

Министерство науки и высшего образования Российской Федерации
ФГБОУ ВО «Пензенский государственный университет»
Кафедра «Вычислительная техника»

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовой работе по курсу «Программирование»
на тему «Файловый менеджер»

Выполнил:
студент группы 25ВВВ2

Гладков А. М.

Приняла:
к.т.н., доцент Генералова А. А.

Отлично
28.05.2026
Г. Генералова

Пенза 2026

ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет Вычислительной техники

Кафедра "Вычислительная техника"

"УТВЕРЖДАЮ"

Зав. кафедрой ВТ

М.А. Митрохин

« 11 » февраля 2026г

ЗАДАНИЕ

на курсовое проектирование по курсу

Программирование

Студенту Гладкову Александру Максимовичу Группа 25ВВВ2

Тема проекта: **Разработка программы сложной структуры методом нисходящего программирования.**

Исходные данные (технические требования) на проектирование

Программа "Файловый менеджер" выводит список файлов. Пользователь осуществляет поиск, сортировку, копирование и перемещение файлов.

Обязательные требования к программе:

1. Многомодульность.
2. Использование сложных типов данных – структур, списков, файлов.
3. Режим работы видеосистемы – текстовый / графический.
4. Устройство ввода информации – клавиатура, мышь.
5. Пользовательский интерфейс должен быть построен на основе меню и панели инструментов.
6. Наличие заставки.
7. Операционная система MS Windows.
8. Язык программирования – Си и Ассемблер.
9. Среда разработки ПО – Microsoft Visual Studio.

Объем работы по курсу

1. Расчетная часть

Разработка программы.

2. Графическая часть

Схема данных, схема ресурсов системы, схема работы системы,
иерархическая структура программы, схема взаимодействия программы.

3. Экспериментальная часть

Отладка программы.

Срок выполнения проекта по разделам

В соответствии с графиком.

Дата выдачи задания " 71 " февраля

Дата защиты проекта " 18 " июль 2020

Руководитель Г. Рендаль А.И.

Задание получил " 71 " февраля 2020г.

Студент А. Глазков А.М.

Оглавление

Введение.....	5
Декомпозиция задач и модульная структура программы.....	7
Выбор языка программирования, архитектура и структура программы	10
Разработка функции на языке Assembler.....	14
Руководство пользователя.....	18
Отладка и возникшие трудности	21
Заключение	23
Список использованных источников	25
Приложение А. Листинг программы.....	26

Введение

В настоящее время практически любой пользователь персонального компьютера ежедневно сталкивается с необходимостью работы с файлами и каталогами. Документы, изображения, исходные коды программ, архивы и другие данные должны храниться в удобной структуре, быстро находиться, копироваться и перемещаться между папками. Для выполнения этих действий в операционных системах используются файловые менеджеры.

Файловый менеджер представляет собой программу, предназначенную для просмотра содержимого каталогов и выполнения основных операций над файлами. К таким операциям относятся вывод списка файлов, поиск по имени, сортировка, копирование, перемещение и навигация по каталогам. Несмотря на наличие готовых программ такого типа, разработка собственного файлового менеджера является полезной учебной задачей, так как позволяет изучить работу с файловой системой на низком уровне.

Актуальность темы курсовой работы обусловлена тем, что работа с файлами является одной из базовых задач системного и прикладного программирования. При создании файлового менеджера необходимо использовать функции операционной системы, организовывать пользовательский интерфейс, обрабатывать ошибки ввода-вывода, управлять памятью и продумывать структуру программы. Всё это позволяет закрепить знания языка C и принципов модульного программирования.

Целью данной курсовой работы является разработка простой консольной программы на языке C [1, 2] для операционной системы Windows [3], выполняющей функции файлового менеджера. Программа должна быть совместима со средой Microsoft Visual Studio и использовать Windows API для работы с каталогами и файлами.

Для достижения поставленной цели были сформулированы следующие задачи: изучить особенности работы с файлами и каталогами в Windows;

разработать структуру программы; реализовать вывод содержимого каталога; реализовать поиск файлов по части имени; реализовать сортировку файлов; реализовать копирование и перемещение файлов; добавить ассемблерную вставку с использованием ключевого слова `__asm`; провести тестирование разработанной программы.

