**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра информационной безопасности**

**Курсовая РАБОТА**

**по дисциплине «Программирование»**

**Тема: Приложение «Сигнатурный сканер»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3364 |  | Юсфи А. |
| Преподаватель |  | Халиуллин Р.А. |

Санкт-Петербург

2024

**ЗАДАНИЕ на курсовую работу**

|  |  |  |
| --- | --- | --- |
| Студент Юсфи А. | | |
| Группа 3364 | | |
| Тема работы: Приложение «Сигнатурный сканер» | | |
| Исходные данные:  Разработать на языке программирования C или C++ (по выбору студента) сигнатурный сканер. Сигнатурный сканер должен детектировать заданный образец по наличию в сканируемом файле определенной последовательности байтов (сигнатуры) по определенному смещению. Размер сигнатуры должен быть не менее, чем 6 (шесть) байт и не более, чем 8 (восемь) байт, по выбору студента. Поиск сигнатуры необходимо выполнять только в исполняемых файлах, определение формата файла необходимо выполнять по первым байтам файла. Если сигнатура обнаружена в файле, то необходимо вывести имя файла с указанием имени образца, которому соответствует найденная сигнатура. Сигнатура, смещение сигнатуры и название образца должны храниться в отдельном файле и считываться при запуске сигнатурного сканера, путь к этому файлу может вводиться пользователем. Путь к файлу для сигнатурного сканирования должен вводиться пользователем. Приложение должно иметь консольный или графический интерфейс (по выбору студента). В интерфейсе приложения допускается использовать буквы латинского алфавита для транслитерации букв алфавита русского языка. Интерфейс приложения должен быть интуитивно понятным и содержать подсказки для пользователя. В исходном коде приложения должны быть реализованы функции. В исходном коде приложения должны быть реализованы проверки аргументов реализованных функций и проверки возвращаемых функциями значений (для всех функций — как сторонних, так и реализованных). Приложение должно корректно обрабатывать ошибки, в том числе ошибки ввода/вывода, выделения/освобождения памяти и т. д. | | |
| Содержание пояснительной записки:  Введение, теоретическая часть, реализация программы, результат тестирования программы, заключение, список используемых источников, приложение 1 – руководство пользователь, приложение 2 – блок-схема алгоритма, приложение 3 – исходный код программы. | | |
| Предполагаемый объем пояснительной записки:  Не менее 25 страниц. | | |
| Дата выдачи задания: 12.11.2023 | | |
| Дата сдачи реферата: 25.12.2023 | | |
| Дата защиты реферата: 25.12.2023 | | |
| Студент гр. 3364 |  | Юсфи А. |
| Преподаватель |  | Халиуллин Р.А. |

**Аннотация**

Курсовая работа заключается в проверке целостности исполняемого файла путем сравнения сигнатур. Пользователи вводят пути к исполняемому и файлу с сигнатурой. Программа считывает сигнатуру и смещение из файла, убеждаясь в наличии заголовка MZ в исполняемом файле. Она вычисляет размеры для проверки и проверяет целостность файла. После чтения сигнатуры из исполняемого файла она сравнивает ее с предоставленной, чтобы убедиться в совпадении. Наконец, она уведомляет пользователей, безопасен ли файл или нет.

**Summary**

The course work involves validating the integrity of an executable file through signature comparison. Users input paths for the executable and signature files. The program reads the signature and offset from the file, ensuring an MZ header in the executable. It calculates sizes for verification and checks file integrity. After reading the signature from the executable, it compares it with the provided one to ensure a match. Finally, it notifies users whether the file is safe or not..

**содержание**

|  |  |
| --- | --- |
| Введение | 6 |
| 1. Теоретическая часть | 7 |
| * 1. История программированных калькуляторов | 7 |
| 1.2. Эволюция калькуляторов  1.3. Прогноз и возможности перспектив развития | 8  8 |
| 1.4. Роль калькуляторов в цифровой эпохе | 8 |
| 2. Реализация программы | 9 |
| 2.1. Сведения о выбранном языке программирования | 9 |
| 2.2. Сведения о программном обеспечении | 9 |
| 2.3. Сведения о реализованных функциях | 9 |
| 1. Результаты тестирования программы    1. Сведения о результатах тестирования программы | 13  13 |
| Заключение | 21 |
| Список использованных источников | 22 |
| Приложение 1. Руководство пользователя | 23 |
| Приложение 2. Блок-схема алгоритма | 25 |
| Приложение 3. Исходный код программы | 26 |

**введение**

Цель курсового проекта – заключается в разработке приложения для сканирования подписей с использованием языка программирования C и создании отдельных функций для каждой задачи. В начале процесса создаётся блок-схема для визуального представления функционала программы. Интуитивный интерфейс приложения играет важную роль в улучшении взаимодействия с пользователем, поэтому разрабатывается "руководство пользователя". Значимость данной программы заключается в скорости её выполнения и эффективности языка C, что делает его подходящим выбором для разработки сканера подписей, так как программы на языке C могут легко портироваться на различные платформы и использоваться.

1. **Теоретическая часть**
   1. **История программированных калькуляторов**

История программы калькулятора представляет собой удивительный путь развития, начиная с древних механических устройств, таких как Паскалина и ступенчатый счетчик Лейбница. Эти простые, но важные инструменты позволяли производить базовые арифметические операции и положили начало тому, что впоследствии стало эпохой вычислительной техники.

С развитием электроники в 20-м веке появились электромеханические калькуляторы, такие как Гарвардский марк I, ставший одним из первых программируемых устройств. Он использовал перфокарты для ввода инструкций, что открыло путь к созданию более сложных программ.

В 1960-х и 1970-х годах с появлением электронных калькуляторов стало возможным разработать программные модели, которые можно было программировать непосредственно пользователем. К примеру, HP-65 и Texas Instruments SR-52 предлагали функционал хранения последовательностей команд и выполнения повторяющихся вычислений.

С наступлением 1980-х и 1990-х годов программируемые калькуляторы стали более мощными и гибкими благодаря функциям, таким как условное ветвление и пользовательские функции. Их интеграция с персональными компьютерами расширила функциональность, а графические калькуляторы, такие как модели серии TI-83/84, стали неотъемлемым инструментом в образовании.

С появлением 21 века программные калькуляторы на смартфонах и компьютерах сделали программирование и выполнение сложных вычислений более доступными. Они предлагают широкие возможности, включая скриптовые языки программирования типа Python или JavaScript, продолжая традицию улучшения эффективности и доступности математических вычислений. написания одноразовых скриптов. Эволюция языков программирования происходит и сегодня и будет продолжаться долгое время.

