**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра информационной безопасности**

Курсовая РАБОТА

**по дисциплине «Программирование»**

**Тема работы: Сигнатурный сканер**

|  |  |  |
| --- | --- | --- |
| Студент гр. 8363 |  | Панфилович А.И. |
| Преподаватель |  | Халиуллин Р.А. |

Санкт-Петербург

2019

**ЗАДАНИЕ на курсовую работу**

Студент Панфилович А.И.

Группа 8363

Тема работы: Сигнатурный сканер

Исходные данные:

Разработать на языке программирования C или C++ сигнатурный сканер. Сигнатурный сканер должен сканировать файлы и обнаруживать заданные образцы по наличию сигнатуры в файле. При поиске сигнатуры в файле необходимо учитывать формат файла. Пути к файлам для сканирования вводятся пользователем. Если сигнатура обнаружена в файле, то необходимо вывести имя файла с указанием того, какому образцу соответствует найденная сигнатура. Информация о сигнатурах должна храниться в отдельном файле и считываться сканером при запуске. Приложение должно иметь консольный или графический интерфейс, по выбору студента. Приложение должно корректно обрабатывать ошибки, в том числе ошибки ввода/вывода, выделения/освобождения памяти.

Содержание пояснительной записки:

Введение, теоретическая часть, термины и определения, принцип игры, постановка задачи, алгоритмы решения, реализация программы, описание реализации, результаты тестирования, заключение, список используемых источников, приложение 1 — блок–схема, приложение 2 — руководство пользователя, приложение 3 — исходный код.

Предполагаемый объем пояснительной записки:   
Не менее 30 страниц.

Дата выдачи задания: 07.02.2019

Дата сдачи реферата: 31.05.2019

Дата защиты реферата: 06.06.2019

|  |  |  |
| --- | --- | --- |
| Студент гр. 8363 |  | Панфилович А.И. |
| Преподаватель |  | Халиуллин Р.А. |

**Аннотация**

Задача курсовой работы – реализация сигнатурного сканера на языке программирования C++. В сигнатурном сканере должна быть реализована поддержка формата PE (Portable executable). Путь к сканируемым файлам вводится пользователем. Если сигнатура обнаружена в файле, то необходимо вывести имя файла с указанием того, что сигнатура этого файла соответствует указанной сигнатуры. Информация о сигнатуре должна храниться в отдельном файле и считываться сканером при запуске. Приложение должно корректно обрабатывать ошибки ввода/вывода. Основной функционал должен быть реализован с помощью собственных функций.

**Summary**

The task of the course work - the implementation of a signature scanner in the C ++ programming language. The signature scanner must support the PE (Portable executable) format. The path to the scanned files is entered by the user. If the signature is found in the file, then you must output the file name with the indication that the signature of this file matches the specified signature. Information about the signature should be stored in a separate file and read by the scanner at startup. The application must correctly handle I / O errors. The main functionality should be implemented using eigenfunctions.

**содержание**

|  |  |
| --- | --- |
| Введение | 5 |
| 1. Теоретическая часть | 6 |
| * 1. Методы обнаружения вирусов | 6 |
| * + 1. Метод обнаружения изменений | 6 |
| * + 1. Эвристический анализ | 7 |
| * + 1. Метод использования резидентных сторожей | 7 |
| * + 1. Вакцинирование программ | 8 |
| * + 1. Аппаратно-программная защита от вирусов | 8 |
| * 1. Компьютерный вирус | 9 |
| * + 1. Классификация вирусов | 9 |
| * + 1. Распространение вирусов | 10 |
| * 1. Информация о PE-файлах | 12 |
| * + 1. Сигнатура | 13 |
| * + 1. Структура | 13 |
| * + 1. Таблица импорта | 13 |
| * + 1. Таблица экспорта | 14 |
| * + 1. Таблица перемещений | 14 |
| 1. Реализация программы | 16 |
| * 1. Использованное программное обеспечение | 16 |
| * 1. Описание реализации | 16 |
| 1. Результаты тестирования | 20 |
| Заключение | 22 |
| Список использованных источников | 23 |
| Приложение 1. Блок-схема | 24 |
| Приложение 2. Руководство пользователя | 25 |
| Приложение 3. Исходный код | 29 |

**введение**

Сканирование - один из самых простых методов обнаружения вирусов. Сканирование осуществляется программой-сканером, которая просматривает файлы в поисках опознавательной части вируса - сигнатуры. Программа фиксирует наличие уже известных вирусов, за исключением полиморфных вирусов, которые применяют шифрование тела вируса, изменяя при этом каждый раз и сигнатуру. Программы-сканеры могут хранить не сигнатуры известных вирусов, а их контрольные суммы и могут удалять обнаруженные вирусы. Такие программы называются полифагами.

Метод сканирования применим для обнаружения вирусов, сигнатуры которых уже выделены и являются постоянными. Для эффективного использования метода необходимо регулярное обновление сведений о новых вирусах.

Данной программой вы сможете находить вредоносные программы зная их сигнатуры.

1. **ТЕОРЕТИЧЕСКАЯ ЧАСТЬ**
   1. **Методы обнаружения вирусов**

Известны ещё несколько методов обнаружения вирусов кроме сканирования сигнатуры, например, обнаружение изменений, эвристический анализ, использование резидентных сторожей, вакцинирование программ и аппаратно-программная защита от вирусов.

##### Метод обнаружения изменений

Методобнаруженияизменений базируется на использовании программ-ревизоров. Эти программы определяют и запоминают характеристики всех областей на дисках, в которых обычно размещаются вирусы. При периодическом выполнении программ-ревизоров сравниваются хранящиеся характеристики и характеристики, получаемые при контроле областей дисков. По результатам ревизии программа выдает сведения о предположительном наличии вирусов.

Обычно программы-ревизоры запоминают в специальных файлах образы главной загрузочной записи, загрузочных секторов логических дисков, характеристики всех контролируемых файлов, каталогов и номера дефектных кластеров. Могут контролироваться также объем установленной оперативной памяти, количество подключенных к компьютеру дисков и их параметры.

Главным достоинством метода является возможность обнаружения вирусов всех типов, а также новых неизвестных вирусов. Совершенные программы-ревизоры обнаруживают даже «стелс»-вирусы. Например, программа-ревизор Аdinf, разработанная Д. Ю. Мостовым, работает с диском непосредственно по секторам через Bios. Это не позволяет использовать «стелс»-вирусам возможность перехвата прерываний и «подставки» для контроля нужной вирусу области памяти.

Имеются у этого метода и недостатки. С помощью программ-ревизоров невозможно определить вирус в файлах, которые поступают в систему уже зараженными. Вирусы будут обнаружены только после размножения в системе.

Программы-ревизоры непригодны для обнаружения заражения макровирусами, так как документы и таблицы очень часто изменяются.

##### Эвристический анализ

Эвристический анализ сравнительно недавно начал использоваться для обнаружения вирусов. Как и метод обнаружения изменений, данный метод позволяет определять неизвестные вирусы, но не требует предварительного сбора, обработки и хранения информации о файловой системе.

Сущность эвристического анализа заключается в проверке возможных сред обитания вирусов и выявление в них команд (групп команд), характерных для вирусов. Такими командами могут быть команды создания резидентных модулей в оперативной памяти, команды прямого обращения к дискам, минуя ОС. Эвристические анализаторы при обнаружении «подозрительных» команд в файлах или загрузочных секторах выдают сообщение о возможном заражении. После получения таких сообщений необходимо тщательно проверить предположительно зараженные файлы и загрузочные сектора всеми имеющимися антивирусными средствами. Эвристический анализатор имеется, например, в антивирусной программе Doctor Web.

