|  |
| --- |
|  |
| 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 | Яськов Л.В. |
| Принял ассистент | Петрова А.А. |

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

Москва 2024

# 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 простого калькулятора, 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.

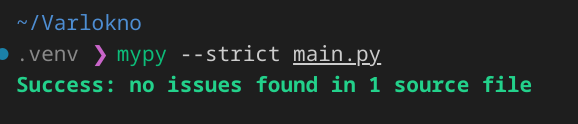


Рисунок 1.1.1 – Анализ программой MyPy

**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.

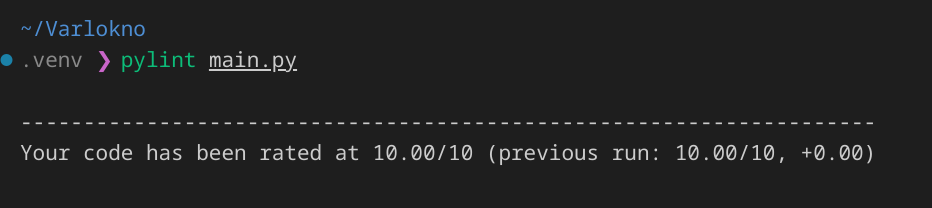


Рисунок 1.1.2 – Анализ программой pylint

**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.

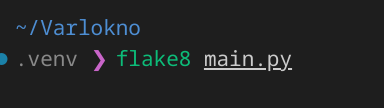


Рисунок 1.1.3 – Анализ программой flake8

### 1.1.1 Добавление ошибок в код

В код исходной программы добавлено 5 ошибок:

1. Из функций были убраны строки документации.
2. Одна строка была сделана большой длины (больше 79 символов).
3. Добавлено использование eval.
4. Допущена ошибка в типизации функции.
5. Добавлено переопределение переменной со сменой её типа.

Результаты анализа представлены на Рисунках 1.1.4 - 1.1.6.

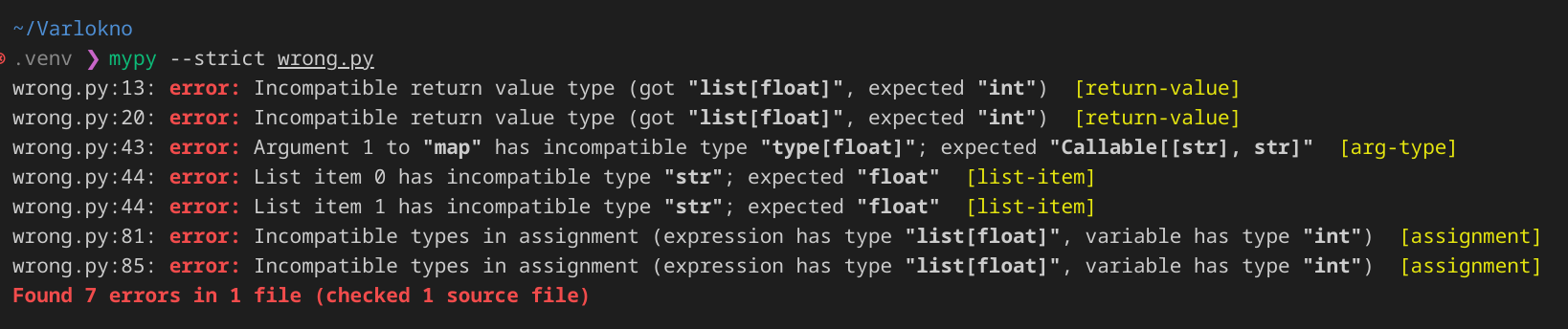


Рисунок 1.1.4 – Анализ программой mypy

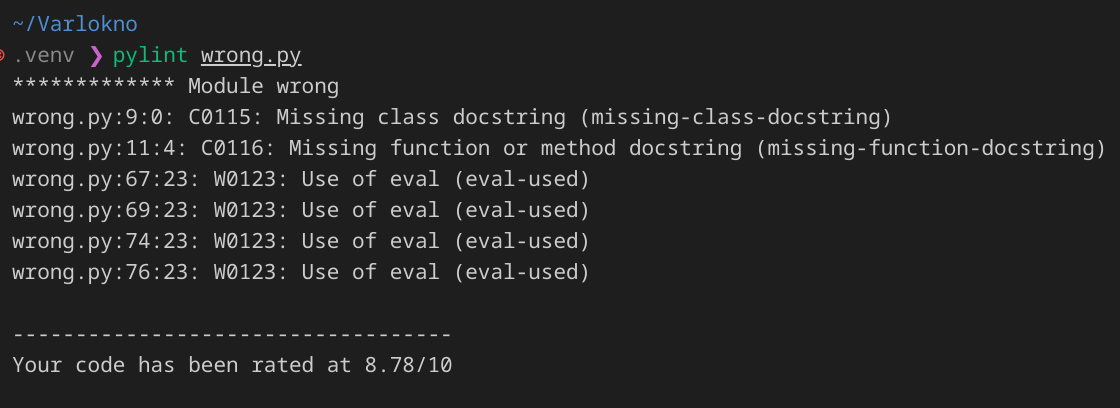


Рисунок 1.1.5 – Анализ программой pylint

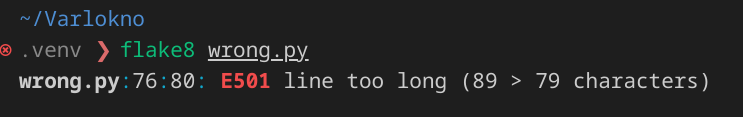


Рисунок 1.1.6 – Анализ программой flake8

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

## 1.2 Анализ кода на языке C++

Для анализа взята программа на языке C++, выполняющая кодирование строки методом RLE. Её код представлен в Листинге А.2.