* 1. **Эволюция калькуляторов**

Технологический прогресс преобразовал калькуляторы из простых механических устройств в сложные цифровые инструменты. Программируемые функции, интеграция с компьютерами и мобильными платформами расширили возможности. Графические калькуляторы и программные решения дополнили функционал, отражая постоянное развитие в математических вычислениях.

* 1. **Прогноз и возможности перспектив развития**

Перспективы развития - представляют собой прогнозы и возможности, связанные с будущим развитием общества, технологий, экономики и науки. Это включает в себя ожидания относительно того, как могут измениться жизнь людей, бизнес-процессы, методы производства, а также научные и технические достижения. Анализ перспектив развития помогает понять, какие тренды и возможности могут стать ключевыми в формировании нашего будущего и какие вызовы нам предстоит преодолеть.

* 1. **Роль калькуляторов в цифровой эпохе**

Роль калькуляторов в цифровой эпохе - выражает значимость этих инструментов в современном мире, где цифровые технологии играют ключевую роль. С развитием технологий калькуляторы становятся мощными инструментами для выполнения различных математических операций, обеспечивая широкий спектр возможностей от базовых арифметических расчетов до сложного анализа данных и программирования. В образовании они помогают студентам освоить математические концепции и развить аналитическое мышление, а в бизнесе используются для финансового анализа, прогнозирования и принятия решений. Калькуляторы становятся неотъемлемой частью повседневной жизни, обеспечивая удобство, точность и эффективность в выполнении различных задач в различных сферах деятельности.

**2. Реализация Программы**

* 1. **Сведения о выбранном языке программирования**

Язык программирования C – объединяет в себе характеристики языков высокого уровня и производительность ассемблера, обеспечивая универсальную компилируемую среду со статической типизацией. Это позволяет эффективно взаимодействовать с аппаратной частью компьютера и создавать сложные конструкции на высоком уровне, а также обеспечивает переносимость исходного кода программы на широкий спектр платформ.

* 1. **Сведения о программном обеспечении**

Операционная система Windows 11 Pro, версия 22H2, 64-разрядная, среды разработки – Clion и CodeBlocks, компилятор – GNU GCC Compiler.

* 1. **Сведения о реализованных функциях**

Реализованные функции: Somme(), Substruction(), Multiplication(), division(), main().

Функция main.

Функция main принимает три значения в качестве входных параметров: первое число, второе число и номер операции, которую пользователь желает выполнить над числами (сложение, вычитание, умножение или деление). Затем программа определяет результат выбранной операции, введенной пользователем с клавиатуры.

Исходный код функции находится в файле main.c.

Объявление функции:

int main();

Тип функции: int.

Аргументы функции:

* Аргументы отсутствуют.

Возвращаемое функцией значение:

* 0 — выполнение функции прошло без ошибок;
* 1 — ошибка функции printf();
* 2 — ошибка функции printf();
* 2 — значение второго аргумента из какой-либо внешней реализованной функции равно 0;
* 3 — ошибка функции printf();
* 3 — значение третьего аргумента из какой-либо внешней реализованной функции некорректно;
* 4 — ошибка функции printf();
* 5 — ошибка функции Somme();
* 6 — ошибка функции Substruction();
* 7 — ошибка функции Multiplication();
* 8 — ошибка функции Division();
* 9 — ошибка функции printf();
* 10 — ошибка функции printf().

Функция Somme.

Функция Somme определяет значение целочисленного сложения двух чисел.

Исходный код функции Somme находится в файле main.c.

Объявление функции:

int Somme(int a, int b, int\*c);

Тип функции: int.

Аргументы функции:

* a — переменная, хранящая значение первого слагаемого, тип аргумента: int;
* b — переменная, хранящая значение второго слагаемого, тип аргумента: int;
* c — указатель на переменную, куда возвращается значение после операции сложения двух слагаемых a и b;

Возвращаемое функцией значение:

* 0 — выполнение функции прошло без ошибок;
* 3 — значение третьего аргумента функции некорректно.

Функция Substruction.

Функция Substruction определяет значение целочисленного вычитания двух чисел.

Исходный код функции sub находится в файле main.c.

Объявление функции:

int Substruction (int a, int b, int\*c);

Тип функции: int.

Аргументы функции:

* a — переменная, хранящая значение уменьшаемого, тип аргумента: int;
* b — переменная, хранящая значение вычитаемого, тип аргумента: int;
* c — указатель на переменную, куда возвращается значение после операции вычитания двух чисел a и b;

Возвращаемое функцией значение:

* 0 — выполнение функции прошло без ошибок;
* 3 — значение третьего аргумента функции некорректно.

Функция Multiplication.

Функция Multiplication определяет значение целочисленного умножения двух чисел.

Исходный код функции Multiplication находится в файле main.c.

Объявление функции:

int Multiplication (int a, int b, int\*c);

Тип функции: int.

Аргументы функции:

* a — переменная, хранящая значение первого множителя, тип аргумента: int;
* b — переменная, хранящая значение второго множителя, тип аргумента: int;
* c — указатель на переменную, куда возвращается значение после операции умножения двух чисел a и b;

Возвращаемое функцией значение:

* 0 — выполнение функции прошло без ошибок;
* 3 — значение третьего аргумента функции некорректно.

Функция Division.

Функция Division определяет значение целочисленного деления двух чисел.

Исходный код функции div находится в файле main.c.

Объявление функции:

int Division (int a, int b, int\*c);

Тип функции: int.

Аргументы функции:

* a — переменная, хранящая значение делимого, тип аргумента: int;
* b — переменная, хранящая значение делителя, тип аргумента: int;
* c — указатель на переменную, куда возвращается значение частного после операции деления двух чисел a и b;

Возвращаемое функцией значение:

* 0 — выполнение функции прошло без ошибок;
* 2 — значение второго аргумента функции некорректно;
* 3 — значение третьего аргумента функции некорректно.

1. **Результаты тестирования программы** 
   1. **Сведения о результатах тестирования программы**

При старте программы пользователь видит приветственное сообщение и инструкции по использованию калькулятора, включая перечень операций и их соответствующих номеров. После чего программа просит ввести первое число (рисунок 1).