Объектом исследования является процесс разработки консольного программного обеспечения для работы с файловой системой. Предметом исследования является программа файлового менеджера, реализованная на языке C с применением Windows API и встроенного ассемблера Visual Studio.

Практическая значимость работы заключается в том, что разработанная программа может использоваться как учебный пример системного программирования на языке C. В ней показаны основные принципы взаимодействия с операционной системой Windows, организации модульной структуры проекта и обработки пользовательских команд через консольное меню.

При выполнении работы применялся метод нисходящего программирования. Сначала была определена общая задача программы, затем она была разделена на отдельные функциональные блоки: модуль меню, модуль ввода, модуль работы с путями, модуль работы с каталогами, модуль поиска, модуль сортировки, модуль операций с файлами и модуль с ассемблерной вставкой. Такой подход позволил сделать программу более понятной и удобной для дальнейшего сопровождения.

Структура пояснительной записки включает введение, раздел о декомпозиции задач, раздел о выборе языка программирования и архитектуре программы, описание ассемблерной функции, руководство пользователя, описание процесса отладки, заключение, список использованных источников и приложение с листингом программы.

Декомпозиция задач и модульная структура программы

Декомпозиция является одним из основных механизмов разработки программного обеспечения. Она позволяет разбить сложную задачу на набор более простых подзадач, каждая из которых может быть реализована отдельной функцией или модулем. В данной работе исходная задача заключалась в создании файлового менеджера, поэтому на первом этапе были выделены основные действия, которые должен выполнять пользователь.

К основным действиям относятся просмотр текущего каталога, поиск файлов, сортировка файлов, копирование, перемещение и изменение текущей директории. Каждое из этих действий было выделено в отдельный функциональный блок. Такой подход упрощает структуру программы: главный модуль не содержит подробной реализации операций, а только вызывает нужные функции в зависимости от выбора пользователя.

На верхнем уровне программы находится функция `main`. Она отвечает за получение текущего каталога, вывод информации о программе, отображение меню и обработку выбора пользователя. Функция `main` не выполняет непосредственную работу с файловой системой, а передаёт управление специализированным функциям. Это соответствует принципу единственной ответственности, согласно которому каждая часть программы должна решать ограниченный круг задач.

На среднем уровне расположены модули, отвечающие за конкретные области работы программы. Модуль `directory_ops.c` выполняет чтение каталога и вывод списка файлов. Модуль `search_ops.c` реализует рекурсивный поиск по имени. Модуль `sort_ops.c` отвечает за сортировку массива структур `FileInfo` по имени, размеру и времени изменения. Модуль `file_ops.c` содержит функции копирования и перемещения файлов.

На нижнем уровне находятся вспомогательные модули. Файл `input_utils.c` содержит функции для чтения строки и очистки буфера ввода. Файл `path_utils.c`

отвечает за формирование полного пути и маски поиска. Модуль `asm_utils.c` содержит функцию `string_length_asm`, в которой используется ассемблерная вставка. Файл `menu.c` отделяет вывод меню от основной логики программы.

Главный заголовочный файл `main.h` объединяет объявления всех функций, константы, подключаемые библиотеки и описание структуры `FileInfo`. Наличие единого заголовочного файла облегчает подключение модулей друг к другу и позволяет компилятору проверять корректность вызовов функций. На рисунке 1 показана общая структура проекта.