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

**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.

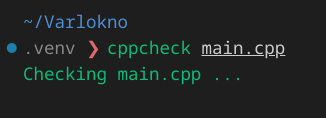


Рисунок 1.2.1 – Результат анализа cppcheck

**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.

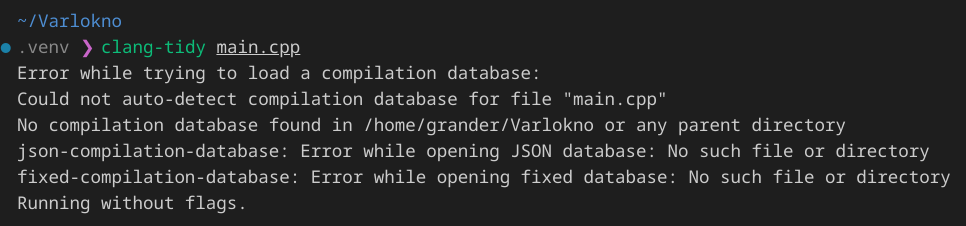


Рисунок 1.2.2 – Результат анализа clang-tidy

Flawfinder — это статический анализатор исходного кода, который помогает обнаруживать потенциальные уязвимости в программах на языках C и C++. Он анализирует исходный код и выдает предупреждения о возможных проблемах безопасности, таких как переполнение буфера, использование небезопасных функций и другие уязвимости. Результат анализа представлен на Рисунке 1.2.3.

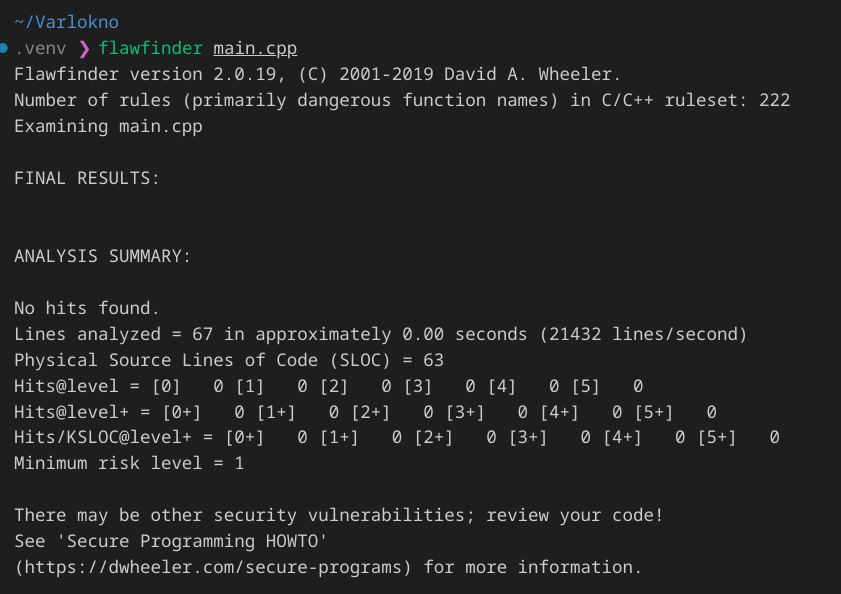


Рисунок 1.2.3 – Результат анализа flawfinder

### 1.2.1 Внесённые ошибки

В код исходной программы добавлено 5 ошибок:

1. В функции передаются аргументы по значению, а не по константной ссылке.
2. Использовано С-шное преобразование строк .
3. Использованы сишные функции строк.
4. Добавлено обращение к неинициализированному участку памяти.
5. Использован free и new в одной программе.

На Рисунках 1.2.4 - 1.2.6 представлены результаты анализа получившегося кода тремя анализаторами.