* + 1. **Метод использования резидентных сторожей**

Метод использования резидентных сторожей основан на применении программ, которые постоянно находятся в ОП ЭВМ и отслеживают все действия остальных программ.

В случае выполнения какой-либо программой подозрительных действий (обращение для записи в загрузочные сектора, помещение в ОП резидентных модулей, попытки перехвата прерываний и т. п.) резидентный сторож выдает сообщение пользователю. Программа-сторож может загружать на выполнение другие антивирусные программы для проверки «подозрительных» программ, а также для контроля всех поступающих извне файлов (со сменных дисков, по сети).

Существенным недостатком данного метода является значительный процент ложных тревог, что мешает работе пользователя, вызывает раздражение и желание отказаться от использования резидентных сторожей. Примером резидентного сторожа может служить программа Vsafe, входящая в состав MS DOS.

* + 1. **Вакцинирование программ**

Под вакцинацией программ понимается создание специального модуля для контроля ее целостности. В качестве характеристики целостности файла обычно используется контрольная сумма. При заражении вакцинированного файла, модуль контроля обнаруживает изменение контрольной суммы и сообщает об этом пользователю. Метод позволяет обнаруживать все вирусы, в том числе и незнакомые, за исключением «стелс»-вирусов.

* + 1. **Аппаратно-программная защита от вирусов**

Самым надежным методом защиты от вирусов является использование аппаратно-программных антивирусных средств. В настоящее время для защиты ПЭВМ используются специальные контроллеры и их программное обеспечение. Контроллер устанавливается в разъем расширения и имеет доступ к общей шине. Это позволяет ему контролировать все обращения к дисковой системе. В программном обеспечении контроллера запоминаются области на дисках, изменение которых в обычных режимах работы не допускается. Таким образом, можно установить защиту на изменение главной загрузочной записи, загрузочных секторов, файлов конфигурации, исполняемых файлов и др.

При выполнении запретных действий любой программой контроллер выдает соответствующее сообщение пользователю и блокирует работу ПЭВМ.

Аппаратно-программные антивирусные средства обладают рядом достоинств перед программными:

* работают постоянно;
* обнаруживают все вирусы, независимо от механизма их действия;
* блокируют неразрешенные действия, являющиеся результатом работы вируса или неквалифицированного пользователя.

Недостаток у этих средств один ⸺ зависимость от аппаратных средств ПЭВМ. Изменение последних ведет к необходимости замены контроллера.

Примером аппаратно-программной защиты от вирусов может служить комплекс Sheriff.

* 1. **Компьютерный вирус**

Компьютерный вирус — вид вредоносного программного обеспечения, способного внедряться в код других программ, системные области памяти, загрузочные секторы, а также распространять свои копии по разнообразным каналам связи.

Основная цель вируса — его распространение. Кроме того, часто его сопутствующей функцией является нарушение работы программно-аппаратных комплексов — удаление файлов и даже удаление операционной системы, приведение в негодность структур размещения данных, блокирование работы пользователей и т. п. Даже если автор вируса не запрограммировал вредоносных эффектов, вирус может приводить к сбоям компьютера из-за ошибок, неучтённых тонкостей взаимодействия с операционной системой и другими программами. Кроме того, вирусы, как правило, занимают место на накопителях информации и потребляют ресурсы системы.

В обиходе «вирусами» называют всё вредоносное ПО, хотя на самом деле это лишь один его вид.

* + 1. **Классификация вирусов**

Ныне существует немало разновидностей вирусов, различающихся по основному способу распространения и функциональности. Если изначально вирусы распространялись на дискетах и других носителях, то сейчас доминируют вирусы, распространяющиеся через локальные и глобальные (Интернет) сети. Растёт и функциональность вирусов, которую они перенимают от других видов программ.

В настоящее время не существует единой системы классификации и именования вирусов (хотя попытка создать стандарт была предпринята на встрече CARO в 1991 году). Принято разделять вирусы:

* по поражаемым объектам (файловые вирусы, загрузочные вирусы, сценарные вирусы, макровирусы, вирусы, поражающие исходный код);
* файловые вирусы делят по механизму заражения: паразитирующие добавляют себя в исполняемый файл, перезаписывающие невосстановимо портят заражённый файл, «спутники» идут отдельным файлом.
* по поражаемым операционным системам и платформам (DOS, Windows, Unix, Linux, Android);
* по технологиям, используемым вирусом (полиморфные вирусы, стелс-вирусы, руткиты);
* по языку, на котором написан вирус (ассемблер, высокоуровневый язык программирования, сценарный язык и др.);
* по дополнительной вредоносной функциональности (бэкдоры, кейлоггеры, шпионы, ботнеты и др.).
  + 1. **Распространение вирусов**

Вирусы распространяются, копируя своё тело и обеспечивая его последующее исполнение: внедряя себя в исполняемый код других программ, заменяя собой другие программы, прописываясь в автозапуск через реестр и другое. Вирусом или его носителем могут быть не только программы, содержащие машинный код, но и любая информация, содержащая автоматически исполняемые команды, — например, пакетные файлы и документы Microsoft Word и Excel, содержащие макросы. Кроме того, для проникновения на компьютер вирус может использовать уязвимости в популярном программном обеспечении (например, Adobe Flash, Internet Explorer, Outlook), для чего распространители внедряют его в обычные данные (картинки, тексты и т. д.) вместе с эксплойтом, использующим уязвимость.

После того как вирус успешно внедрился в коды программы, файла или документа, он будет находиться в состоянии сна, пока обстоятельства не заставят компьютер или устройство выполнить его код. Чтобы вирус заразил ваш компьютер, необходимо запустить заражённую программу, которая, в свою очередь, приведёт к выполнению кода вируса. Это означает, что вирус может оставаться бездействующим на компьютере без каких-либо симптомов поражения. Однако, как только вирус начинает действовать, он может заражать другие файлы и компьютеры, находящиеся в одной сети. В зависимости от целей программиста-вирусописателя, вирусы либо причиняют незначительный вред, либо имеют разрушительный эффект, например, удаление данных или кража конфиденциальной информации.

Существуют различные каналы распространения вирусов:

* Дискеты. Самый распространённый канал заражения в 1980—1990-е годы. Сейчас практически отсутствует из-за появления более распространённых и эффективных каналов и отсутствия флоппи-дисководов на многих современных компьютерах.
* Флеш-накопители («флешки»). В настоящее время USB-накопители заменяют дискеты и повторяют их судьбу — большое количество вирусов распространяется через съёмные накопители, включая цифровые фотоаппараты, цифровые видеокамеры, портативные цифровые плееры, а с 2000-х годов всё большую роль играют мобильные телефоны, особенно смартфоны (появились мобильные вирусы). Использование этого канала ранее было преимущественно обусловлено возможностью создания на накопителе специального файла autorun.inf, в котором можно указать программу, запускаемую проводником Windows при открытии такого накопителя. В Windows 7 возможность автозапуска файлов с переносных носителей была отключена.
* Электронная почта. Обычно вирусы в письмах электронной почты маскируются под безобидные вложения: картинки, документы, музыку, ссылки на сайты. В некоторых письмах могут содержаться действительно только ссылки, то есть в самих письмах может и не быть вредоносного кода, но если открыть такую ссылку, то можно попасть на специально созданный веб-сайт, содержащий вирусный код. Многие почтовые вирусы, попав на компьютер пользователя, затем используют адресную книгу из установленных почтовых клиентов типа Outlook для рассылки самого себя дальше.
* Системы обмена мгновенными сообщениями. Здесь также распространена рассылка ссылок на якобы фото, музыку либо программы, в действительности, являющиеся вирусами, по ICQ и через другие программы мгновенного обмена сообщениями.
* Веб-страницы. Возможно также заражение через страницы Интернета ввиду наличия на страницах всемирной паутины различного «активного» содержимого: скриптов, ActiveX-компонент. В этом случае используются уязвимости программного обеспечения, установленного на компьютере пользователя, либо уязвимости в ПО владельца сайта (что опаснее, так как заражению подвергаются добропорядочные сайты с большим потоком посетителей), а ничего не подозревающие пользователи, зайдя на такой сайт, рискуют заразить свой компьютер.
* Интернет и локальные сети (черви). Черви — вид вирусов, которые проникают на компьютер-жертву без участия пользователя. Черви используют так называемые «дыры» (уязвимости) в программном обеспечении операционных систем, чтобы проникнуть на компьютер. Уязвимости — это ошибки и недоработки в программном обеспечении, которые позволяют удалённо загрузить и выполнить машинный код, в результате чего вирус-червь попадает в операционную систему и, как правило, начинает действия по заражению других компьютеров через локальную сеть или Интернет. Злоумышленники используют заражённые компьютеры пользователей для рассылки спама или для DDoS-атак.
  1. **Информация о PE-файлах**

Portable Executable (PE, «переносимый исполняемый») — формат исполняемых файлов, объектного кода и динамических библиотек, используемый в 32- и 64-разрядных версиях операционной системы Microsoft Windows. Формат PE представляет собой структуру данных, содержащую всю информацию, необходимую PE-загрузчику для отображения файла в память. Исполняемый код включает в себя ссылки для связывания динамически загружаемых библиотек, таблицы экспорта и импорта API-функций, данные для управления ресурсами и данные локальной памяти потока ([TLS](https://en.wikipedia.org/wiki/Thread-local_storage)). В операционных системах семейства Windows NT формат PE используется для EXE, DLL, SYS (драйверов устройств) и других типов исполняемых файлов.

* + 1. **Сигнатура**

Первые 2 байта PE-файла содержат сигнатуру 0x4D 0x5A — «MZ» (как наследник MZ-формата). Далее - двойное слово по смещению 0x3C содержит адрес PE-заголовка. Последний начинается с сигнатуры 0x50 0x45 — «PE».

* + 1. **Структура**

Файл PE состоит из нескольких заголовков и секций, которые указывают динамическому компоновщику, как отображать файл в память. Исполняемый образ состоит из нескольких различных областей (секций), каждая из которых требует различных прав доступа к памяти; таким образом, начало каждой секции должно быть выровнено по границе страницы. Например, обычно секция .text, которая содержит код программы, отображена как исполняемая/доступная только для чтения, а секция .data, содержащая глобальные переменные, отображена как неисполняемая/доступная для чтения и записи. Однако, чтобы не тратить впустую пространство на жёстком диске, различные секции на нём на границу страницы не выровнены. Часть работы динамического компоновщика состоит в том, чтобы отобразить каждую секцию в память отдельно и присвоить корректные права доступа получившимся областям согласно указаниям, содержащимся в заголовках.

* + 1. **Таблица импорта**

Одна из известных секций — таблица адресов импорта (IAT — Import Address Table), которая используется в качестве таблицы поиска, когда приложение вызывает функцию из другого модуля. Это может быть сделано и в форме импорта по порядковому номеру функции (ordinal), и импорта по её имени. Поскольку скомпилированной программе неизвестно расположение библиотек, от которых она зависит, то требуется производить косвенный переход всякий раз, когда происходит вызов API-функции. Когда динамический компоновщик загружает модули и объединяет их, он записывает действительные адреса в область IAT так, чтобы они указали на ячейки памяти соответствующих библиотечных функций. Хотя это добавляет дополнительный переход внутри модуля, приводящий к потере производительности, это предоставляет ключевое преимущество: количество страниц памяти, которые должны быть скопированы загрузчиком при записи, минимизировано, что приводит к экономии памяти и дискового времени ввода-вывода. Если компилятору будет известно заранее, что вызов будет межмодульным (через атрибут dllimport), то он сможет произвести более оптимизированный код, который просто приводит к коду операции косвенного вызова.

* + 1. **Таблица экспорта**

Таблица адресов экспорта (EAT — Export Address Table) нужна для того, чтобы один модуль (обычно это динамически загружаемая библиотека) мог указать другим модулям, какие функции они могут из него импортировать, и по каким адресам последние расположены.

* + 1. **Таблица перемещений**

Файлы PE не содержат позиционно-независимого кода. Вместо этого они скомпилированы для предпочтительного базового адреса, и все адреса, генерируемые компилятором/компоновщиком, заранее фиксированы. Если PE-файл не может быть загружен по своему предпочтительному адресу (потому что он уже занят чем-то ещё), операционная система будет перебазировать его. Это включает в себя перевычисление каждого абсолютного адреса и изменение кода для того, чтобы использовать новые значения. Загрузчик делает это, сравнивая предпочтительный и фактический адреса загрузки, и вычисляя значение разности. Тогда для получения нового адреса ячейки памяти эта разность складывается с предпочтительным адресом. Базовые адреса перемещений хранятся в списке и при необходимости добавляются к существующей ячейке памяти. Полученный код является теперь отдельным по отношению к процессу и не является больше разделяемым, так что при таком способе теряются многие из преимуществ экономии памяти динамически загружаемых библиотек. Такой способ также значительно замедляет загрузку модуля. По этой причине следует избегать перебазирования везде, где это возможно; например, библиотеки, поставляемые Microsoft, имеют предварительно вычисленные неперекрывающиеся базовые адреса. В случае отсутствия необходимости перебазировании PE-файлы имеют преимущество очень эффективного кода, но при наличии перебазирования издержки в использовании памяти могут быть значительными. Это отличает формат PE от ELF, который использует полностью позиционно-независимый код и глобальную таблицу смещений, которая жертвует временем выполнения в пользу расходования памяти.

1. **РЕАЛИЗАЦИЯ ПРОГРАММЫ**
   1. **Использованное программное обеспечение**

Компилятор: TDM-GCC 4.9.2 64-bit Release.

Среда разработки: Dev C++.

* 1. **Описание реализации**

1. Функция SignatureComparison

Функция SignatureComparison выполняет сравнение сигнатуры файла с заданной и вывод сообщения в случае совпадения сигнатур.

Объявление функции: int SignatureComparison(long int SignIndent, char \*FileAdrr, char \*SignName);

Тип функции: int.

Аргументы функции:

long int SignIndent – отступ сигнатуры в файле;

char \*FileAdrr – указатель на символьную строку, в которой указан адрес файла;

char \*SignName ­ – указатель на символьную сроку, в которой указана сигнатура для сравнения с сигнатурами других файлов.