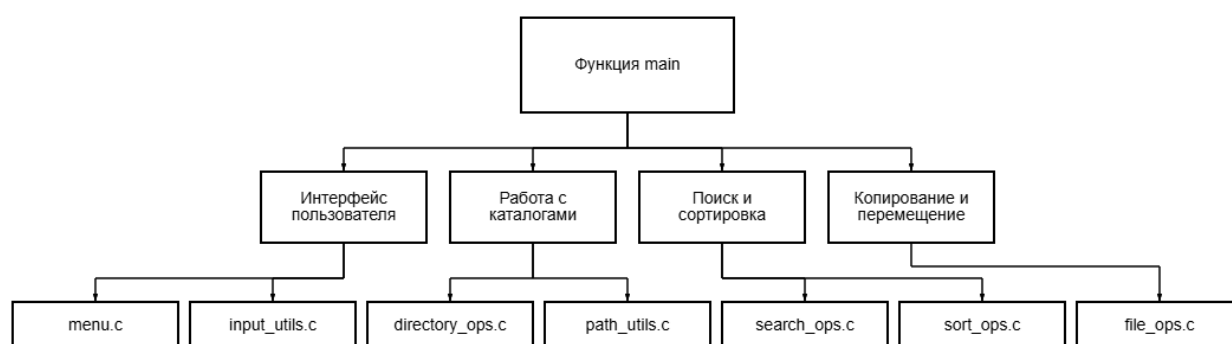


Рисунок 1 — Иерархическая декомпозиция программы файлового менеджера

Центральным элементом является главный модуль программы. От него расходятся основные функциональные блоки, каждый из которых отвечает за собственную часть логики. Такая структура делает программу более устойчивой к изменениям. Например, если необходимо изменить алгоритм сортировки, достаточно изменить файл `sort_ops.c`, не затрагивая остальные модули.

Модульная структура [4] также облегчает процесс отладки. Если ошибка возникает при поиске файлов, разработчик сразу понимает, что в первую очередь следует проверить функции из `search_ops.c` и функции формирования путей из `path_utils.c`. Если некорректно работает копирование, анализ начинается с `file_ops.c`. Это позволяет быстрее локализовать проблему.

В таблице 1 приведено назначение основных файлов проекта.

Таблица 1 — Модули программы и их назначение

Файл	Назначение	Основные функции
main.c	Главный управляющий модуль	main
main.h	Заголовочный файл проекта	объявления функций, структура FileInfo
input_utils.c	Работа с вводом	read_line, clear_input_buffer
path_utils.c	Формирование путей	make_path, make_search_mask
directory_ops.c	Работа с каталогами	load_directory, list_directory, change_directory
sort_ops.c	Сортировка	compare_by_name, compare_by_size, compare_by_time
search_ops.c	Поиск файлов	search_files, search_files_recursive
file_ops.c	Операции с файлами	copy_file_menu, move_file_menu
asm_utils.c	Ассемблерная вставка	string_length_asm
menu.c	Вывод меню	print_menu

Таким образом, декомпозиция задачи позволила создать понятную архитектуру программы, где каждая функция имеет определённое назначение. Благодаря этому исходный код становится более читаемым, его проще расширять и использовать в качестве основы для дальнейших учебных проектов.

Выбор языка программирования, архитектура и структура программы

Для реализации программы был выбран язык C. Этот язык хорошо подходит для задач системного программирования, так как позволяет эффективно работать с памятью, указателями, строками, файлами и функциями операционной системы. Кроме того, язык C широко используется в учебном процессе, так как помогает понять внутренние механизмы работы программ.

Средой разработки является Microsoft Visual Studio. Выбор данной среды обусловлен необходимостью разработки программы именно для Windows. Visual Studio предоставляет удобный редактор, компилятор Microsoft C/C++, средства отладки и возможность использовать встроенный ассемблер в проектах, собираемых под платформу x86.

Для работы с файловой системой используется Windows API [5]. Программа реализована через представленные в заголовочном файле Windows.h функции [6]. Обход каталогов реализован с помощью функций FindFirstFileA, FindNextFileA и FindClose. Эти функции позволяют получать информацию о файлах и папках в выбранном каталоге.

Копирование файлов выполняется с помощью функции CopyFileA, а перемещение — с помощью MoveFileExA. Использование системных функций Windows делает программу более надёжной, так как обработка низкоуровневых деталей файловой системы выполняется операционной системой.

Основной структурой данных программы является FileInfo. Она хранит имя объекта, размер файла, время последнего изменения и признак того, является ли объект каталогом. Массив таких структур используется при выводе и сортировке содержимого каталога.

Важной особенностью реализации является использование динамической памяти для хранения массива FileInfo. Изначально массив создавался как локальная переменная внутри функций list_directory и sort_and_show_files.

Однако при большом количестве файлов это могло привести к переполнению стека. Поэтому массив был перенесён в динамическую память с помощью `malloc`, а после завершения работы освобождается функцией `free`.

Схема взаимодействия пользователя, операционной системы и программы представлена на рисунке 2.



Рисунок 2 — Схема взаимодействия пользователя, программы и операционной системы

Пользователь взаимодействует с программой через консоль: вводит номер пункта меню, пути к файлам и каталогам, а также строку поиска. Программа передаёт соответствующие запросы операционной системе Windows, получает данные о файлах и выводит результат на экран.