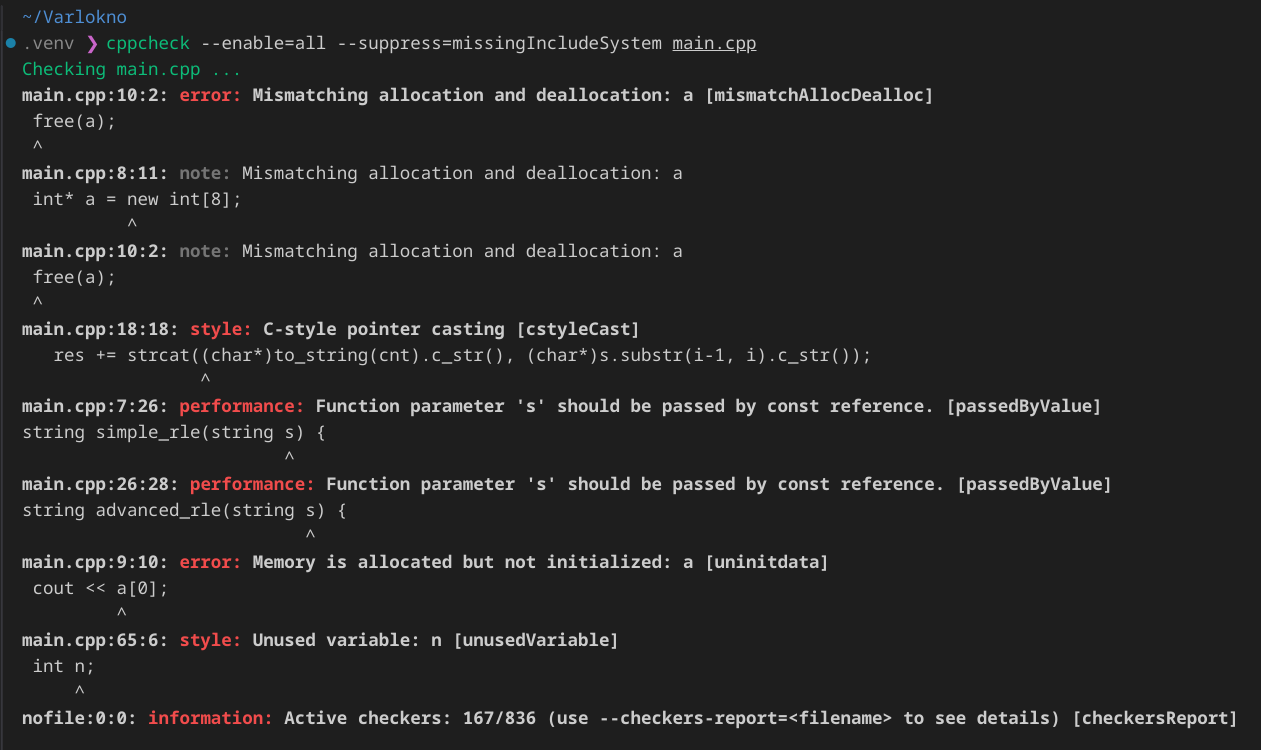


Рисунок 1.2.4 – Результат анализа cppcheck

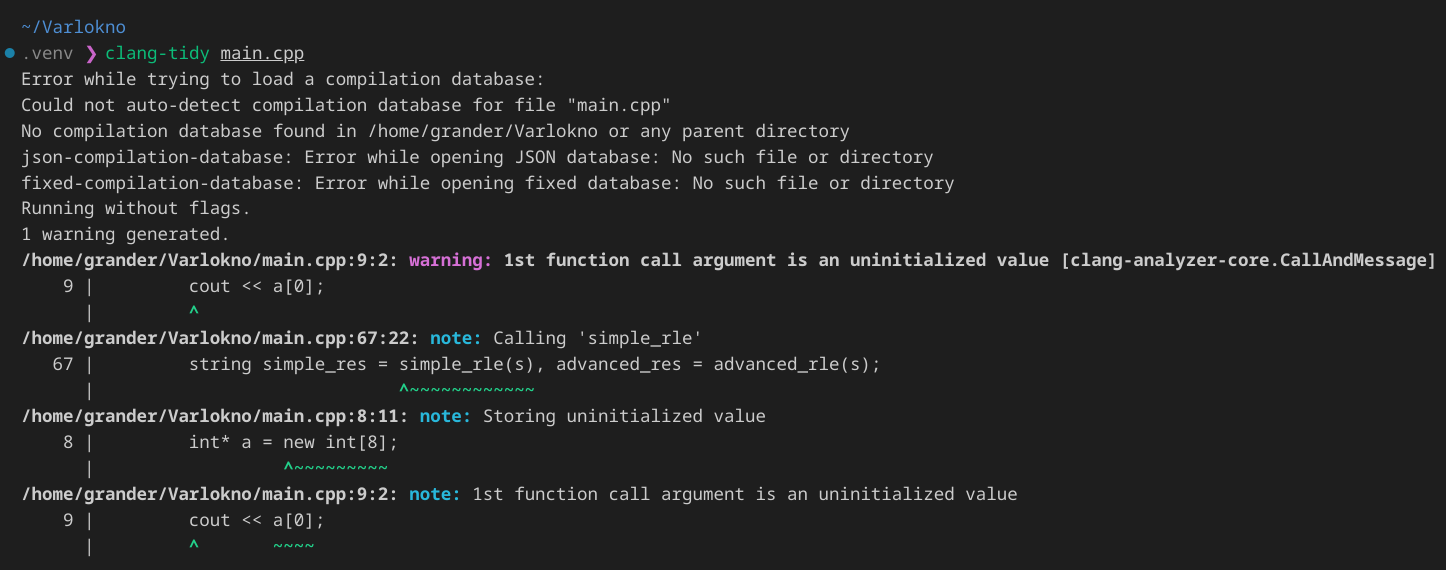


Рисунок 1.2.5 – Результат анализа clang-tidy

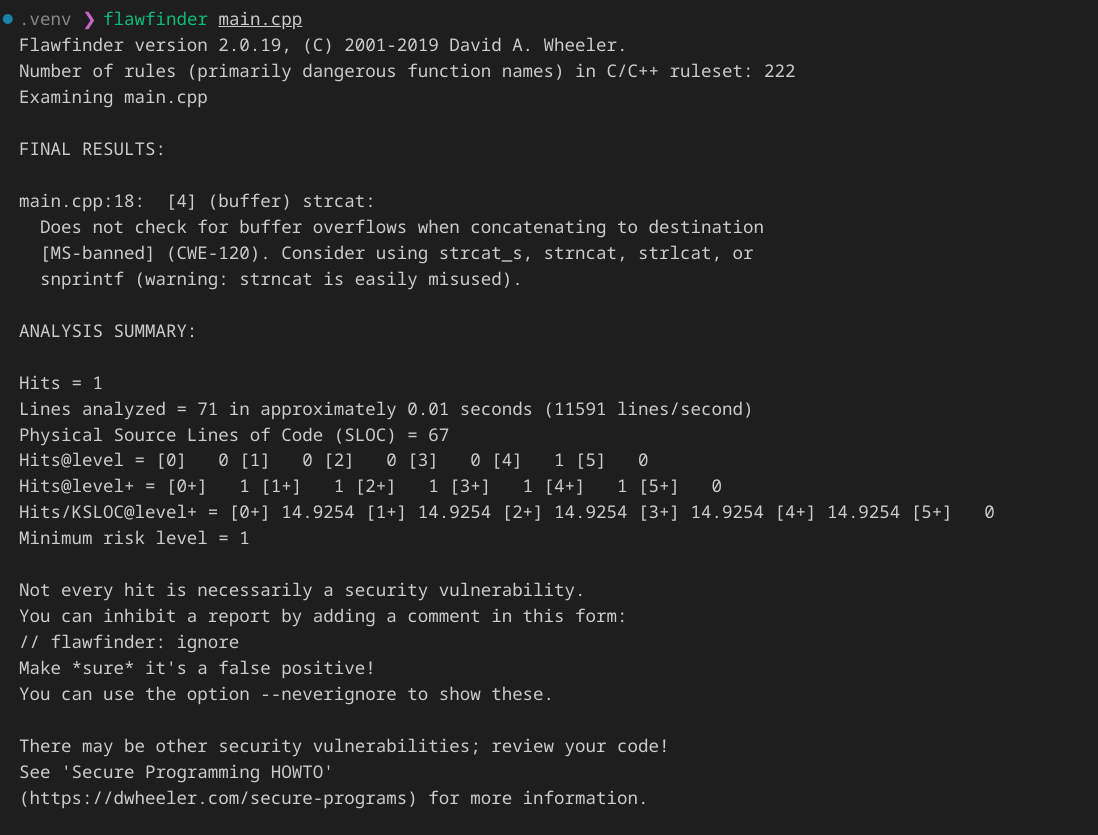


Рисунок 1.2.6 – Результат анализа flawfinder

Все программы смогли обнаружить обращение к неинициализированному участку памяти; cppcheck также обнаружил передачу аргумента в функцию по значению и объявление неиспользуемой переменной; flawfinder указал на использование функции из C strcat; gcc –fanalyzer указал на бесконечный цикл.