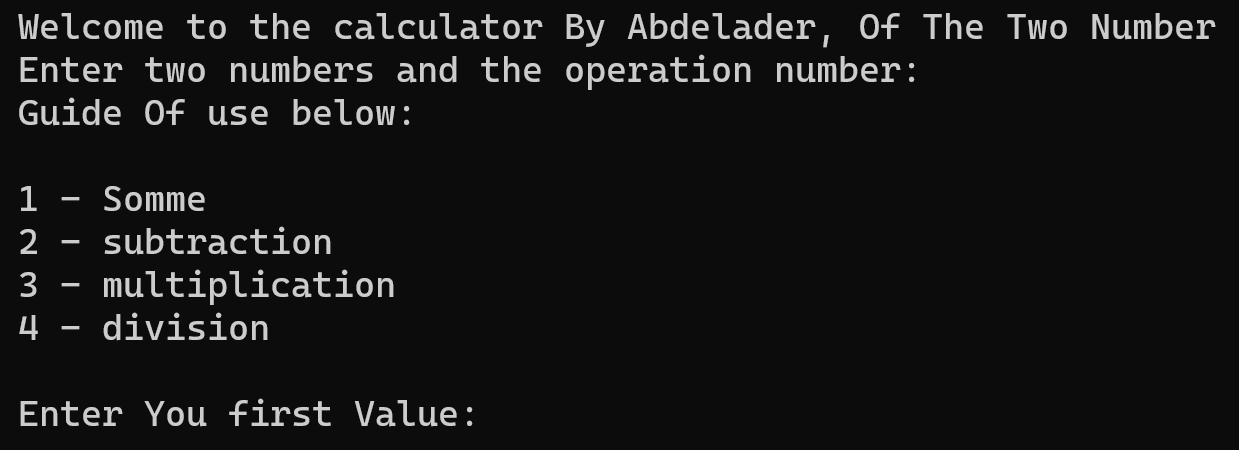
****

Рисунок 1 — Главный экран запуска программы

Приложение запрашивает ввод числа от пользователя. Если введенный символ не является числом, программа выводит сообщение об ошибке и просит пользователя ввести число снова. Этот процесс повторяется, пока не будет введено корректное значение (рисунок 2).

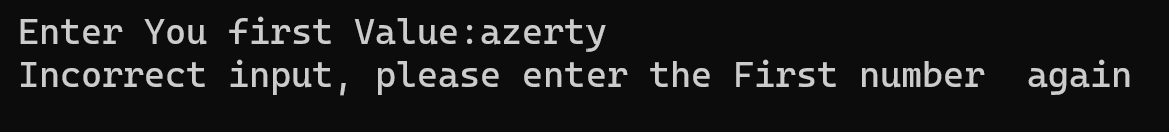


Рисунок 2 — Ввод значения вида «символ»

При вводе комбинации символа и числа программа также будет выводить сообщение с запросом на повторный ввод числа, продолжая этот процесс до тех пор, пока не будет получено корректное значение (рисунок 3).

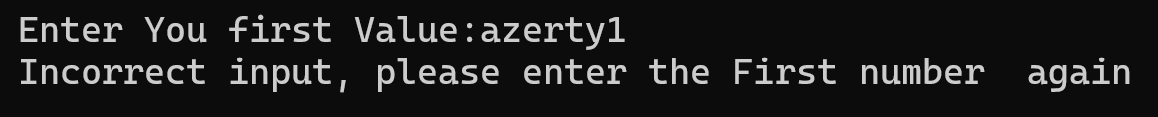


Рисунок 3 — Ввод значения вида «символ» + «число»

При вводе комбинации числа и символа, программа запишет в качестве первого числа значение, указанное до символа, и затем перейдет к запросу второго числа (рисунок 4).

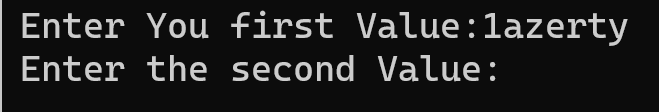


Рисунок 4 — Ввод значения вида «число» + «символ»

При вводе числа, как показано на (рисунке 5), или числа с плавающей точкой, как показано на (рисунке 6), программа обрабатывает его следующим образом: она просто считывает это значение и записывает его как первое число, после чего запрашивает ввод второго числа.

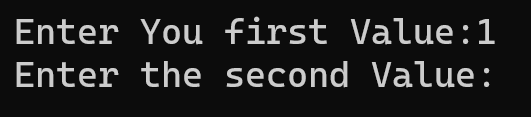


Рисунок 5 — Ввод значения вида «число»

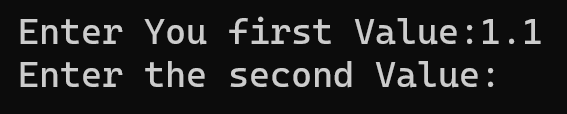


Рисунок 6 — Ввод значения вида «число с плавающей точкой»

После ввода первого числа, на вход подается второе, что программа и потребует от пользователя (рисунок 7).

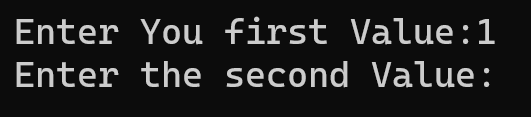


Рисунок 7 — Запрос на ввод второго числа

Приложение запрашивает от пользователя ввод числа. Если введенный символ не является числом, программа выводит сообщение об ошибке и просит повторно ввести число. Этот процесс повторяется до тех пор, пока не будет введено корректное значение (рисунок 8).

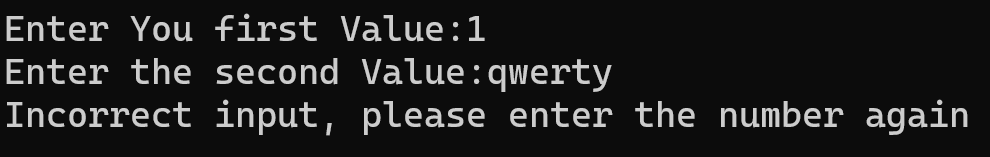


Рисунок 8 — Ввод значения вида «символ»

При вводе комбинации символа и числа, программа также будет выводить сообщение с запросом на повторный ввод числа, продолжая этот процесс до получения корректного значения (рисунок 9).