Работа программы начинается с получения текущего каталога с помощью функции `_getcwd`. Затем выводится информация о программе, включая длину её названия, вычисленную ассемблерной функцией [7]. После этого запускается основной цикл меню. Цикл продолжается до тех пор, пока пользователь не выберет пункт выхода.

Алгоритм работы основной функции можно представить следующим образом: получить текущий каталог, вывести информацию о программе, показать меню, считать выбор пользователя, выполнить соответствующую

операцию, снова показать меню. Такой цикл, показанный на рисунке 3, делает программу интерактивной и позволяет выполнять несколько операций подряд без повторного запуска. [8]



Рисунок 3 — Блок-схема работы главного цикла программы

Функция `load_directory` является одной из ключевых функций программы. Она формирует маску поиска вида `путь\*`, вызывает `FindFirstFileA`, затем в цикле получает сведения обо всех найденных объектах. Для каждого объекта заполняется структура `FileInfo`. Каталоги `.` и `..` пропускаются, так как они обозначают текущий и родительский каталог.

Для сортировки используется стандартная функция `qsort` из библиотеки `C`. Ей передаётся массив `FileInfo`, количество элементов, размер одного элемента и функция сравнения. В зависимости от выбора пользователя используется одна из трёх функций сравнения: по имени, по размеру или по времени изменения.

Поиск файлов реализован рекурсивно. Функция `search_files_recursive` открывает текущий каталог, проверяет каждый объект и, если имя содержит введённую пользователем подстроку, выводит полный путь. Если найденный объект является каталогом, функция вызывает сама себя для этого каталога.

Благодаря этому поиск выполняется не только в текущей папке, но и во вложенных каталогах.

Копирование и перемещение реализованы через отдельные пункты меню. Пользователь вводит путь к исходному файлу и путь назначения. При копировании вызывается CopyFileA. При перемещении вызывается MoveFileExA с флагами MOVEFILE_COPY_ALLOWED и MOVEFILE_REPLACE_EXISTING, что позволяет перемещать файл даже между разными дисками и заменять существующий файл назначения.

В таблице 2 приведены основные функции Windows API, используемые в программе.

Таблица 2 — Используемые функции Windows API и библиотеки Visual C

Функция	Назначение	Где используется
FindFirstFileA	начало поиска файлов по маске	load_directory, search_files_recursive
FindNextFileA	получение следующего найденного объекта	load_directory, search_files_recursive
FindClose	закрытие дескриптора поиска	load_directory, search_files_recursive
CopyFileA	копирование файла	copy_file_menu
MoveFileExA	перемещение файла	move_file_menu
GetLastError	получение кода ошибки Windows	file_ops.c
_getcwd	получение текущего каталога	main.c
_chdir	смена текущего каталога	change_directory

Таким образом, архитектура программы построена на сочетании стандартных средств языка C и функций Windows API. Это обеспечивает совместимость с Visual Studio и позволяет продемонстрировать практические навыки работы с операционной системой Windows.

Разработка функции на языке Assembler

Одним из требований к курсовой работе являлось использование ассемблерной вставки. В Microsoft Visual Studio встроенный ассемблер доступен только при сборке проекта под платформу x86. Поэтому для корректной компиляции необходимо выбрать конфигурацию x86, а не x64.

В программе была реализована функция `string_length_asm`, предназначенная для подсчёта длины строки. Эта функция принимает указатель на строку и возвращает количество символов до завершающего нулевого символа. По смыслу она похожа на стандартную функцию `strlen`, однако реализована с помощью блока `__asm`

Выбор именно такой функции обусловлен тем, что работа со строками является простой и наглядной задачей для демонстрации низкоуровневых операций. В ассемблерном коде можно явно увидеть работу с указателем, чтение байта из памяти, проверку конца строки и увеличение счётчика.

Прототип функции выглядит следующим образом: `int string_length_asm(const char* str);`. Внутри функции используется локальная переменная `len`, в которую после завершения цикла записывается результат. Для обхода строки применяется регистр `ESI`, а для подсчёта символов — регистр `ECX`.

```
int string_length_asm(const char* str) {
    int len = 0;

    __asm {
        mov esi, str
        xor ecx, ecx

    count_loop:
        mov al, [esi]
        cmp al, 0
        je end_count
        inc esi
        inc ecx
        jmp count_loop
    }
```

```

    end_count:
        mov len, ecx
    }

    return len;
}

```

Команда `mov esi, str` записывает адрес начала строки в регистр ESI. Далее команда `xor ecx, ecx` обнуляет регистр ECX, который используется как счётчик символов. В цикле `count_loop` команда `mov al, [esi]` считывает текущий символ строки в младший байт регистра EAX.