## 1.3 Вывод

1. Предотвращение ошибок на ранних этапах: Статический анализ позволяет выявлять потенциальные проблемы с типами данных и другие ошибки еще до запуска программы, что значительно ускоряет процесс разработки и улучшает качество кода.

2. Улучшение ясности и поддержки кода: Аннотации типов, используемые в статическом анализе, делают код более понятным и упрощают его поддержку, особенно в крупных проектах.

3. Автоматизация контроля качества: Инструменты статического анализа могут быть интегрированы в процессы CI/CD, что позволяет автоматически проверять код на соответствие стандартам и правилам.

4. Повышение уровня безопасности: Статический анализ помогает выявлять уязвимости, связанные с неправильными типами данных, что особенно важно для обеспечения безопасности приложений.

# 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.

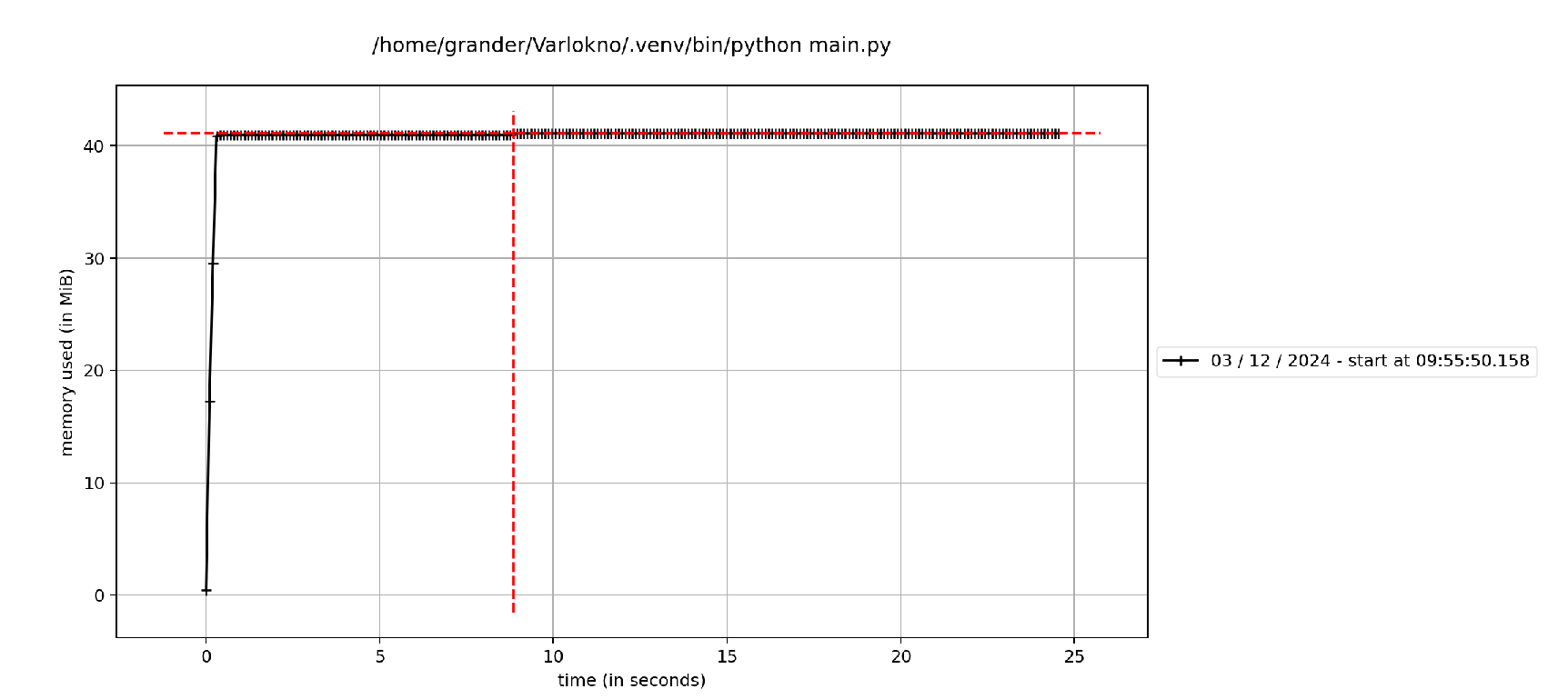


Рисунок 2.1.1 – Результат анализа memory\_profiler

**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.

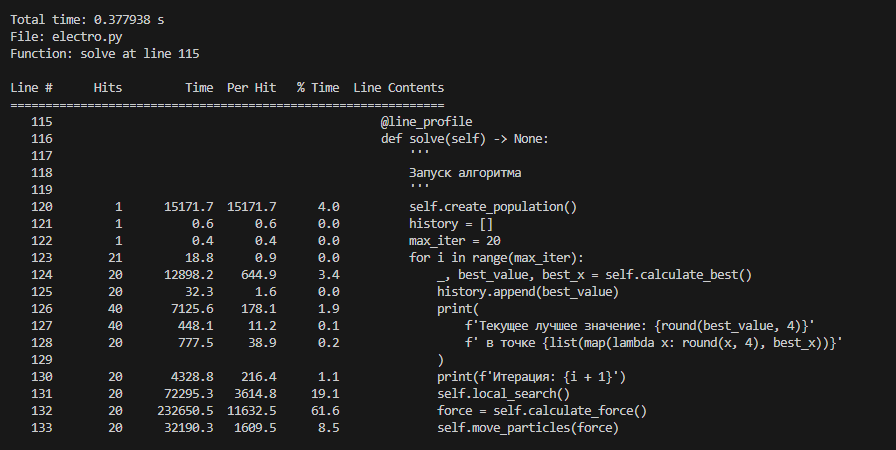


Рисунок 2.1.2 – Результат анализа line\_profiler

### 2.1.1 Внесённые ошибки

В код программы внесены три ошибки:

* в функции вычисления суммы матриц добавлена задержка на 10 секунд;
* в главную функцию добавлено лишнее заполнение массива;
* неэффективное вычисление случайного числа.