Возвращаемое значение:

• 0 – в случае успешного выполнения;

• 2 – в случае ошибки аргумента char \*FileAdrr;

• 3 – в случае ошибки аргумента char \*SignName;

• 4 – в случае возникновении ошибки при выделении памяти;

• 5 – в случае возникновении ошибки при использовании функции fopen;

• 6 – в случае возникновении ошибки при 1-ом использовании функции feof;

• 7 – в случае возникновении ошибки при 1-ом использовании функции printf;

• 8 – в случае возникновении ошибки при 1-ом использовании функции fclose;

• 9 – в случае возникновении ошибки при 1-ом использовании функции fseek;

• 10 – в случае возникновении ошибки при 2-ом использовании функции feof;

• 11 – в случае возникновении ошибки при 2-ом использовании функции printf;

• 12 – в случае возникновении ошибки при использовании функции system;

• 13 – в случае возникновении ошибки при 2-ом использовании функции fclose;

• 14 – в случае возникновении ошибки при 3-ем использовании функции printf;

• 15 – в случае возникновении ошибки при 3-ем использовании функции fclose.

1. Функция ScanfDerr

Функция ScanfDerr запрашивает адрес директории для последующего сканирования.

Объявление функции: int ScanfDerr(char \*DerName);

Тип функции: int.

Аргументы функции:

char \*DerName – указатель на символьную строку, в которой храниться адрес сканируемой директории.

Возвращаемое значение:

• 0 – в случае успешного выполнения;

• 1 – в случае ошибки аргумента char \*DerName;

• 2 – в случае возникновении ошибки при 1-ом использовании функции printf;

• 3 – в случае возникновении ошибки при использовании функции getchar;

• 4 – в случае возникновении ошибки при использовании функции fflush.

1. Функция ScanSign

Функция ScanSign читает из файла сигнатуру для поиска и её отступ в файле.

Объявление функции: int ScanSign(char \*SignName, long int \*SignIndent);

Тип функции: int.

Аргументы функции:

char \*SignName – указатель на символьную строку, в которой хранится сигнатура;

long int \*SignIndent – указатель на переменную целого типа, в которой хранится отступ сигнатуры в файле.

Возвращаемое значение:

• 0 – в случае успешного выполнения;

• 1 – в случае ошибки аргумента char \*SignName;

• 3 – в случае возникновении ошибки при использовании функции fopen;

• 4 – в случае возникновении ошибки при 1-ом использовании функции fgetc;

• 5 – в случае возникновении ошибки при 2-ом использовании функции fgetc;

• 6 – в случае возникновении ошибки при использовании функции fclose.

1. Функция FindFileInDer

Функция FindFileInDer реализует рекурсивный обход по директории и сравнение сигнатур найденных файлов со сканированной из файла

Объявление функции: int FindFileInDer(char \*DerName, char \*SignName, long int SignIndent);

Тип функции: int

Аргументы функции:

char \*DerName – указатель на символьную строку, в которой храниться адрес сканируемой директории;

char \*SignName ­ – указатель на символьную сроку, в которой указана сигнатура для сравнения с сигнатурами других файлов;

long int SignIndent – отступ сигнатуры в файле.

Возвращаемое значение:

• 0 – в случае успешного выполнения;

• 1 – в случае ошибки аргумента char \*DerName;

• 2 – в случае ошибки аргумента char \*SignName;

• 4 – в случае возникновении ошибки при 1-ом использовании функции printf;

• 5 – в случае возникновении ошибки при 1-ом использовании функции strcat;

• 6 – в случае возникновении ошибки при 2-ом использовании функции strcat;

• 7 – в случае возникновении ошибки при 2-ом использовании функции printf;

• 8 – в случае возникновении ошибки при 3-ем использовании функции strcat;

• 9 – в случае возникновении ошибки при использовании функции FindFileInDer;

• 10 – в случае возникновении ошибки при 4-ом использовании функции strcat;

• 11 – в случае возникновении ошибки при 3-ем использовании функции printf;

• 12 – в случае возникновении ошибки при 5-ом использовании функции strcat;

• 13 – в случае возникновении ошибки при 6-ом использовании функции strcat;

• 14 – в случае возникновении ошибки при использовании функции SignatureComparison.

1. **РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ**

При запуске программы в консоль надо будет ввести адрес нужной вам директории, которую вы хотите просканировать (рисунок 1).

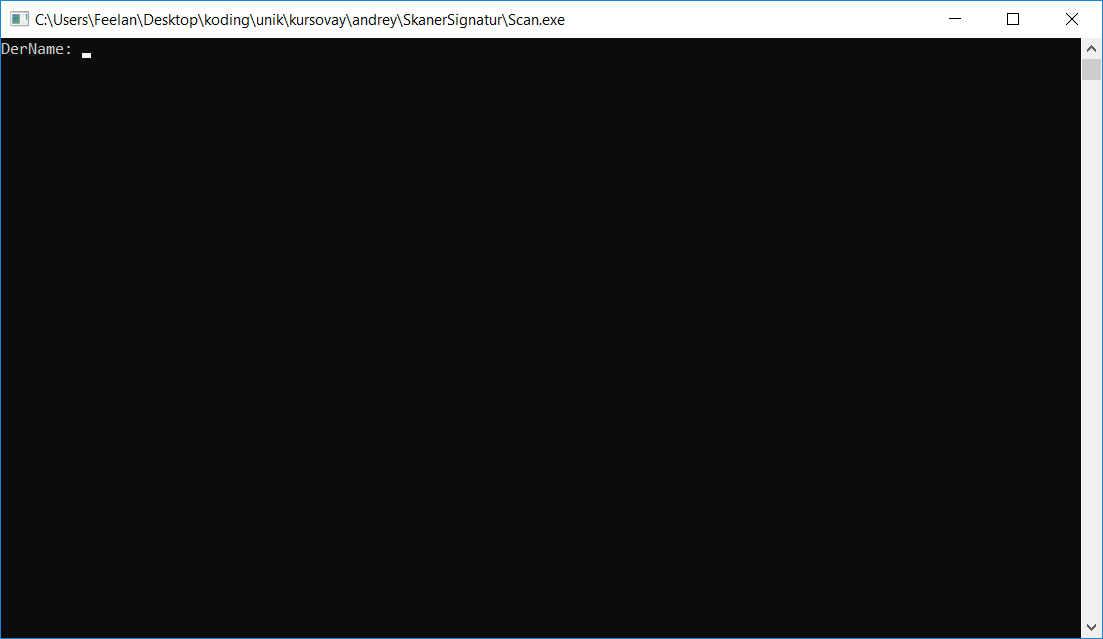


Рисунок 1 — Один из исходов игры

После ввода директории программа выводит сообщение об ошибке в случае, если адрес введён неправильно (рисунок 2).