После чтения символа выполняется сравнение `cmp al, 0`. Если текущий символ равен нулю, значит достигнут конец строки, и команда `je end_count` передаёт управление на завершение цикла. Если символ не равен нулю, указатель ESI увеличивается на единицу, счётчик ECX также увеличивается, и выполнение возвращается к началу цикла.

После завершения цикла значение счётчика копируется в переменную `len` командой `mov len, ecx`. Затем функция возвращает это значение обычным оператором `return` языка C. Таким образом, ассемблерный блок выполняет основную вычислительную часть функции, а код на C обеспечивает удобный интерфейс вызова.

Блок-схема работы ассемблерной функции представлена на рисунке 4.

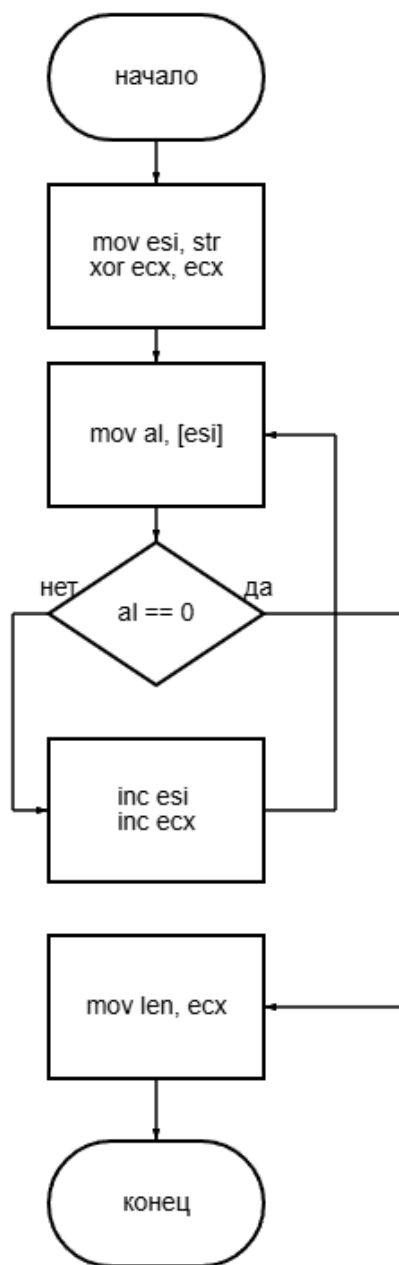


Рисунок 4 — Блок-схема функции string_length_asm

При разработке функции возникла особенность, связанная с именем локальной переменной. Изначально переменная называлась `length`, однако встроенный ассемблер Visual Studio воспринял это имя как служебное слово MASM, что привело к ошибке синтаксиса. Для устранения проблемы переменная была переименована в `len`.

Также необходимо учитывать, что встроенный ассемблер Visual Studio не поддерживается в 64-разрядных проектах. Поэтому при сборке программы

нужно выбрать платформу x86. Это ограничение указано в руководстве пользователя и учитывается при настройке проекта.