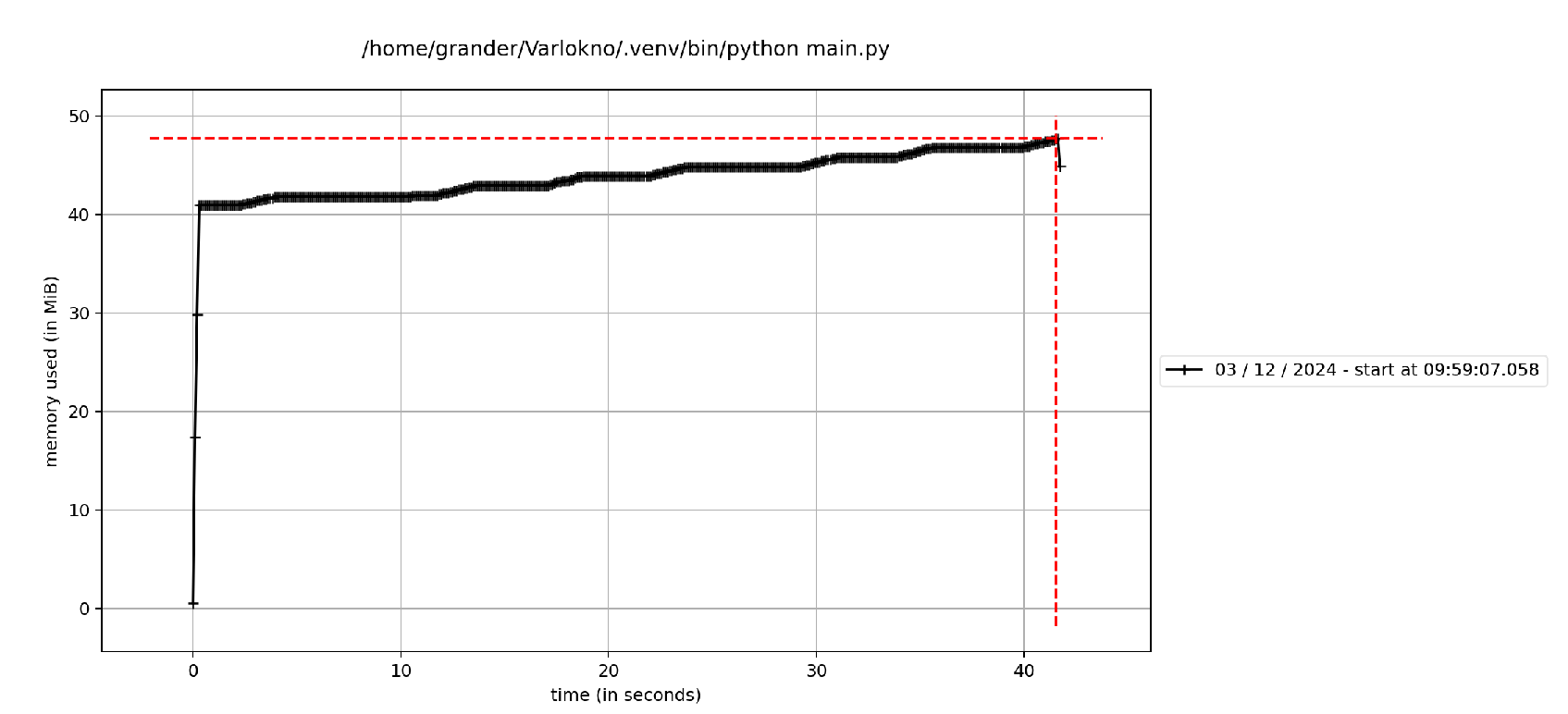


Рисунок 2.1.6 – Результат анализа memory\_profiler

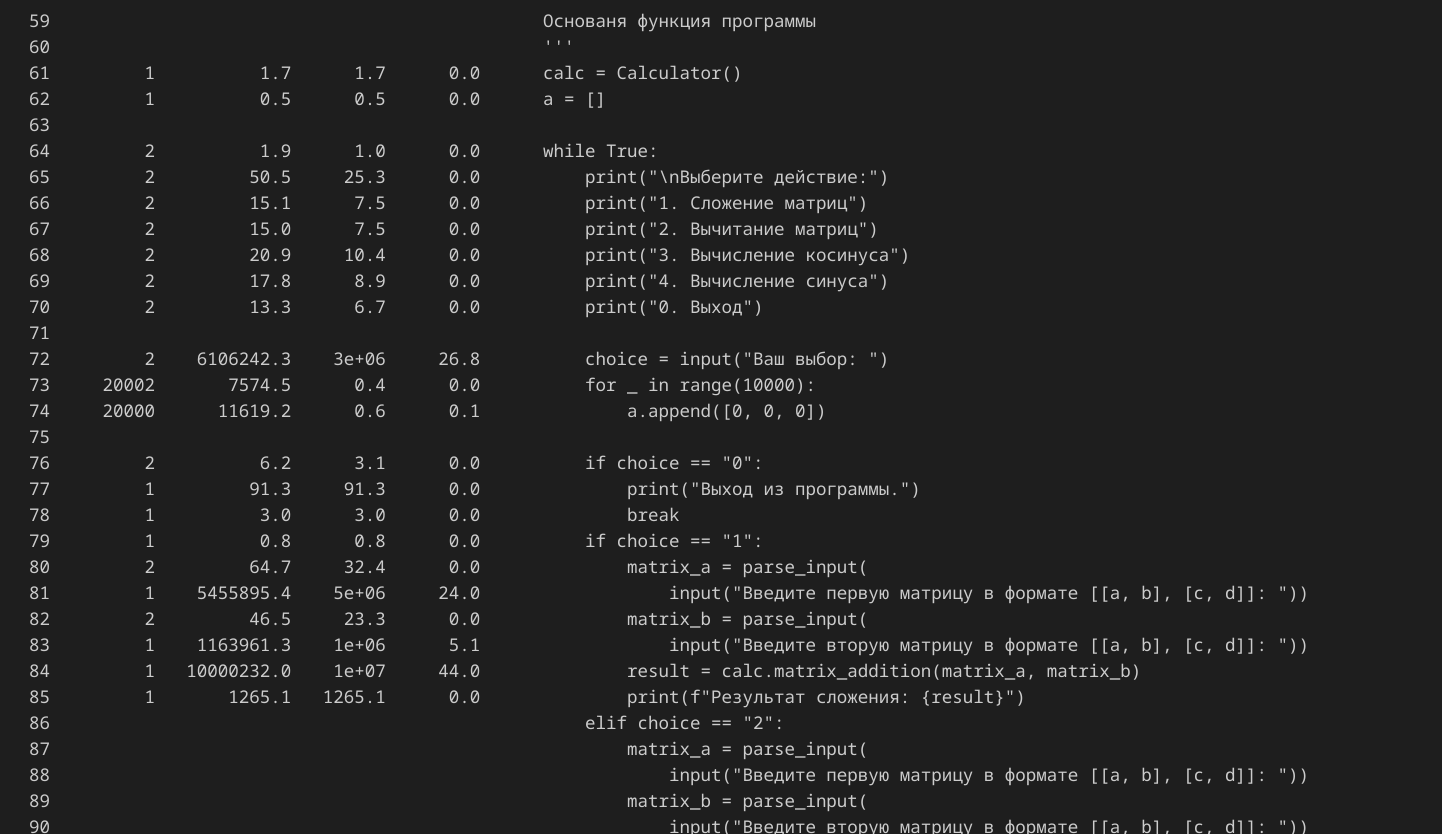


Рисунок 2.1.7 – Результат анализа line\_profiler

## 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.

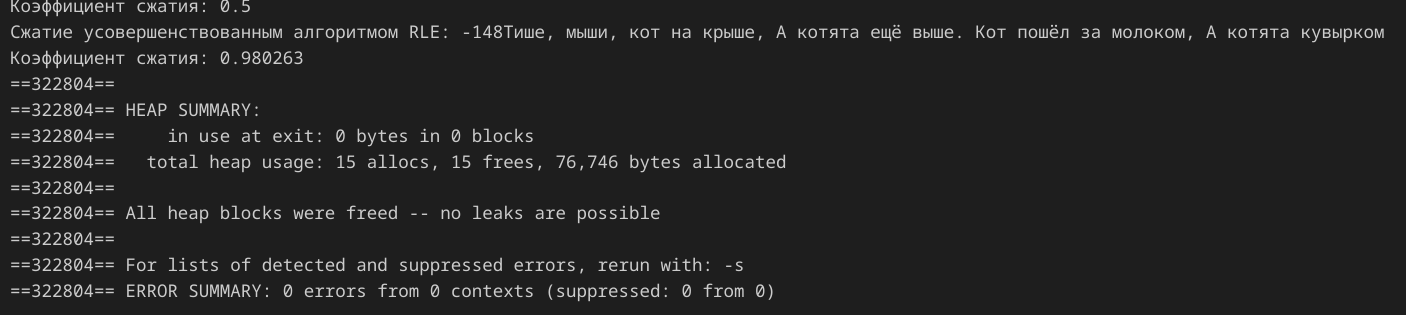


Рисунок 2.2.1 – Результат анализа Valgrind