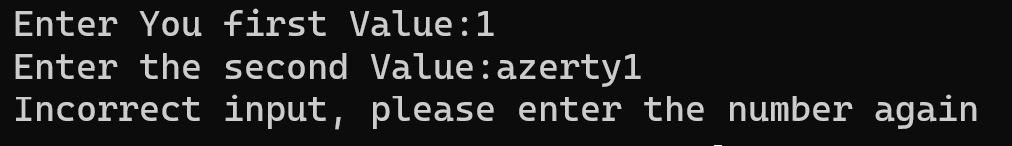


Рисунок 9 — Ввод значения вида «символ» + «число»

При вводе числа, за которым следует символ, программа запишет второе число как значение, указанное до символа, а затем запросит ввод номера операции (рисунок 10).

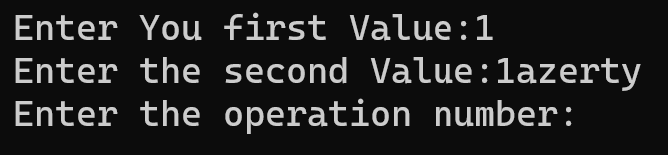


Рисунок 10 — Ввод значения вида «число» + «символ»

При вводе только числа, как показано на (рисунке 11), или числа с плавающей точкой, как показано на (рисунке 12) (поскольку программа не обрабатывает такие значения и принимает только целочисленные значения), она обычным образом принимает это число и записывает его как второе число, после чего запрашивает ввод номера операции.

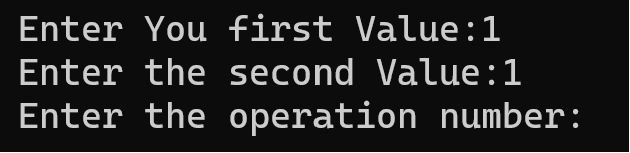


Рисунок 11 — Ввод значения вида «число»

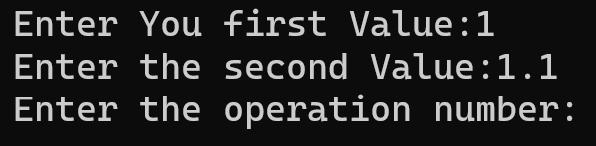


Рисунок 12 — Ввод значения вида «число с плавающей точкой»

После ввода второго числа, на вход подается номер операции, что программа и потребует от пользователя (рисунок 13).

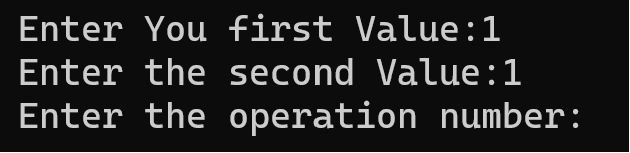


Рисунок 13 — Запрос на ввод номера операции

Пользователь должен ввести число, соответствующее выбранной арифметической операции из представленной выше инструкции. Если введенный символ не является числом, программа выводит сообщение об ошибке и просит пользователя повторно ввести число. Этот процесс повторяется до тех пор, пока не будет получено корректное значение (рисунок 14).

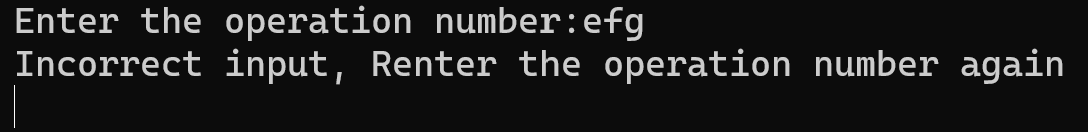


Рисунок 14 — Ввод значения вида «символ»

При вводе комбинации символа и числа, программа также будет выводить сообщение с запросом на повторный ввод числа и будет продолжать это делать до получения корректного значения (рисунок 15).

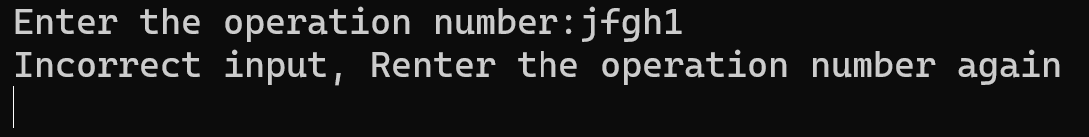


Рисунок 15 — Ввод значения вида «символ» + «число»

При вводе комбинации числа, соответствующего номеру арифметической операции, и символа, программа запишет в качестве номера операции значение, указанное до символа. Затем она выведет результат выполнения введенной операции и завершит программу (рисунок 16).

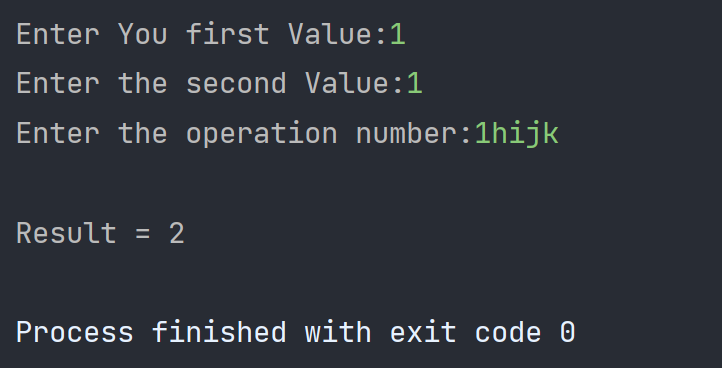


Рисунок 16 — Ввод значения вида: «число, соответствующее номеру арифметической операции» + «символ»

При вводе числа, которое не является номером операции, как показано на (рисунке 17), программа выведет сообщение об ошибке с просьбой ввести значение снова. Это будет продолжаться даже в случае, когда вводится значение в виде: "число, несоответствующее номеру арифметической операции" + "символ", как показано на (рисунке 18), и будет продолжаться до тех пор, пока не будет получено корректное значение.