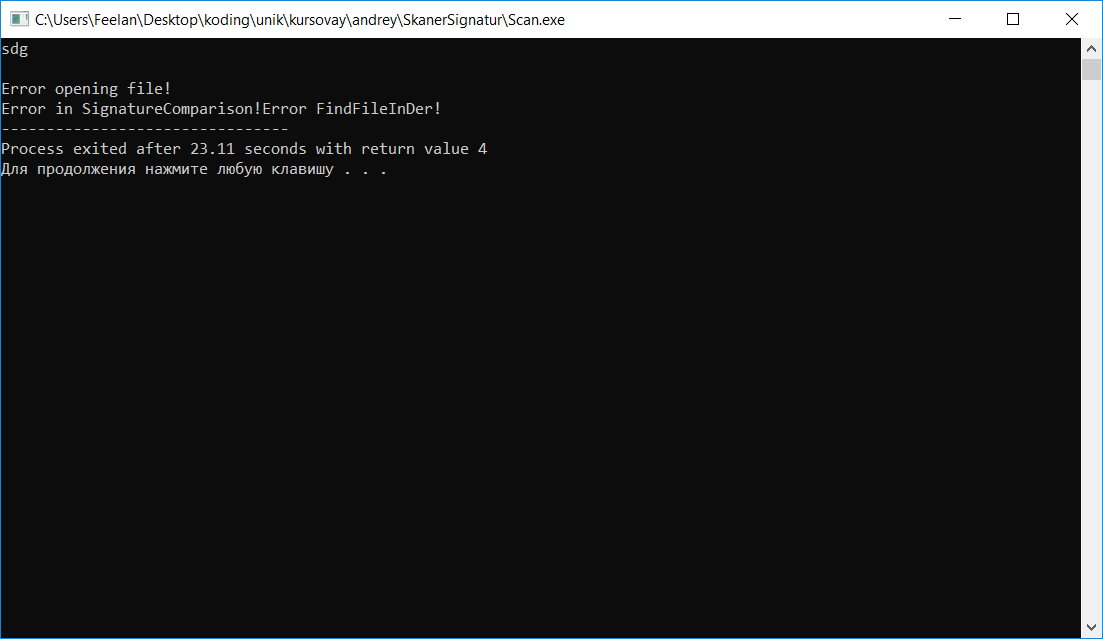


Рисунок 2 — Вывод при вводе неверного адреса директории

При верном вводе адреса будет выводиться список директорий, поддиректорий и файлов. Если сигнатура файла совпадёт с сигнатурой, которая находилась в файле, то вывод списка будет приостановлен, справа от названия файла будет выведено «Danger» и имя сигнатуры, которая была обнаружена в данном файле. Иначе справа без остановки вывода списка файлов будет выведено «Clear» (рисунок 3).

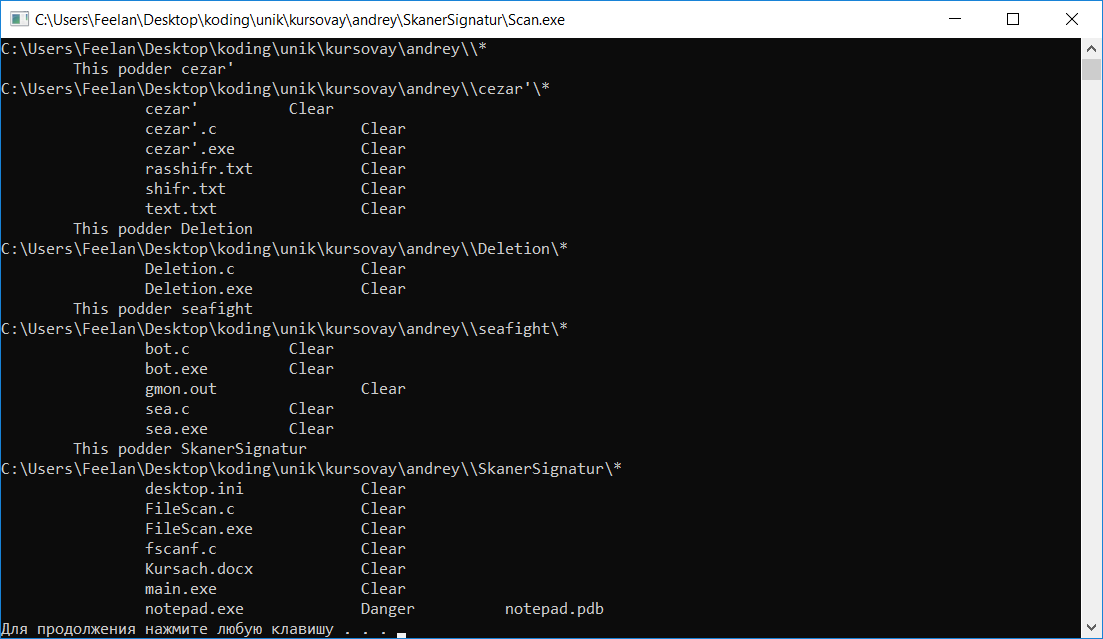


Рисунок 3 — Вывод списка файлов

При завершении сканирования пользователю будет доступен весь список файлов, находящихся в указанной раннее директории и поддиректориях (рисунок 4).

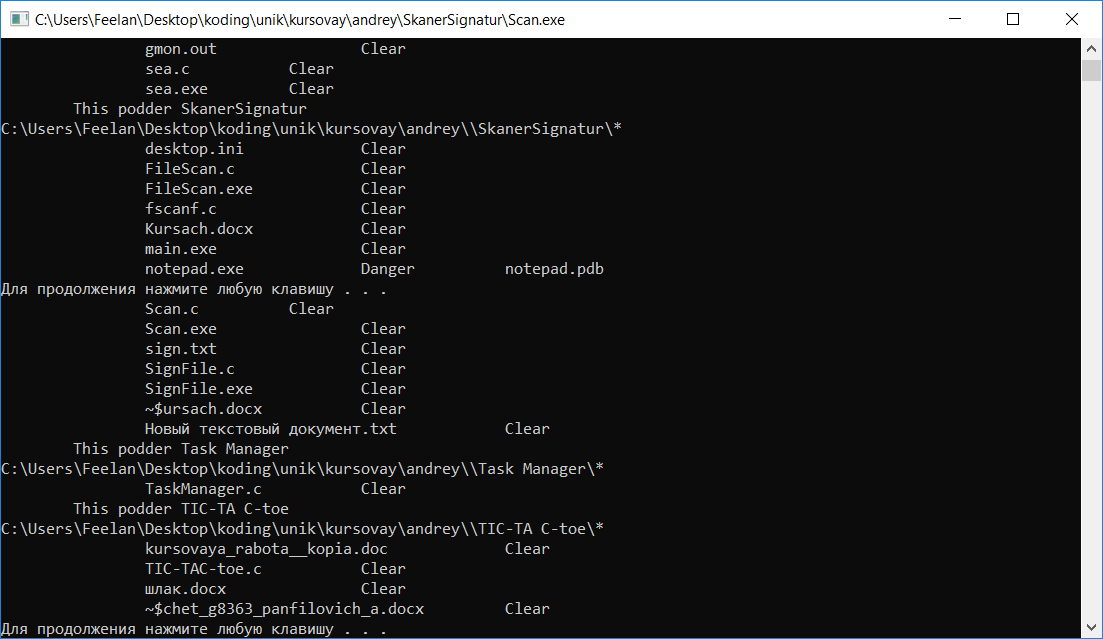


Рисунок 4 — Конец работы программы**заключение**

Используя алгоритм рекурсии по директориям и функции из библиотеки «windows.h», мы смогли осуществить рекурсивный обход по директориям. Зная, как устроены файлы формата PE, мы легко смогли найти исполняемые файлы, сравнить их сигнатуры и выявить «опасные файлы».

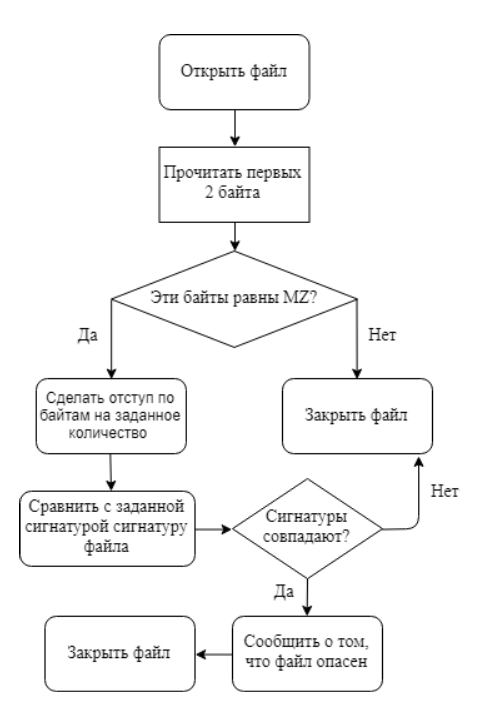
**список использованных источников**

1. Статья о методах обнаружения вирусов по сслыке: http://sumk.ulstu.ru/docs/mszki/Zavgorodnii/10.5.1.html

2. «Язык программирования Си», 2-е издание. Авторы: Брайан Керниган, Деннис Ритчи. Издательство: «Вильямс» ISBN 978-5-8459-0891-9, 2013. 304 с. Переводчик: Перевод с английского и редакция И.В. Берштейна.