**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.

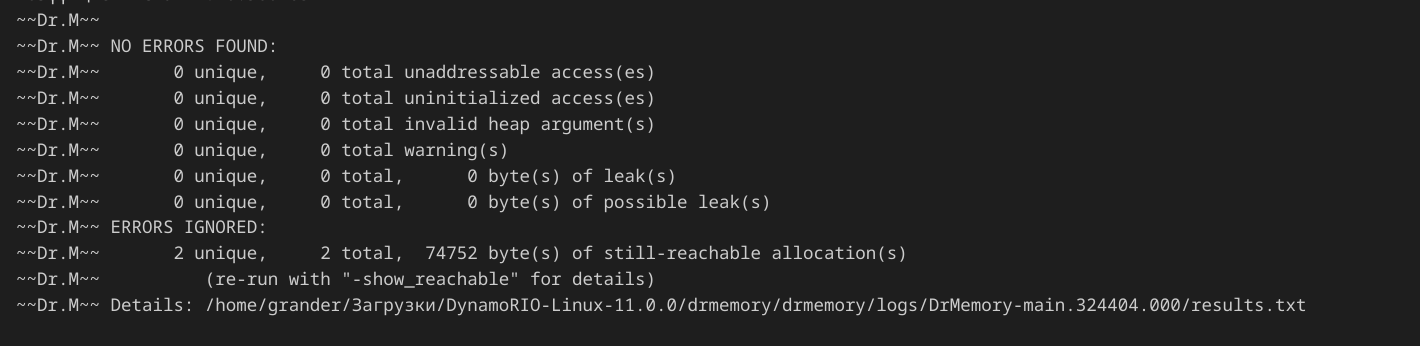


Рисунок 2.2.2 – Результат анализа DynamoRIO

### 2.2.1 Внесённые ошибки

В код программы внесены следующие ошибки:

1. Добавлено создание динамического массива без очистки памяти.
2. Добавлено обращение к неинициализированному участку памяти.
3. Сделана очистка незанятой памяти с помощью функции free.