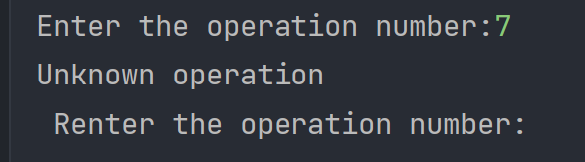


Рисунок 17 — Ввод числа, не являющегося номером арифметической операции

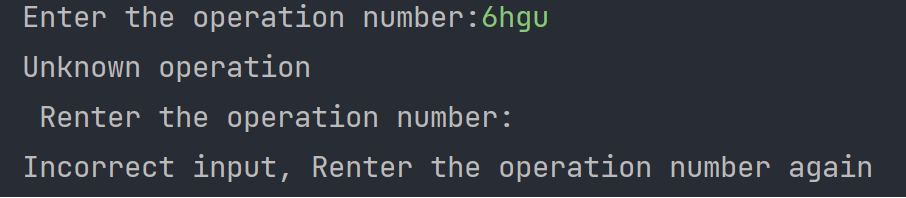


Рисунок 18 — Ввод значения вида «число, несоответствующее номеру арифметической операции» + «символ»

При вводе значения вида «символ» + «число, несоответствующее номеру арифметической операции», программа выдаст сообщение об ошибке и запросит ввести номер операции снова и будет это делать до тех пор, пока корректное значение не будет получено (рисунок 19).

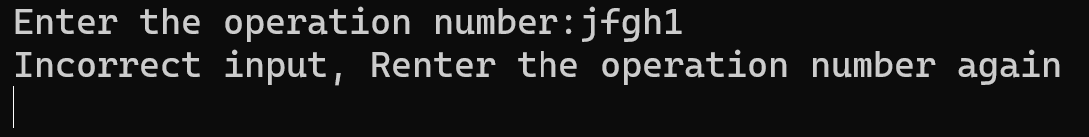


Рисунок 19 — Ввод значения вида «символ» + «число, несоответствующее номеру арифметической операции»

При вводе числа, которое соответствует номеру арифметической операции, программа записывает значение этого числа и выводит результат выполнения соответствующей арифметической операции с ранее введенными числами. После этого программа завершает свою работу, как показано на (рисунке 20).

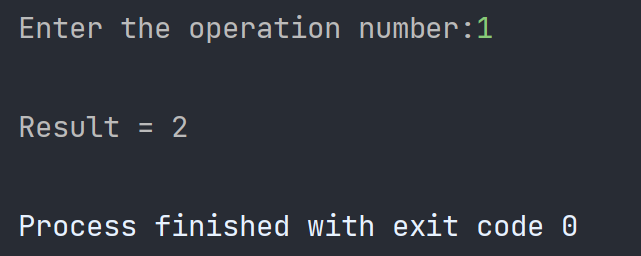


Рисунок 20 — Ввод значения вида «число, соответствующее номеру арифметической операции»

Вид законченной программы (рисунок 21).

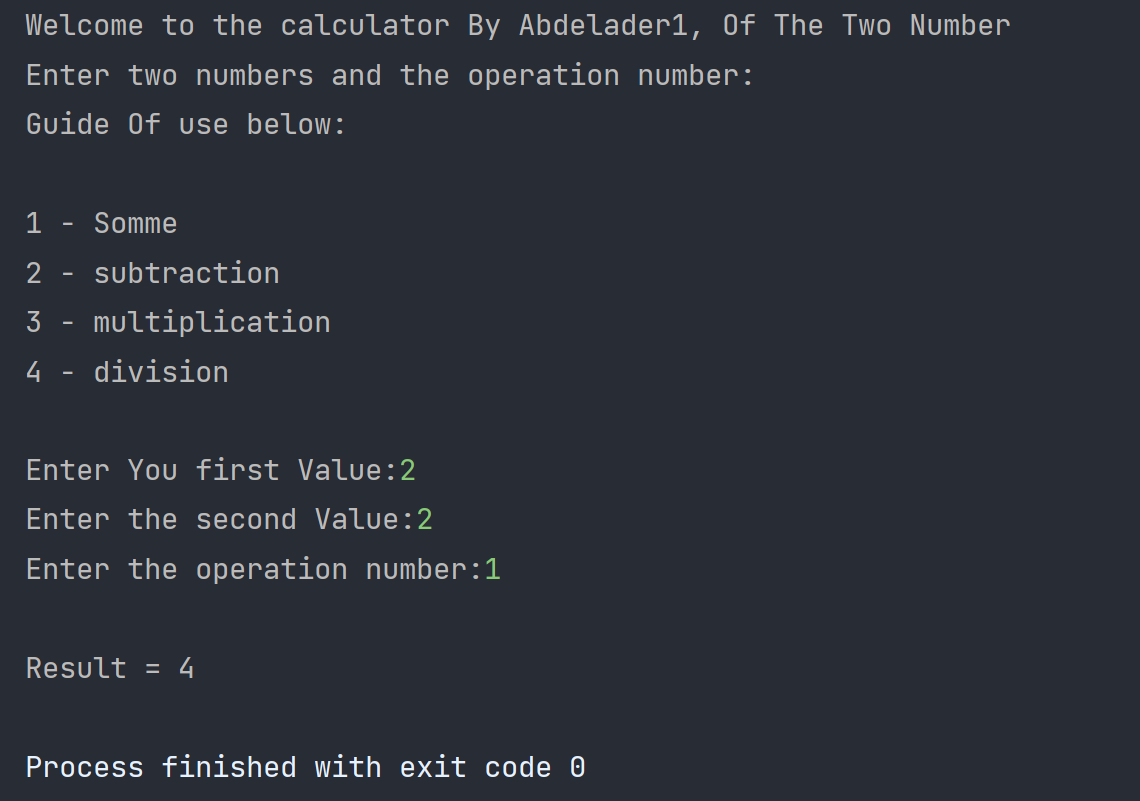


Рисунок 21 — Окно успешно выполненной программы

Допустим, что второй аргумент, введенный пользователем, равен нулю, а затем выбрана операция деления (номер 4). В таком случае, программа выдаст сообщение об ошибке и завершит свою работу (Рисунок 22).

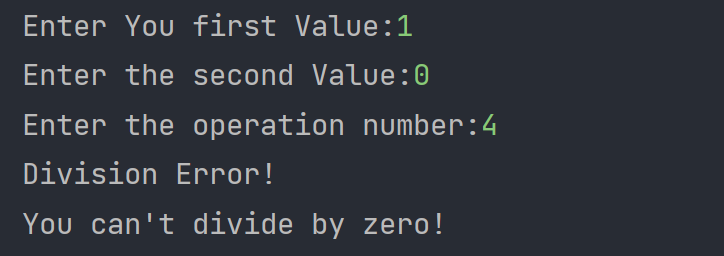


Рисунок 22 — Ошибка деления на ноль

**заключение**

В результате выполнения проекта была достигнута основная цель — представляет собой эффективный инструмент, способный обеспечить проверку целостности исполняемых файлов путем сравнения подписей. Создание отдельных функций для каждой задачи и разработка интуитивного интерфейса, сопровождаемого "руководством пользователя", демонстрируют высокий уровень организации проекта. Благодаря использованию языка C, приложение обладает быстрой скоростью выполнения и высокой эффективностью. Результаты данного проекта подтверждают перспективность и практичность разработки сканера подписей на основе C, делая его важным вкладом в область информационной безопасности.