3. Сайт: https://ru.wikipedia.org/wiki/Заглавная\_страница

4. Сайт: https://www.microsoft.com/ru-ru/



|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | **Приложение 1. Блок-схема** | | | | | |
|  |  |  |  |  |
| Изм. | Лист | № докум | Подпись | Дата |
| Разраб. | | Панфилович А. |  |  |  | Литера | | | Лист | Листов |
| Пров. | | Халиуллин Р.А |  |  |  | y |  | 1 | 1 |
| Реценз. | | Халиуллин Р.А |  |  |  | | | | |
| Н. Контр. | | Халиуллин Р.А |  |  |
| Утв | | Халиуллин Р.А |  |  |

**Приложение 2. руководство пользователя**

**Краткое описание программы:** данное консольное приложение является примитивным сигнатурным сканером, с помощью которого вы сможете найти файлы в вашей системе по заданной вами сигнатуре.

**Минимальные системные требования:**

**Операционная система:** Windows 10 64 – разрядная операционная система;

**Процессор:** с частотой не ниже 2.3 ГГц;

**Дисплей:** 1920 × 1080;

**Видеоадаптер:** DirectX версии не ниже 9 с драйвером WDDM 1.0;

**ОЗУ:** 8 ГБ;

**Установка программы:** скопировать исполняемый файл Scan.exe и текстовый файл sign.txt в исполняемую директорию.

**Запуск программы:** для запуска сканера необходимо дважды щелкнуть левой кнопкой мыши по файлу Scan.exe.

**Работа с программой:** после запуска исполняемого файла на экране пользователя появится консольное окно с запросом адреса директории, как показано на рисунке 5.

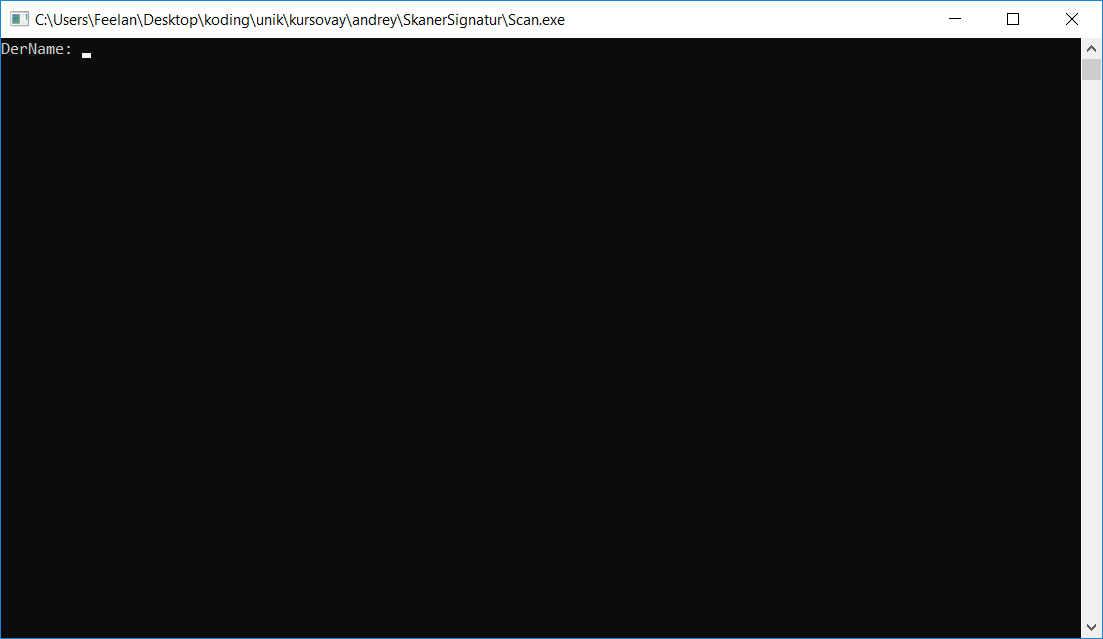


Рисунок 5 — Запрос адреса директории

Ввод сигнатуры проводится следующим образом: вначале идёт имя диска, после «:\», после этого вы вводите дальнейший адрес нужной вам директории, отделяя каждую поддиректорию «\». Дойдя до нужной вам директории в конце адреса должно быть «\\\*» (C:\Users\Feelan\Desktop\koding\unik\kursovay\andrey\\\* или C:\\\*). Если всё введено верно и такая директория существует, то начнётся вывод списка проверенных файлов (рисунок 6).

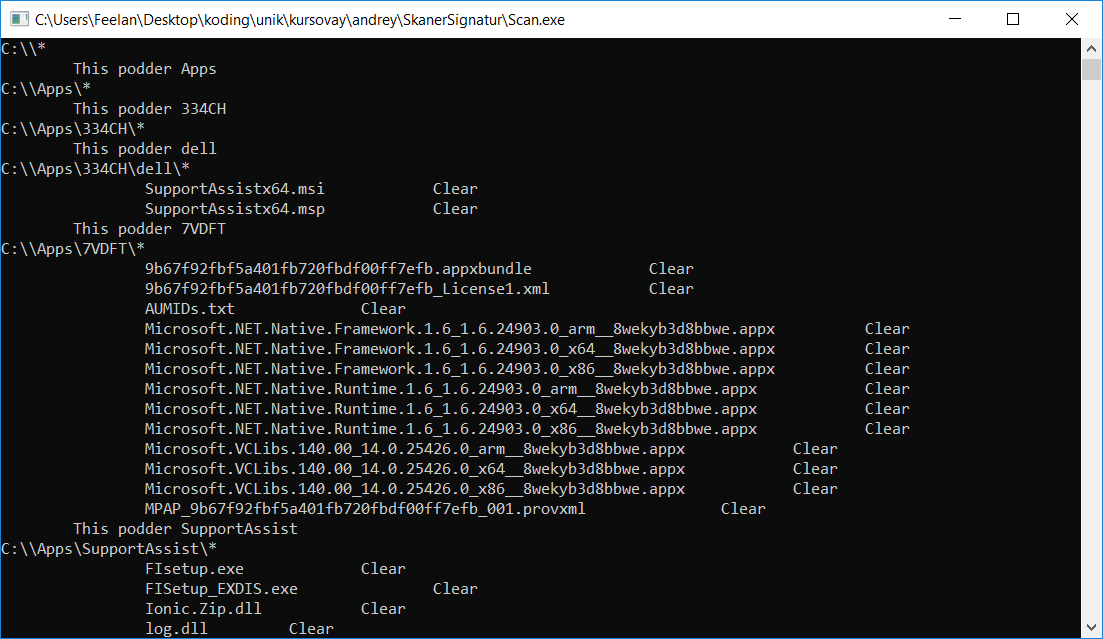


Рисунок 6 — Вывод списка файлов

Иначе будет выведено сообщение об ошибке (рисунок 7).