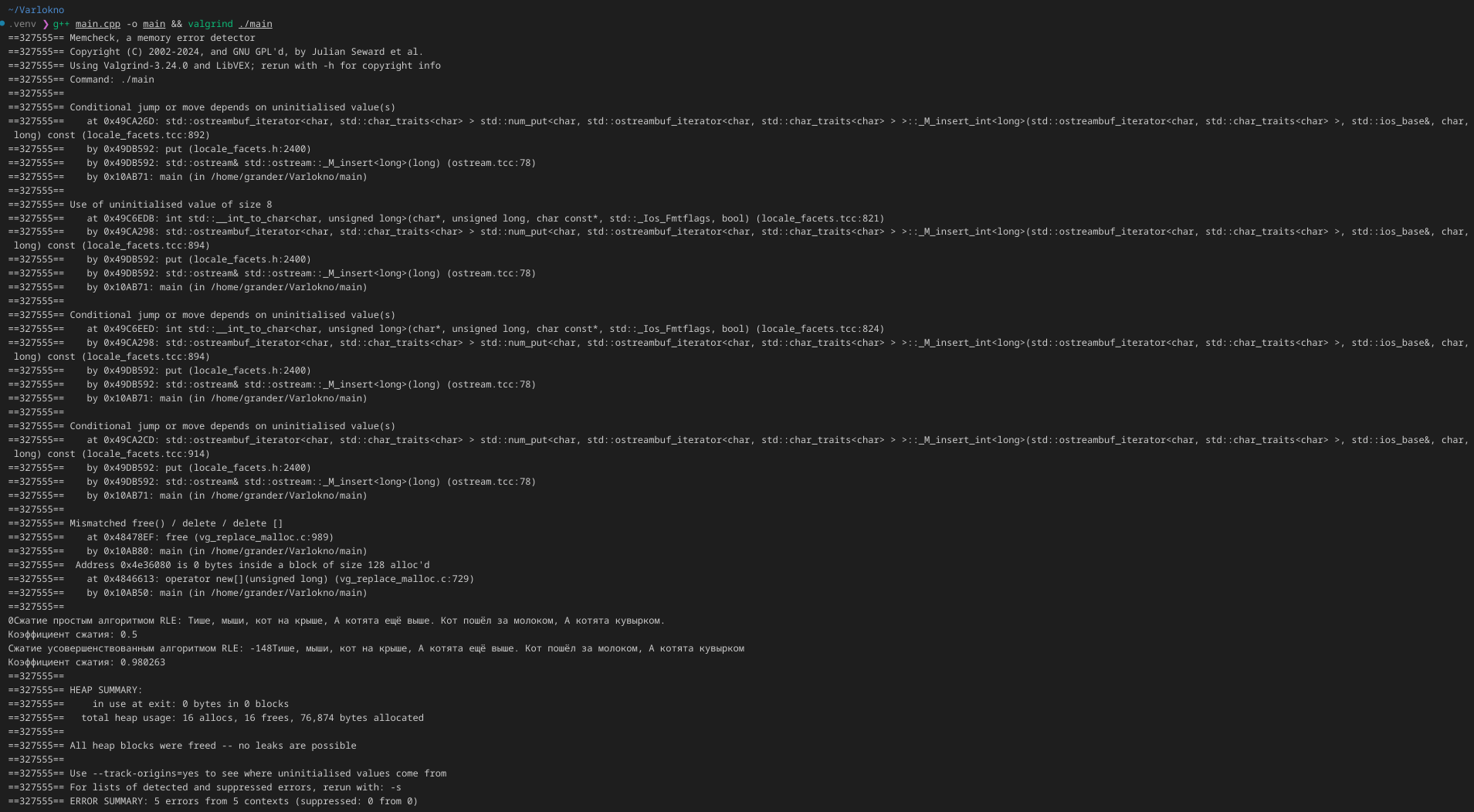


Рисунок 2.2.3 – Результат анализа Valgrind

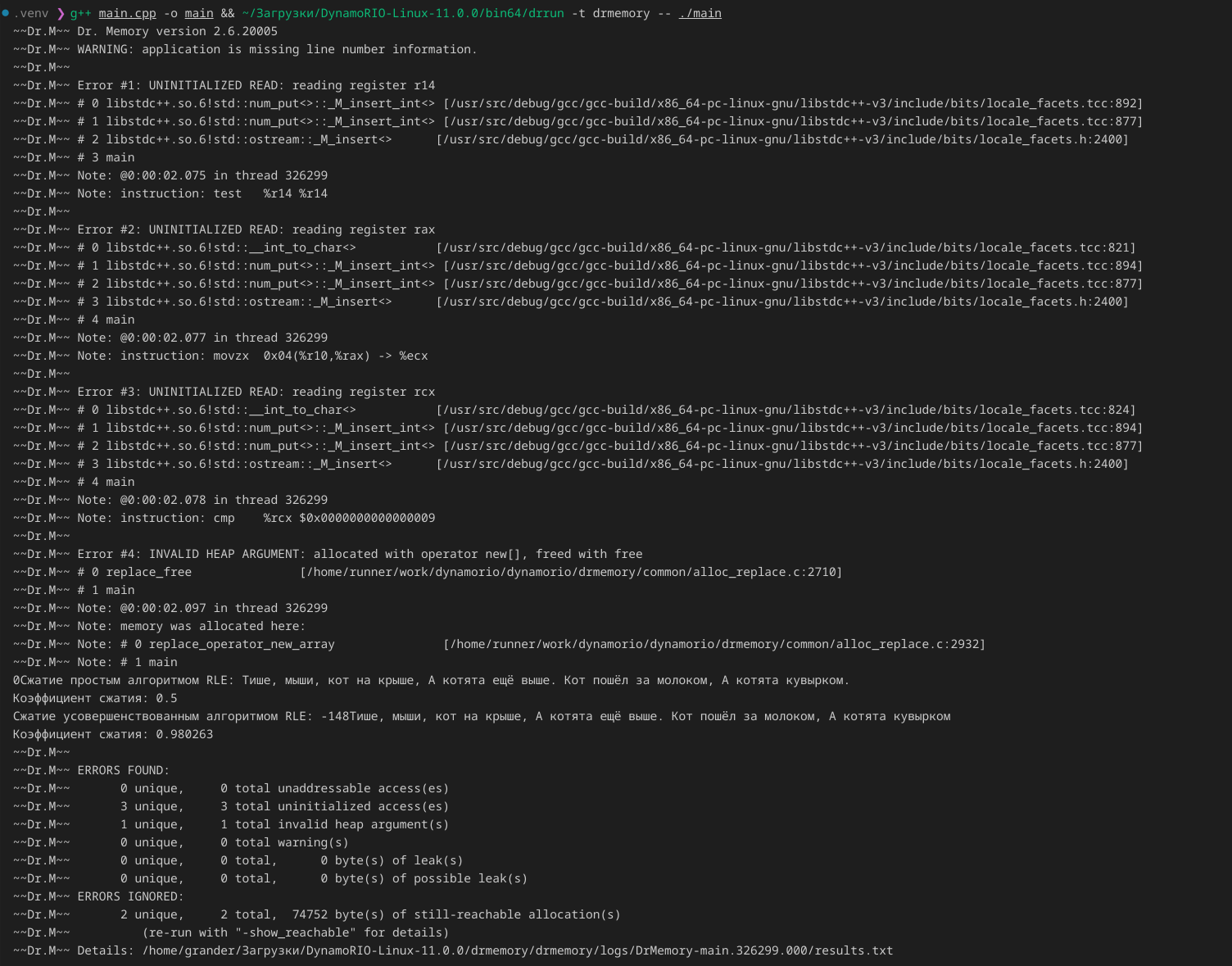


Рисунок 2.2.4 – Результат аналиха DynamoRIO

## 2.3 Вывод

1. Анализ поведения во время выполнения: Динамический анализ позволяет отслеживать поведение программы в реальном времени, что полезно для выявления ошибок, которые невозможно обнаружить статическим анализом.

