализа Valgrind

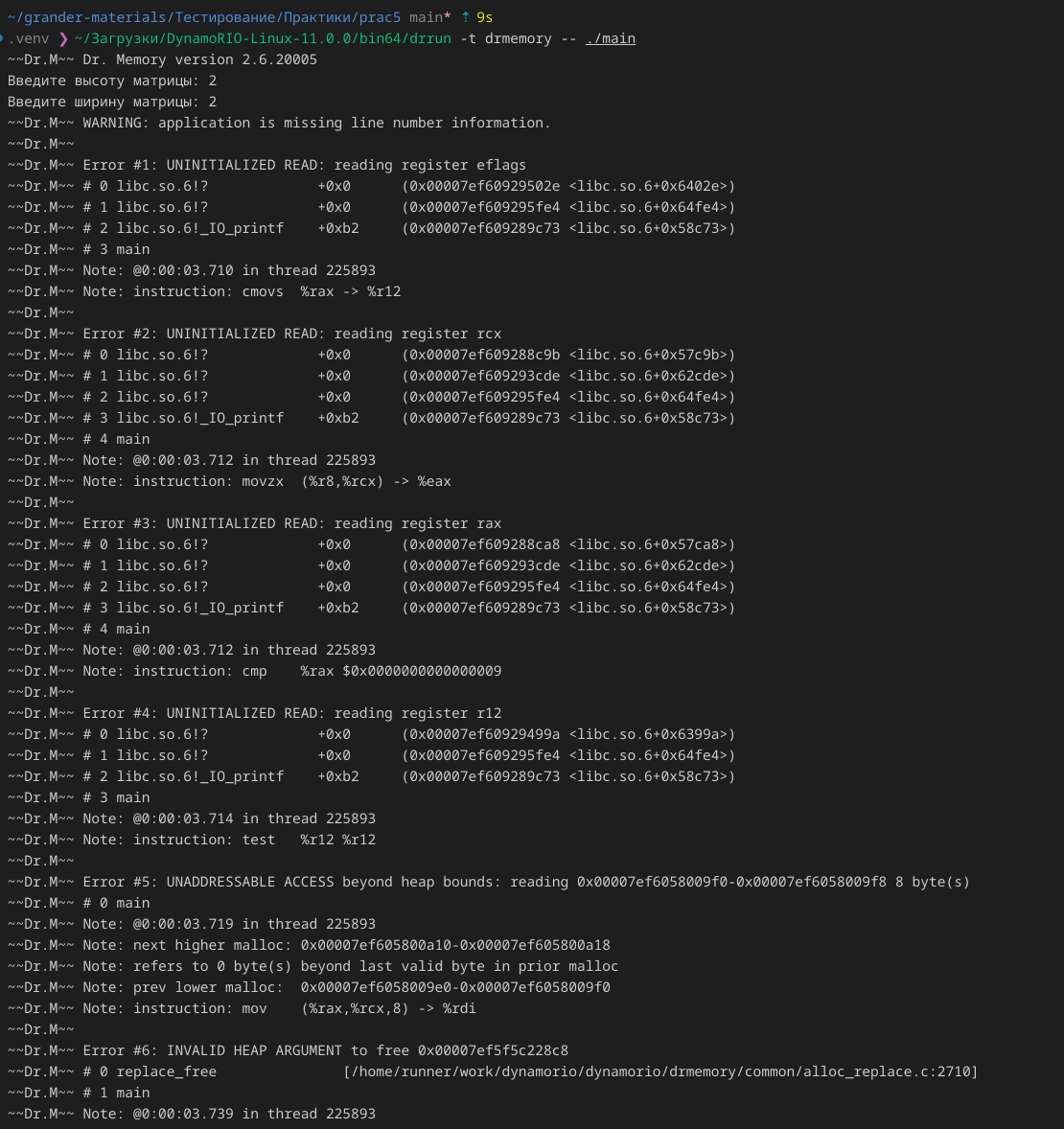


Рисунок 2.2.8 – Результат аналиха DynamoRIO

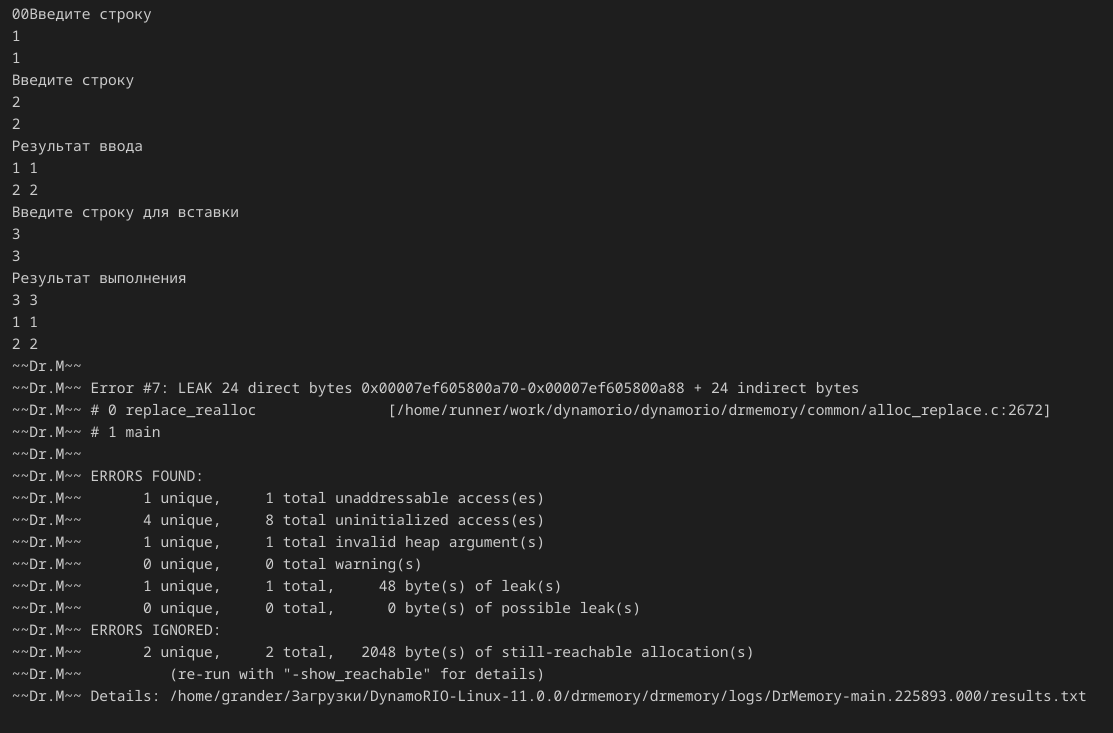


Рисунок 2.2.9 – Результат анализа DynamoRIO

## 2.3 Вывод

1. Анализ во время выполнения: Динамический анализ позволяет отслеживать поведение программы во время её выполнения, что полезно для выявления ошибок, которые невозможно обнаружить статическим анализом (например, ошибки, связанные с состоянием программы).

2. Профилирование и оптимизация: Инструменты динамического анализа, такие как профилировщики, помогают выявить узкие места в производительности и оптимизировать код для более эффективного выполнения.

3. Отладка и трассировка: Динамический анализ позволяет отслеживать поток выполнения программы, что полезно для отладки и понимания причин возникновения ошибок.

4. Анализ взаимодействия с внешними системами: Динамический анализ может помочь выявить проблемы, связанные с взаимодействием программы с внешними системами (базы данных, API и т.д.), которые сложно проверить статически.

# Вывод

Статический анализ предоставляет возможность раннего обнаружения ошибок и улучшения качества кода на этапе разработки, что сокращает время на отладку и повышает безопасность.

Динамический анализ позволяет отслеживать поведение программы во время выполнения, выявляя ошибки и проблемы, которые невозможно обнаружить статическим анализом.

Совместное использование статического и динамического анализа обеспечивает комплексный подход к обеспечению качества и надежности программного обеспечения. Статический анализ помогает предотвратить ошибки на ранних этапах, а динамический анализ позволяет убедиться, что программа работает корректно в реальных условиях.