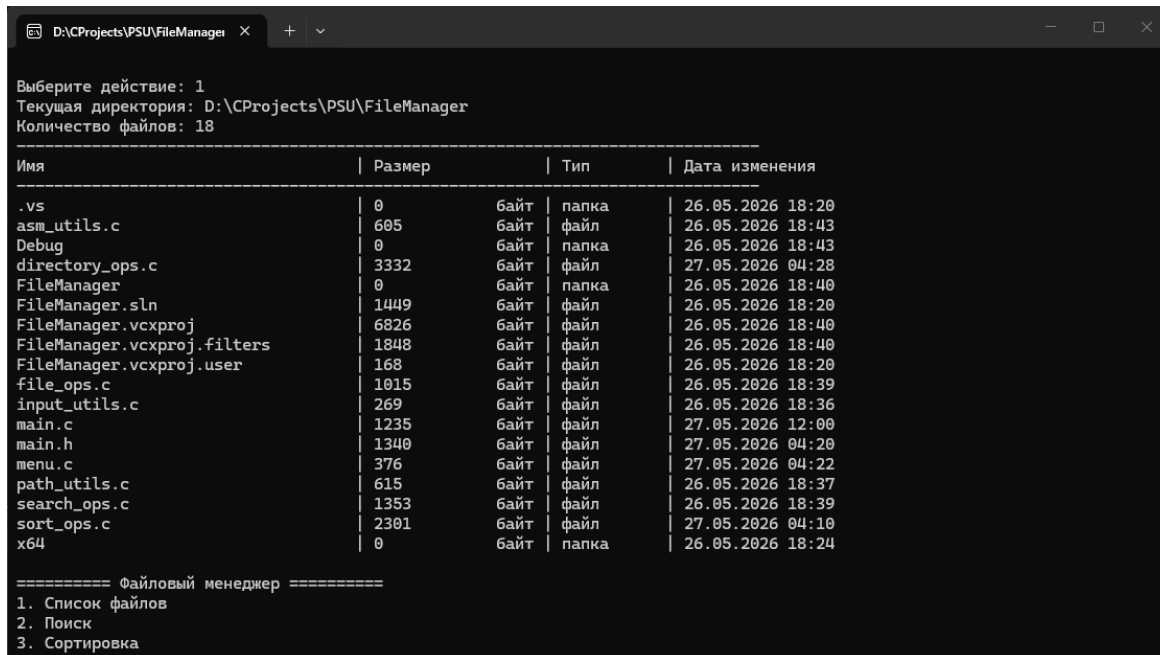
Использование ассемблерной вставки в данной программе имеет в первую очередь учебное значение. Она демонстрирует связь между языком С и машинными командами процессора, показывает работу с регистрами и памятью, а также помогает понять, как могут быть реализованы стандартные функции обработки строк.

Руководство пользователя

Разработанная программа представляет собой консольный файловый менеджер для операционной системы Windows. Для запуска программы необходимо открыть проект в Microsoft Visual Studio, выбрать платформу x86 и выполнить сборку. После запуска пользователь видит главное меню с доступными действиями.

Главное меню содержит следующие пункты: Список файлов, Поиск, Сортировка, Копирование, Перемещение, Изменение директории и Выход. Для выбора действия пользователь вводит соответствующую цифру и нажимает клавишу Enter.

Пункт «Список файлов» выводит список файлов и папок в текущем каталоге, как на рисунке 5. Для каждого объекта отображается имя, размер, тип, длина имени и дата последнего изменения. Если объект является каталогом, в столбце Тип выводится значение «папка», иначе выводится «файл».



Имя	Размер	Тип	Дата изменения
.vs	0	байт папка	26.05.2026 18:20
asm_utils.c	605	байт файл	26.05.2026 18:43
Debug	0	байт папка	26.05.2026 18:43
directory_ops.c	3332	байт файл	27.05.2026 04:28
FileManager	0	байт папка	26.05.2026 18:40
FileManager.sln	1449	байт файл	26.05.2026 18:20
FileManager.vcxproj	6826	байт файл	26.05.2026 18:40
FileManager.vcxproj.filters	1848	байт файл	26.05.2026 18:40
FileManager.vcxproj.user	168	байт файл	26.05.2026 18:20
file_ops.c	1015	байт файл	26.05.2026 18:39
input_utils.c	269	байт файл	26.05.2026 18:36
main.c	1235	байт файл	27.05.2026 12:00
main.h	1340	байт файл	27.05.2026 04:20
menu.c	376	байт файл	27.05.2026 04:22
path_utils.c	615	байт файл	26.05.2026 18:37
search_ops.c	1353	байт файл	26.05.2026 18:39
sort_ops.c	2301	байт файл	27.05.2026 04:10
x64	0	байт папка	26.05.2026 18:24