**список использованных источников**

1. Трохименко Я.К., Любич Ф.Д. Микрокалькулятор, Ваш ход! М.: Радио и связь, 1985. 223 с.
2. Calculating machines and instrument: history, technology , and applications by Geoffrey W.A.Dummer.
3. О Standard C Library Functions Table, By Name [Электронный ресурс].URL:https://www.ibm.com/support/knowledgecenter/ssw\_ibm\_i\_74/rtref/stalib.htm (дата обращения: 21.12.2020).
4. О C/C++ Refences – All C Functions [Электронный ресурс]. – URL:

https://doc.bccnsoft.com/docs/cppreference\_en/all\_c\_functions.html(дата обращения: 13.12.2020).

**приложение 1. Руководство пользователя**

Программа «Калькулятор».

Эта программа последовательно запрашивает три числа: первый аргумент, второй аргумент и номер операции, которую следует выполнить над введенными числами. Перед вводом данных пользователю предоставляется инструкция. В случае ввода некорректных значений программа выдаст ошибку, запрашивая повторный ввод числа, что обеспечивает интуитивно понятное взаимодействие с калькулятором.

Минимальные требования:

* Операционная система: Windows 7 и вся версия после;
* Процессор: 1 (ГГц) или быстрее, с 2 или большим количеством ядер на совместимом 64-разрядном процессоре (32). Процессор на компьютере будет основным фактором, определяющим запуск Windows 7 и вся версия после;
* ОЗУ: 2 гигабайта (ГБ);
* 32 ГБ или большее хранилище;
* Требуемое свободное место на жёстком диске: не менее 1 Мб;
* Отображения: дисплей высокой четкости (720p), который больше 9” по диагонали, 8 бит на цветной канал;
* Среда разработки, например, Clion последней версии, Microsoft Visual Studio и другие;

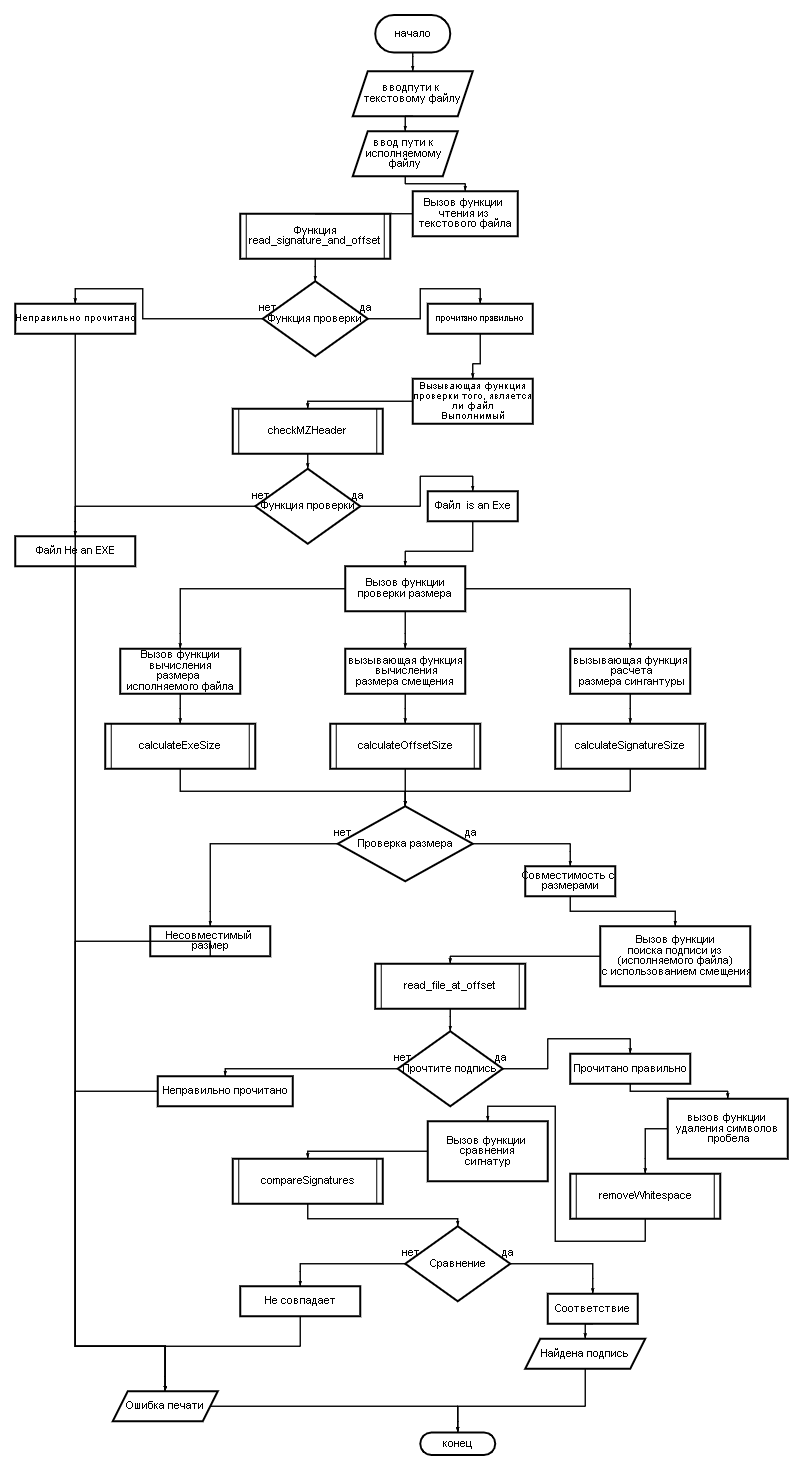
1. Запуск программы.

Запустите файл main.exe. Для запуска программы не требуется установка, нужно запустить исполняемый файл main.exe, скопировав его в любую директорию.

1. Работа с программой «Калькулятор».