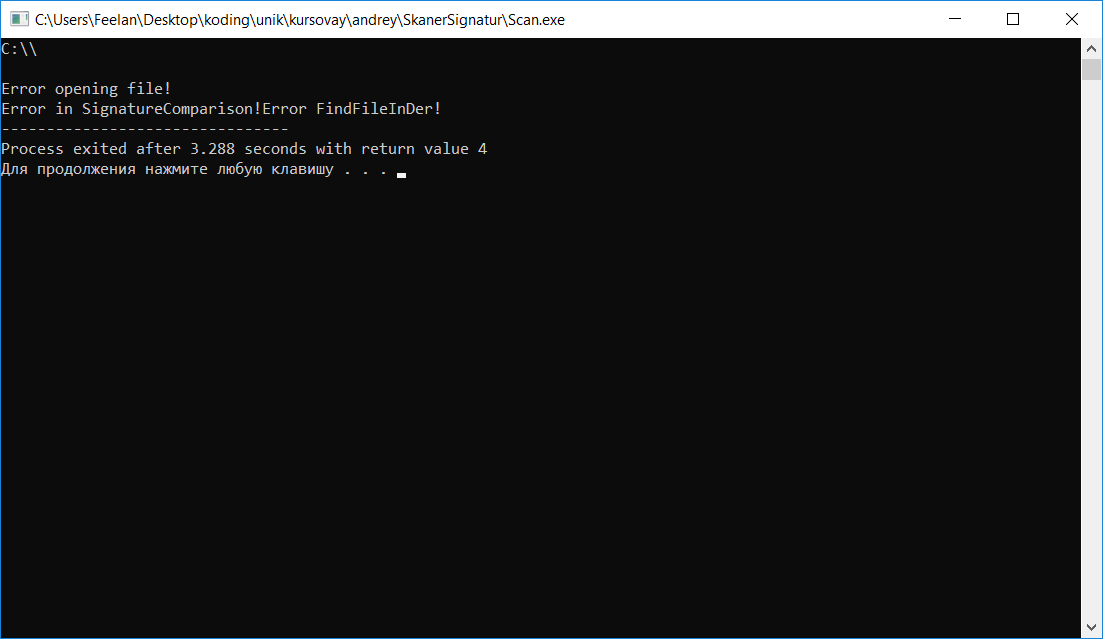


Рисунок 7 — Вывод сообщения об ошибке при неверном адресе директории

При обнаружении угрозы в файле вывод будет приостановлен и справа от имени файла будет выведено «Danger» и имя сигнатуры в сопровождение с звуковым сигналом (рисунок 8).

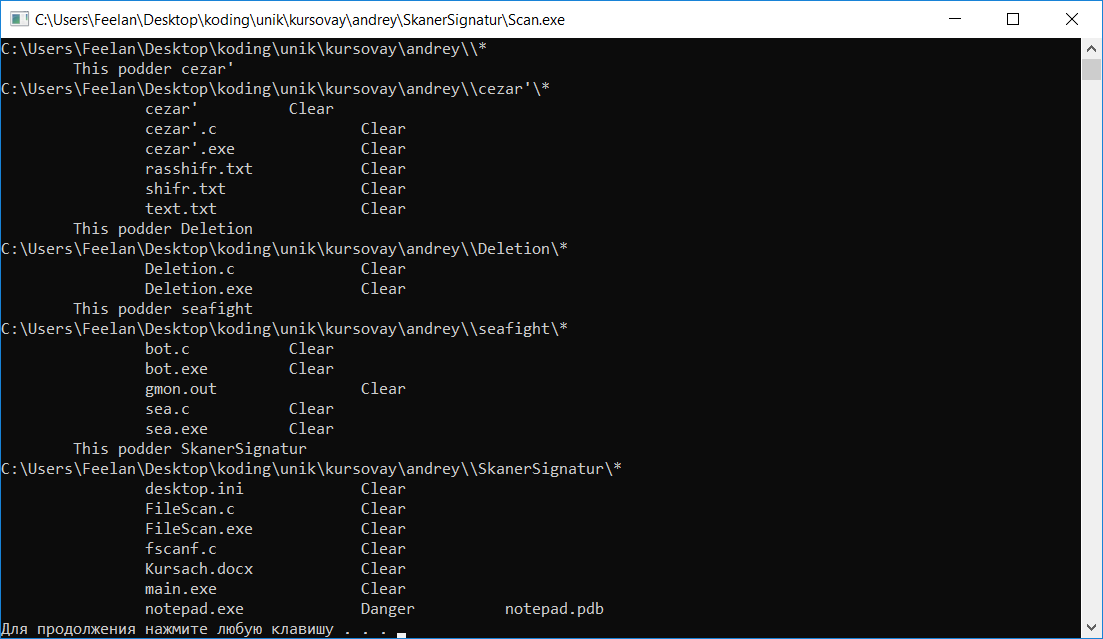


Рисунок 8 — Сообщение о нахождении «опасного»

Когда файлов больше не останется, для завершения программа попросит вас нажать любую клавишу для завершения (рисунок 9).

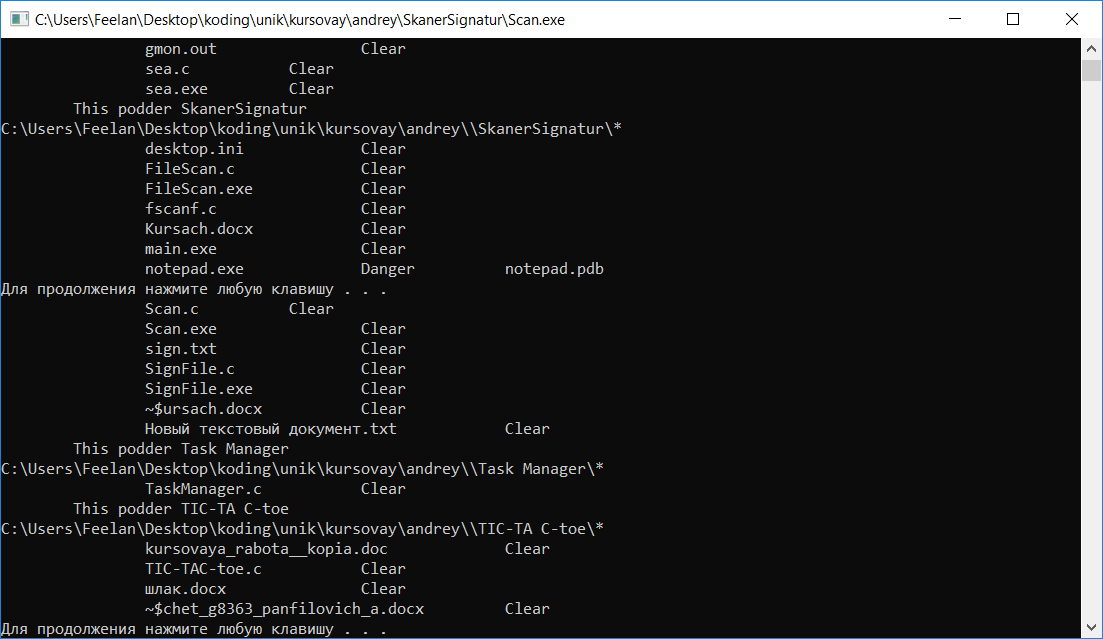


Рисунок 8 — Конец работы программы

**ПРИЛОЖЕНИЕ 3. ИСХОДНЫЙ КОД**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>

#include <windows.h>

#include <locale.h>

#define SIGNLEN 1000

int SignatureComparison(long int SignIndent, char \*FileAdrr, char \*SignName)