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

3. Отладка и трассировка: Динамический анализ позволяет отслеживать поток выполнения программы, что упрощает отладку и понимание причин возникновения ошибок.

4. Анализ взаимодействия с внешними системами: Динамический анализ помогает выявить проблемы, связанные с взаимодействием программы с внешними системами, которые сложно проверить статически.

# Вывод

- Статический анализ обеспечивает раннее выявление ошибок и улучшение качества кода на этапе разработки, что сокращает время на отладку и повышает безопасность.

- Динамический анализ позволяет отслеживать поведение программы во время её 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 main.py

'''

main.py

Calculate with numpy

'''

import numpy as np

from numpy.typing import NDArray

from memory\_profiler import profile as mem\_profile

from line\_profiler import profile as line\_profile

class Calculator:

'''

Класс калькулятора

'''

def matrix\_addition(self, matrix\_a: list[list[float]],

matrix\_b: list[list[float]]) -> NDArray[np.float32]:

'''

Сложение двух матриц

'''

return np.add(matrix\_a, matrix\_b)

def matrix\_subtraction(self, matrix\_a: list[list[float]],

matrix\_b: list[list[float]]) -> NDArray[np.float32]:

'''

Вычитание двух матриц

'''

return np.subtract(matrix\_a, matrix\_b)

def cosinus\_calculation(self, angle: float) -> float:

'''

Вычисление косинуса

'''

return float(np.cos(angle))

def sinus\_calculation(self, angle: float) -> float:

'''

Вычисление синуса

'''

return float(np.sin(angle))

def parse\_input(s: str) -> list[list[float]]:

'''

Превращение ввода в матрицу

'''

s = s.replace('[', '').replace(']', '').replace(' ', '')

splitted = s.split(',')

n = list(map(float, splitted))

return [[n[0], n[1]], [n[2], n[3]]]

def main() -> None:

'''

Основаня функция программы

'''

calc = Calculator()

a = []

while True:

print("\nВыберите действие:")

print("1. Сложение матриц")

print("2. Вычитание матриц")

print("3. Вычисление косинуса")

print("4. Вычисление синуса")

print("0. Выход")

choice = input("Ваш выбор: ")

if choice == "0":

print("Выход из программы.")

break

if choice == "1":

matrix\_a = parse\_input(

input("Введите первую матрицу в формате [[a, b], [c, d]]: "))

matrix\_b = parse\_input(

input("Введите вторую матрицу в формате [[a, b], [c, d]]: "))

result = calc.matrix\_addition(matrix\_a, matrix\_b)

print(f"Результат сложения: {result}")

elif choice == "2":

matrix\_a = parse\_input(

input("Введите первую матрицу в формате [[a, b], [c, d]]: "))

matrix\_b = parse\_input(

input("Введите вторую матрицу в формате [[a, b], [c, d]]: "))

result = calc.matrix\_subtraction(matrix\_a, matrix\_b)

print(f"Результат вычитания: {result}")

elif choice == "3":

angle = float(input("Введите угол в радианах: "))

float\_result = calc.cosinus\_calculation(angle)

print(f"Косинус угла: {float\_result}")

elif choice == "4":

angle = float(input("Введите угол в радианах: "))

float\_result = calc.sinus\_calculation(angle)

print(f"Синус угла: {float\_result}")

else:

print("Неверный выбор, попробуйте снова.")

if \_\_name\_\_ == "\_\_main\_\_":

main()

Листинг А.2 – Код файла main.cpp

#include <iostream>

#include <string>

#include <string.h>

using namespace std;

string simple\_rle(const string& s) {

string res = "";

int cnt = 1;

for (int i = 1; i < s.size(); i++) {

if (s[i] == s[i - 1]) {

cnt++;

}

else {

res += to\_string(cnt);

res += s[i - 1];

cnt = 1;

}

}

res += to\_string(cnt) + s[s.size() - 1];

return res;

}

string advanced\_rle(const string& s) {

string res = "";

string group = s.substr(0, 1);

int cnt = 1;

for (int i = 1; i < s.size(); i++) {

if (s[i] == s[i - 1]) {

if (group.size() > 1) {

// res += strcat((char\*)to\_string(cnt + 1).c\_str(), (char\*)group.substr(0, -cnt - 1).c\_str());

res += to\_string(cnt + 1) + group.substr(0, -cnt - 1);

cnt = 1;

}

group = "";

cnt++;

}

else {

if (cnt >= 2) {

res += to\_string(cnt) + s[i - 1];

cnt = 1;

}

else if (cnt == 1) {

cnt = -2;

}

else {

cnt--;

}

group += s[i];

}

}

if (group.size() > 1) {

res += to\_string(cnt + 1) + group.substr(0, -cnt - 1);

}

else {

res += to\_string(cnt) + s[s.size() - 1];

}

return res;

}

int main() {

int\* a = new int[32];

cout << a[0];

free(a);

string s = "Тише, мыши, кот на крыше, А котята ещё выше. Кот пошёл за молоком, А котята кувырком.";

string simple\_res = simple\_rle(s), advanced\_res = advanced\_rle(s);

cout << "Сжатие простым алгоритмом RLE: " << s <<

"\nКоэффициент сжатия: " << (double) s.size() / simple\_res.size() << '\n';

cout << "Сжатие усовершенствованным алгоритмом RLE: " << advanced\_res <<

"\nКоэффициент сжатия: " << (double) s.size() / advanced\_res.size() << '\n';

}