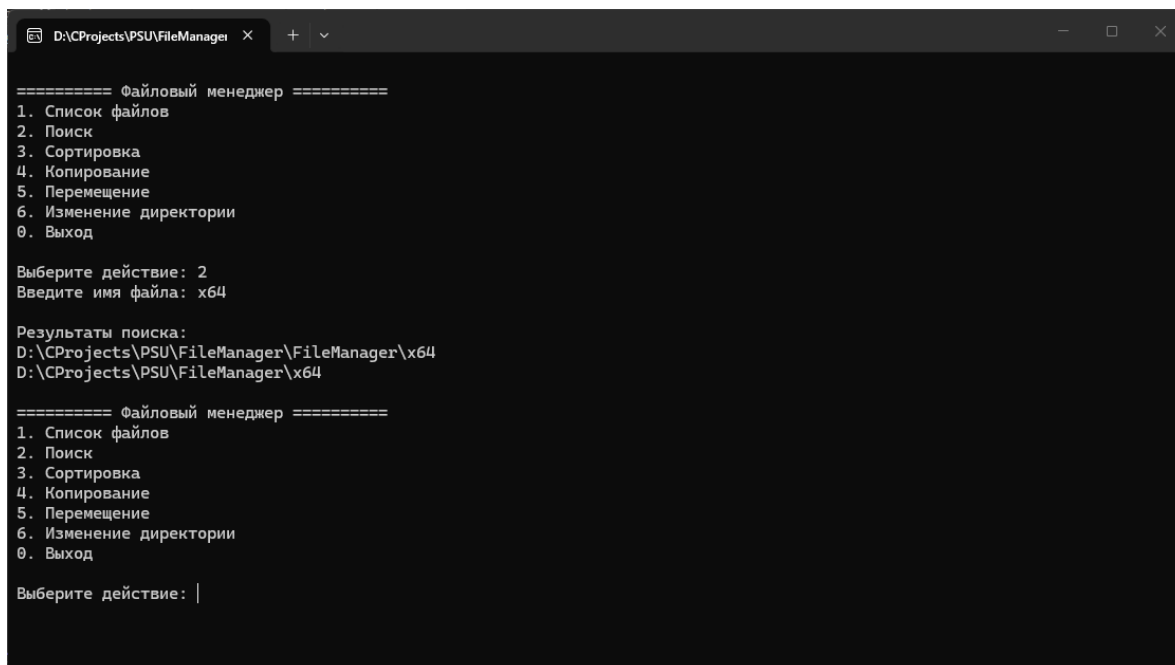
===== Файловый менеджер =====

1. Список файлов
2. Поиск
3. Сортировка

Рисунок 5 — Пример работы списка файлов

Пункт «Поиск» предназначен для поиска файлов и папок по части имени, как на рисунке 6. Пользователь вводит строку поиска, после чего программа

рекурсивно просматривает текущий каталог и все вложенные каталоги. Все найденные совпадения выводятся в виде полных путей.



```
===== Файловый менеджер =====
1. Список файлов
2. Поиск
3. Сортировка
4. Копирование
5. Перемещение
6. Изменение директории
0. Выход

Выберите действие: 2
Введите имя файла: x64

Результаты поиска:
D:\CProjects\PSU\FileManager\FileManager\x64
D:\CProjects\PSU\FileManager\x64

===== Файловый менеджер =====
1. Список файлов
2. Поиск
3. Сортировка
4. Копирование
5. Перемещение
6. Изменение директории
0. Выход

Выберите действие: |
```

Рисунок 6 — Пример работы поиска

Пункт «Сортировка» позволяет вывести содержимое текущего каталога в отсортированном виде. Пользователь может выбрать сортировку по имени, по размеру или по дате изменения. После выбора критерия программа выводит таблицу файлов в новом порядке.

Пункт «Копирование» выполняет копирование файла. Пользователь вводит путь к исходному файлу и путь назначения. Если операция проходит успешно, программа выводит сообщение «Файл успешно скопирован». Если возникает ошибка, выводится код ошибки Windows.

Пункт «Перемещение» выполняет перемещение файла. Вводятся исходный путь и путь назначения. При успешном выполнении файл перемещается в новое место. Если файл с таким именем уже существует, он может быть заменён, так как используется флаг `MOVEFILE_REPLACE_EXISTING`.

Пункт «Изменение директории» позволяет изменить текущий каталог. После ввода нового пути программа вызывает `_chdir`, затем обновляет значение `current_path` с помощью `_getcwd`. Все последующие операции выполняются уже относительно нового каталога.

Для выхода из программы используется пункт 0. После выбора этого пункта выводится сообщение «Программа завершена», и главный цикл завершает работу.

Отладка и возникшие трудности

В процессе разработки программы возникло несколько трудностей, связанных с особенностями среды Visual Studio и операционной системы Windows. Первая проблема была связана с заголовочным файлом `dirent.h`. Этот файл часто используется в Linux и MinGW для обхода каталогов, однако в стандартном проекте Visual Studio он отсутствует. Поэтому работа с каталогами была переписана на Windows API.