# Приложение А

Листинг А.1 – Код файла electro.py

'''

electro.py

Реализация электромагнитного алгоритма

'''

from typing import Tuple

import numpy as np

import matplotlib.pyplot as plt

from numpy.typing import NDArray

def rastrigin(x: NDArray[np.float32]) -> float:

'''

Вычисление функции Растригина

'''

return float(np.sum(x\*\*2 - 10 \* np.cos(2 \* np.pi \* x) + 10))

class EMA:

'''

Класс, реализующий электромагнитный алгоритм

'''

def \_\_init\_\_(self, n: int):

self.n = n

self.population\_size = 10 \* n

self.local\_iter = 10

self.scale = 0.005

self.\_min = -5.12

self.\_max = -self.\_min

self.x: NDArray[np.float32]

self.values = None

def calculate\_best(

self) -> Tuple[NDArray[np.float32], float, NDArray[np.float32]]:

'''

Расчёт лучшего решения на итерации

'''

values = np.array([rastrigin(x) for x in self.x])

best\_value = float(np.min(values))

best\_x = self.x[np.where(abs(values - best\_value) < 1e-3)].flatten()

return values, best\_value, best\_x

def create\_population(self):

'''

Создание агентов в популяции

'''

self.x = np.vstack([

self.\_min + np.random.uniform(0, 1, size=self.n) \*

(self.\_max - self.\_min) for \_ in range(self.population\_size)

])

self.calculate\_best()

def local\_search(self) -> None:

'''

Реализация локального поиска

'''

search\_field = self.scale \* (self.\_max - self.\_min)

for k, particle in enumerate(self.x):

cnt = 0

while cnt < self.local\_iter:

for i in range(self.n):

sign = np.random.randint(0, 2) \* 2 - 1

y = particle.copy()

velocity = np.random.uniform()

y[i] += sign \* velocity \* search\_field

if rastrigin(y) < rastrigin(particle):

self.x[k] = y.copy()

cnt = self.local\_iter

break

cnt += 1

def calculate\_force(self) -> NDArray[np.float32]:

'''

Расчёт электромагнитной силы

'''

values, best\_value, \_ = self.calculate\_best()

q = np.exp(-self.n \* (values - best\_value) /

(np.sum(values - best\_value)))

force = np.zeros\_like(self.x)

for i in range(self.population\_size):

for j in range(self.population\_size):

if i != j:

if values[j] < values[i]:

force[i] += (

((self.x[j] - self.x[i]) /

np.linalg.norm(self.x[j] - self.x[i])\*\*2) \* q[i] \*

q[j])

else:

force[i] += (

((self.x[i] - self.x[j]) /

np.linalg.norm(self.x[j] - self.x[i])\*\*2) \* q[i] \*

q[j])

return force

def move\_particles(self, force) -> None:

'''

Передвижение частиц

'''

values, best\_value, \_ = self.calculate\_best()

for i in range(self.population\_size):

if abs(values[i] - best\_value) > 1e-3:

alpha = 1

alpha = np.random.uniform()

velocity = np.ones\_like(self.x[i])

normalized\_force = force[i] / np.linalg.norm(force[i])

for j in range(self.n):

if force[i][j] > 0:

velocity[j] = self.\_max - self.x[i][j]

else:

velocity[j] = self.x[i][j] - self.\_min

self.x[i] += alpha \* np.multiply(normalized\_force, velocity)

def solve(self) -> None:

'''

Запуск алгоритма

'''

self.create\_population()

history = []

max\_iter = 100

for i in range(max\_iter):

\_, best\_value, best\_x = self.calculate\_best()

history.append(best\_value)

print(

f'Текущее лучшее значение: {round(best\_value, 4)}'

f' в точке {list(map(lambda x: round(x, 4), best\_x))}'

)

print(f'Итерация: {i + 1}')

self.local\_search()

force = self.calculate\_force()

self.move\_particles(force)

def main() -> None:

'''

Главная функция

'''

ema = EMA(2)

ema.solve()

if \_\_name\_\_ == '\_\_main\_\_':

main()

Листинг А.2 – Код файла main.c

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <malloc.h>

#include <string.h>

#include <stdlib.h>

int read\_number(int\* n) {

char buffer[100];

char \*endptr;

if (fgets(buffer, sizeof(buffer), stdin) != NULL) {

char \*newline = strchr(buffer, '\n');

if (newline) {

\*newline = '\0';

}

\*n = strtol(buffer, &endptr, 10);

if (\*endptr != '\0') {

printf("Ошибка: введенная строка содержит нечисловые символы\n");

return -1;

}

return 0;

} else {

printf("Ошибка ввода\n");

return -1;

}

}

int main()

{

int n, m;

n = 0;

m = 0;

while (n < 1) {

printf("Введите высоту матрицы: ");

if (read\_number(&n) != 0) {

return 1;

};

}

while (m < 1) {

printf("Введите ширину матрицы: ");

if (read\_number(&m) != 0) {

return 1;

};

}

int\*\* a = (int\*\*)malloc(n \* sizeof(int\*));

if (a == NULL) {

return -1;

}

for (int i = 0; i < n; i++) {

a[i] = (int\*)malloc(m \* sizeof(int));

if (a[i] == NULL) {

for (int j = 0; j < i; j++) {

free(a[j]);

}

free(a);

return -1;

}

}

for (int i = 0; i < n; i++) {

printf("Введите строку\n");

for (int j = 0; j < m; j++) {

if (read\_number(&a[i][j]) != 0) {

for (int k = 0; k < i; k++) {

free(a[k]);

}

free(a);

return 1;

};

}

}

printf("Результат ввода\n");

for (int i = 0; i < n; i++) {

for (int j = 0; j < m; j++) {

if (j == m - 1) {

printf("%d\n", a[i][j]);

}

else {

printf("%d ", a[i][j]);

}

}

}

n += 1;

int\*\* b = (int\*\*)realloc(a, n \* sizeof(int\*));

if (b == NULL) {

free(a);

return -1;

}

a = b;

a[n - 1] = (int\*)calloc(1, m \* sizeof(int));

if (a[n - 1] == NULL) {

for (int j = 0; j < n; j++) {

free(a[j]);

}

free(a);

return -1;

}

for (int i = n - 1; i > 0; i--) {

for (int j = 0; j < m; j++) {

a[i][j] = a[i - 1][j];

}

}

printf("Введите строку для вставки\n");

for (int j = 0; j < m; j++) {

if (read\_number(&a[0][j]) != 0) {

for (int i = 0; i < n; j++) {

free(a[i]);

}

free(a);

return 1;

};

}

printf("Результат выполнения\n");

for (int i = 0; i < n; i++) {

for (int j = 0; j < m; j++) {

printf("%d ", a[i][j]);

}

printf("\n");

}

for (int i = 0; i < n; i++) {

free(a[i]);

}

free(a);

return 0;

}