После открытия, программа выводит приветственное окно с инструкциями. Пользователь вводит первое число, затем второе число и выбирает арифметическую операцию из предложенного списка, вводя ее в консоль. После этого программа выводит результат выполненной операции. В случае ввода некорректных значений или возникновения ошибок, программа сообщает об ошибке, предлагая ввести значение заново, либо завершает свою работу. Номер ошибки можно увидеть в строке «process returned «номер ошибки или код успешного выполнения». Пользователь несет ответственность за результат работы программы в случае несоблюдения правил из «Руководства пользователя».

**приложение 2. блок-схема алгоритма**

****

**приложение 3. Исходный код программы**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAX\_SIGNATURE\_LENGTH 8

#define MAX\_LENGTH 100

#define SIGNATURE\_SIZE 8

//function to read the offset and the signature and the name from the text file

int read\_signature\_and\_offset(const char \*file\_name, char \*signature, char \*offset, char \*name) {

if (file\_name == NULL){

return 1;

}

if(signature== NULL){

return 2;

}

if (name == NULL){

return 3;

}

FILE \*file = fopen(file\_name, "r");

if (file == NULL) {

int CheackPrint = printf("Error opening file.\n");

if (CheackPrint<0)

{

printf("error of printing\n");

return -1;

}

return 4;

}

// Read the first line (signature)

if (fgets(signature, MAX\_LENGTH, file) == NULL) {

int CheackPrint = printf("Error reading signature.\n");

if (CheackPrint<0)

{

printf("error of printing\n");

return -2;

}

if (fclose(file) != 0) { // Attempt to close the file

printf("Error closing the file.\n");

return -3;

}

return 5;

}

// Remove newline character if present

if (signature[strlen(signature) - 1] == '\n')

signature[strlen(signature) - 1] = '\0';

// Read the second line (offset)

if (fgets(offset, MAX\_LENGTH, file) == NULL) {

int CheackPrint = printf("Error reading offset.\n");

if (CheackPrint<0)

{

printf("error of printing\n");

return -5;

}

if (fclose(file) != 0) { // Attempt to close the file

printf("Error closing the file.\n");

return -6;

}

return 6;

}

// Remove newline character if present

if (offset[strlen(offset) - 1] == '\n')

offset[strlen(offset) - 1] = '\0';

// Read the third line (offset)

if (fgets(name, MAX\_LENGTH, file) == NULL) {

int CheackPrint =printf("Error reading offset.\n");

if (CheackPrint<0)

{

printf("error of printing\n");

return -7;

}

if (fclose(file) != 0) { // Attempt to close the file

printf("Error closing the file.\n");

return -8;

}

return 7;

}

// Remove newline character if present

if (offset[strlen(offset) - 1] == '\n')

offset[strlen(offset) - 1] = '\0';

if (fclose(file) != 0) { // Attempt to close the file

printf("Error closing the file.\n");

return -9;

}

return 0;

}

// Function to calculate the size of the executable file

long calculateExeSize(const char \*file\_path) {

if (file\_path == NULL){

return 1;

}

FILE \*file = fopen(file\_path, "rb");

if (file == NULL) {

int CheackPrint = printf("Error opening file '%s'.\n", file\_path);

if (CheackPrint<0)

{

printf("error of printing\n");

return -1;

}

}

// Seek to the end of the file to get its size

fseek(file, 0, SEEK\_END);

long file\_size = ftell(file);

// Close the file

if (fclose(file) != 0) { // Attempt to close the file

printf("Error closing the file.\n");

return -2;

}

return file\_size;

}

// Function to calculate the size of the offset

size\_t calculateOffsetSize(unsigned long long int offset) {

size\_t size = 0;

while (offset != 0) {

offset /= 16; // Assuming hexadecimal offset, divide by 16

size++;

}

return size;

}

// Function to calculate the size of the signature

size\_t calculateSignatureSize(const char \*signature) {

if (signature == NULL){

return 1;

} else {

size\_t size = 0;

// Iterate through the signature until a null terminator is encountered

while (signature[size] != '\0') {

size++;

}

return size;

}

}

//function of cheacking with the file size

int check\_file\_size(int exe\_file\_size, int offset, int signature\_size) {

if (exe\_file\_size < 0) {

int CheackPrint = printf("ERROR: Invalid executable file size.\n");

if (CheackPrint<0)

{

printf("error of printing\n");

return -1;

}

exit(1) ;

}

if (offset < 0) {

int CheackPrint = printf("\nERROR: Invalid offset.\n");

if (CheackPrint<0)

{

printf("error of printing\n");

exit(-2);

}

return 2;

}

if (signature\_size <= 0) {

int CheackPrint = printf("\nERROR: Invalid signature size.\n");

if (CheackPrint<0)

{

printf("error of printing\n");

return -3;

}

exit(3) ;

}

if (exe\_file\_size < offset + signature\_size) {

int CheackPrint = printf("\nERROR: Offset and signature size exceed file size.\n");

if (CheackPrint<0)

{

printf("error of printing\n");

return -4;

}

exit(4) ;

}

return 0;

}

char \*read\_signature\_from\_exe(const char \*file\_path, long offset) {

if(file\_path == NULL) {

printf("The first argument is NULL");

exit(1);

}

FILE \*file = fopen(file\_path, "rb");

if (file == NULL) {

printf("Unable to open file '%s'.\n", file\_path);

exit(2);

}

// Seek to the specified offset

if (fseek(file, offset, SEEK\_SET) != 0) {

printf("Error seeking to offset.\n");

fclose(file);

return NULL;

}

// Read 8 bytes (64 bits) from the file

unsigned char \*data = (unsigned char \*)malloc(8);

if (data == NULL) {

printf("Memory allocation failed.\n");

fclose(file);

return NULL;

}

size\_t bytes\_read = fread(data, 1, 8, file);

if (bytes\_read != 8) {

printf("Error reading signature from file.\n");

fclose(file);

free(data);

return NULL;

}

if (fclose(file) != 0) { // Attempt to close the file

printf("Error closing the file.\n");

free(data);

return NULL;

}

// Store the first 8 bytes as a char array

char \*signatureFromExe = (char \*)malloc(8 \* 3 + 1); // Each byte represented by 2 characters + 1 space, plus 1 for null terminator

if (signatureFromExe == NULL) {

printf("Memory allocation failed.\n");

free(data);

return NULL;

}

for (int i = 0; i < 8; ++i) {

sprintf(&signatureFromExe[i \* 3], "%02X ", data[i]);

}

free(data);

return signatureFromExe;