{

if(FileAdrr == NULL)

{

printf("\a\nArgument error!");

return 2;

}

if(SignName == NULL)

{

printf("\a\nArgument error!");

return 3;

}

int Error;

FILE\* NoDer;

char Mz[2];

char \*Sign;

Sign = (char\*)malloc((strlen(SignName) + 1) \* sizeof(char));

if(Sign == NULL)

{

printf("\a\nMemory allocation error!");

return 4;

}

NoDer = fopen(FileAdrr, "rb");

if(NoDer == NULL)

{

printf("\a\nError opening file!");

return 5;

}

if(fread(Mz, sizeof(char), 2, NoDer) != 2)

{

if(!feof(NoDer))

{

printf("\a\nError reading from file 1!");

return 6;

}

}

if(Mz[0] != 'M' || Mz[1] != 'Z')

{

Error = printf("\t\tClear\n");

if(Error == 0)

{

printf("\a\nError printf1!");

return 7;

}

if(fclose(NoDer) == EOF)

{

printf("\a\nError closing file!");

return 8;

}

return 0;

}

else

{

Error = fseek(NoDer, SignIndent, SEEK\_SET);

if(Error != 0)

{

printf("\a\nFile shift error!");

return 9;

}

if(fread(Sign, sizeof(char), strlen(SignName) + 1, NoDer) != strlen(SignName) + 1)

{

if(!feof(NoDer))

{

printf("\a\nError reading from file 2!");

return 10;

}

}

if(strcmp(Sign, SignName) == 0)

{

Error = printf("\a\t\tDanger\t\t%s\n", SignName);

if(Error < 0)

{

printf("\a\nError printf 1!");

return 11;

}

Error = system("pause");

if(Error != 0)

{

printf("\a\nError in the \"systems\" function!");

return 12;

}

if(fclose(NoDer) == EOF)

{

printf("\a\nError closing file!");

return 13;

}

return 0;

}

else

{

Error = printf("\t\tClear\n");

if(Error < 0)

{

printf("\a\nError printf 2!");

return 14;

}

if(fclose(NoDer) == EOF)

{

printf("\a\nError closing file!");

return 15;

}

return 0;

}

}

}

int ScanfDerr(char \*DerName)

{

if(DerName == NULL)

{

printf("\a\nArgument error!");

return 1;

}

int Error, i = 0;

Error = printf("DerName: ");

if(Error == 0)

{

printf("\a\nError printf 1!");

return 2;

}

while((DerName[i] = getchar()) != '\n')

{

if(DerName[i] == EOF)

{

printf("\a\nError in %d getchr", i);

return 3;

}

i++;

}

DerName[i] = '\0';

if(fflush(stdin) != 0)

{

printf("\a\nError clearing the stream stdin!");

return 4;

}

return 0;

}

int ScanSign(char \*SignName, long int \*SignIndent)

{

if(SignName == NULL)

{

printf("\a\nArgument error!");

return 1;

}

int Error, j, i = 0, Len = 0;

char Num[100];

FILE\* File;

File = fopen("sign.txt", "r");

if(File == NULL)

{

printf("\a\aCannot open file!");

return 3;

}

while((SignName[i] = fgetc(File)) != '\n')

{

if(SignName[i] == EOF)

{

printf("\a\aError %d fgetc1!", i);

return 4;

}

i++;

}

SignName[i] = '\0';

i = 0;

while((Num[i] = fgetc(File)) != EOF)

{

if(Num[i] == EOF)

{

printf("\a\aError %d fgetc 2!", i);

return 5;

}

i++;

}

Num[i] = '\0';

for(j = 0; i > 0; i--, j++)

{

if(Num[j] >= '0' && Num[j] <= '9') \*SignIndent += (Num[j] - '0') \* pow(10, i - 1);

}

Error = fclose(File);

if(Error == EOF)

{

printf("\a\aError fclose");

return 6;

}

return 0;

}

int FindFileInDer(char \*DerName, char \*SignName, long int SignIndent)

{

if(DerName == NULL)

{

printf("\a\nArgument error!");

return 1;

}

if(SignName == NULL)

{

printf("\a\nArgument error!");

return 2;

}

int Error;

HANDLE hFile = NULL;

FILE\* NoDer;

WIN32\_FIND\_DATAA DataInfo = {0};

char Next1[MAX\_PATH] = {0}, Next2[MAX\_PATH] = {0}, FileAdrr[MAX\_PATH] = {0};

Error = printf("%s\n", DerName);

if(Error < 0)

{

printf("\a\nError printf 1!");

return 4;

}

hFile = FindFirstFileA(DerName, &DataInfo);

do

{

if((0 == lstrcmp(DataInfo.cFileName, ".")) || (0 == lstrcmp(DataInfo.cFileName, "..")) || (0 == lstrcmp(DataInfo.cFileName, "$Recycle.Bin")) || (0 == lstrcmp(DataInfo.cFileName, "Documents and Settings")))

{

continue;

}

if(DataInfo.dwFileAttributes & FILE\_ATTRIBUTE\_DIRECTORY)

{

if(strcat(Next1, DerName) == NULL)

{

printf("\a\nError in str cat!");

return 5;

}

Next1[strlen(Next1) - 1] = '\0';

if(strcat(Next1, DataInfo.cFileName) == NULL)

{

printf("\a\nError in str cat!");

return 6;

}

Error = printf("\tThis podder %s\n", DataInfo.cFileName);

if(Error < 0)

{

printf("\a\nError printf 2!");

return 7;

}

if(strcat(Next1, "\\\*") == NULL)

{

printf("\a\nError in str cat!");

return 8;

}

Error = FindFileInDer(Next1, SignName, SignIndent);

if(Error != 0)

{

printf("\a\nError in FindFileInDer!");

return 9;

}

Next1[0] = '\0';

if(strcat(Next1, Next2) == NULL)

{

printf("\a\nError in str cat!");

return 10;

}

}

else

{

Error = printf("\t\t%s", DataInfo.cFileName);

if(Error < 0)

{

printf("\a\nError printf 3!");

return 11;

}

if(strcat(FileAdrr, DerName) == NULL)

{

printf("\a\nError in str cat!");

return 12;

}

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

if(strcat(FileAdrr, DataInfo.cFileName) == NULL)

{

printf("\a\nError in str cat!");

return 13;

}

Error = SignatureComparison(SignIndent, FileAdrr, SignName);

if(Error != 0)

{

printf("\a\nError in SignatureComparison!");

return 14;

}

FileAdrr[0] = '\0';

}

}while(FindNextFileA(hFile, &DataInfo));

return 0;

}

int main()

{

int Error;

long int SignIndent = 0;

char DerName[MAX\_PATH], SignName[SIGNLEN];

setlocale(LC\_ALL, "Rus");

Error = ScanfDerr(DerName);

if(Error != 0)

{

printf("Error ScanfDerr");

return 0;

}

Error = ScanSign(SignName, &SignIndent);

if(Error != 0)

{

printf("Error ScanSign");

return 2;

}

Error = system("cls");

if(Error != 0)

{

printf("\a\nError in the \"systems\" function!");

return 3;

}

Error = FindFileInDer(DerName, SignName, SignIndent);

if(Error != 0)

{

printf("\a\aError FindFileInDer!");

return 4;

}

system("pause");

return 0;

}