}

// Function to check for the presence of an MZ header in a file

int checkMZHeader(const char \*filename) {

if(filename == NULL) {

return 1;

}

FILE \*file = fopen(filename, "rb");

if (file == NULL) {

perror("Error opening file");

return -1;

}

// Read the first two bytes of the file

unsigned char header[2];

if (fread(header, 1, 2, file) != 2) {

printf("Error reading file.\n");

fclose(file);

return -2;

}

if (fclose(file) != 0) { // Attempt to close the file

printf("Error closing the file.\n");

return -3;

}

// Check for MZ header (ASCII characters 'M' (0x4D) followed by 'Z' (0x5A))

if (header[0] == 0x4D && header[1] == 0x5A) {

return 1; // Return 1 to indicate MZ header found

} else {

printf("\nThe file '%s' does not have an MZ header, possibly not a Windows executable.\n", filename);

exit(0); // Return 0 to indicate MZ header not found

}

}

// Function to prepare for signature verification by calculating sizes and checking file integrity

int prepareSignatureVerification(const char \*filepathToScan, const char \*offset, const char \*signature) {

if (filepathToScan == NULL){

return 1;

}

if (offset == NULL){

return 2;

}

if (signature == NULL){

return 3;

}

// Convert offset to numeric value

long offsetValue = strtol(offset, NULL, 16);

if (offsetValue == 0 && errno != 0) {

return 4; // Error converting offset to numeric value

}

// Calculate sizes

long exe\_size = calculateExeSize(filepathToScan);

if (exe\_size < 0) {

return 5; // Error calculating exe size

}

unsigned long long int offsetSize = offsetValue;

size\_t offset\_size = calculateOffsetSize(offsetSize);

if (offset\_size == 0) {

return 6; // Error calculating offset size

}

size\_t signature\_size = calculateSignatureSize(signature);

if (signature\_size == 0) {

return 7; // Error calculating signature size

}

// Check file integrity

long result = check\_file\_size(exe\_size, offset\_size, signature\_size);

if (result != 0) {

return 8; // File integrity check failed

}

return 0; // Success

}

char\* removeWhitespace(const char\* str) {

if (str == NULL){

exit(1);

}

// Allocate memory for the new string

size\_t len = strlen(str);

char\* result = (char\*)malloc(len + 1); // +1 for null terminator

if (result == NULL) {

printf("Memory allocation failed.\n");

exit(EXIT\_FAILURE);

}

// Copy non-whitespace characters to the new string

size\_t j = 0;

for (size\_t i = 0; i < len; ++i) {

if (!isspace((unsigned char)str[i])) {

result[j++] = str[i];

}

}

result[j] = '\0'; // Null-terminate the string

return result;

}

bool compareSignatures(const char \*signature1, const char \*signature2, int size) {

if(signature1 == NULL){

return 1;

}

if(signature2 == NULL){

return 2;

}

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

if (signature1[i] != signature2[i]) {

return false;

}

}

return true;

}

int main() {

char SignatureTxt[MAX\_SIGNATURE\_LENGTH + 1]; // Signature from text file

char filepath[MAX\_LENGTH]; // File path for the signature file

char filepathToScan[MAX\_LENGTH]; // File path for the executable

char NameFile[MAX\_LENGTH]; // Name of the file (from the text file)

char offset[MAX\_LENGTH]; // Offset from text file

char \*signatureFromExe; // Signature from executable

int CheackPrint;

char \*ScanCheck;

char Signatureaxe[MAX\_SIGNATURE\_LENGTH + 1]; // Signature from text file

// Prompt the user to input the file path for the signature file

CheackPrint = printf("Please enter the path of the file containing the hexadecimal signature: \n");

if (CheackPrint < 0) {

printf("Error printing.\n");

return 1;

}

ScanCheck = fgets(filepath, sizeof(filepath), stdin);

if (ScanCheck == NULL) {

printf("Error: Unable to read user input.\n");

return 2;

}

filepath[strcspn(filepath, "\n")] = 0; // Remove trailing newline

// Prompt the user to input the file path for the executable

CheackPrint = printf("Please enter the path of the executable: \n");

if (CheackPrint < 0) {

printf("Error printing.\n");

return 1;

}

ScanCheck = fgets(filepathToScan, sizeof(filepathToScan), stdin);

if (ScanCheck == NULL) {

printf("Error: Unable to read user input.\n");

return -1;

}

filepathToScan[strcspn(filepathToScan, "\n")] = 0; // Remove trailing newline

// Read the signature and offset from the file

if (read\_signature\_and\_offset(filepath, SignatureTxt, offset, NameFile) != 0) {

return -2;

}

// Check for MZ header in the executable file

if (checkMZHeader(filepathToScan) != 1) {

return -3;

}

// Convert offset to numeric value

long offsetValue = strtol(offset, NULL, 16);

if (offsetValue == 0 && errno != 0) {

CheackPrint = printf("Error converting offset to numeric value.\n");

if (CheackPrint < 0) {

printf("Error printing.\n");

return 3;

}

return -4;

}

// size cheacking

if (prepareSignatureVerification(filepathToScan, offset, SignatureTxt) != 0) {

return -5;

}

// Read signature from the executable at the specified offset

signatureFromExe = read\_signature\_from\_exe(filepathToScan, offsetValue);

if (signatureFromExe == NULL) {

CheackPrint = printf("Error reading signature from executable.\n");

if (CheackPrint < 0) {

printf("Error printing.\n");

return 4;

}

return -6;

}

strcpy(Signatureaxe, signatureFromExe);

if (strlen(Signatureaxe) == 0) {

CheackPrint = printf("Error: strcpy failed to copy the signature.\n");

if (CheackPrint < 0) {

printf("Error printing.\n");

return 5;

}

}

char\* strippedStr;

strippedStr = removeWhitespace(Signatureaxe);

if ( strippedStr == NULL) {

return -7;

}

// Free allocated memory

free(signatureFromExe);

bool signaturesMatch = compareSignatures(SignatureTxt, strippedStr, SIGNATURE\_SIZE);

// size cheacking

if (signaturesMatch != 0 && signaturesMatch !=1) {

return -5;

}

if (signaturesMatch) {

CheackPrint = printf("%s is safe!",NameFile);

if (CheackPrint < 0) {

printf("Error printing.\n");

return 6;

}

} else {

CheackPrint = printf("File is not safe!\n");

if (CheackPrint < 0) {

printf("Error printing.\n");

return 7;

}

}

return 0;

}