Для обхода каталога были использованы функции `FindFirstFileA`, `FindNextFileA` и `FindClose`. Это позволило сделать программу совместимой с Visual Studio и Windows без необходимости устанавливать дополнительные библиотеки.

Вторая трудность была связана с использованием встроенного ассемблера. В конфигурации x64 Visual Studio не поддерживает блоки `__asm`. При попытке сборки в x64 программа не компилируется. Решением стало переключение проекта на платформу x86 через Configuration Manager.

Третья проблема возникла в функции `string_length_asm`. Переменная `length` вызвала ошибку встроенного ассемблера, так как это имя может конфликтовать со служебными словами MASM. После переименования переменной в `len` ошибка была устранена.

Ещё одна ошибка проявилась при выборе пункта Show files: программа могла аварийно завершаться. Причина заключалась в создании большого массива `FileInfo` как локальной переменной. Так как структура содержит строку большого размера, массив из 1000 элементов занимал значительный объём стека. В Visual Studio это могло приводить к переполнению стека.

Для решения этой проблемы массив был перенесён в динамическую память. Вместо `FileInfo files[MAX_FILES]` используется `FileInfo* files = (FileInfo*)malloc(sizeof(FileInfo) * MAX_FILES)`. После завершения работы с массивом обязательно вызывается `free(files)`. Это позволило устранить

аварийное завершение программы и сделать работу с большим количеством файлов более стабильной.

Также в ходе тестирования проверялась корректность обработки пользовательского ввода. После чтения числа через `scanf` в буфере ввода остаётся символ перехода на новую строку. Если его не удалить, последующий вызов `fgets` может сразу считать пустую строку. Для устранения этой проблемы была добавлена функция `clear_input_buffer`.

Отладка выполнялась с помощью встроенного отладчика Visual Studio. Использовались точки останова, пошаговое выполнение и просмотр значений переменных. При проверке функций Windows API дополнительно выводились коды ошибок с помощью `GetLastError`, что помогало понять причину неудачного копирования или перемещения файла.

Заключение

В ходе выполнения курсовой работы была разработана консольная программа файлового менеджера на языке C для операционной системы Windows. Программа реализует основные операции с файлами и каталогами: вывод содержимого текущей директории, поиск файлов по имени, сортировку, копирование, перемещение и смену текущего каталога.

Поставленная цель была достигнута. Разработанное приложение успешно выполняет заявленные функции и может использоваться как учебный пример работы с файловой системой Windows. В процессе реализации были применены функции Windows API, стандартные средства языка C, динамическое выделение памяти и встроенная ассемблерная вставка Visual Studio.

Использование модульной структуры позволило разделить программу на логические части. Каждый файл проекта отвечает за отдельную область: ввод, меню, пути, каталоги, поиск, сортировку, операции с файлами и ассемблерную функцию. Такая организация делает код более читаемым и удобным для сопровождения.

Особое внимание было уделено совместимости с Microsoft Visual Studio. Так как заголовочный файл `dirent.h` отсутствует в стандартной среде Visual Studio, обход каталогов был реализован средствами Windows API. Ассемблерная вставка была выполнена с учётом ограничения Visual Studio: проект необходимо собирать под платформу x86.

В ходе отладки были устранены ошибки, связанные с синтаксисом встроенного ассемблера, переполнением стека и обработкой пользовательского ввода. Это позволило повысить стабильность работы программы.

Разработанный файловый менеджер можно расширять в дальнейшем. Возможными направлениями развития являются добавление удаления файлов, создание каталогов, переименование объектов, фильтрация по расширению, сортировка в обратном порядке, поддержка Unicode-путей и графический

интерфейс. Тем не менее уже реализованная версия удовлетворяет требованиям курсовой работы и демонстрирует основные принципы системного программирования на языке С.

Список использованных источников

1. Керниган Б. У., Ритчи Д. М. Язык программирования С. — М.: Вильямс, 2015.
2. Прата С. Язык программирования С. Лекции и упражнения. — М.: Вильямс, 2019.
3. Таненбаум Э., Бос Х. Современные операционные системы. — СПб.: Питер, 2022.
4. Харрис Д., Харрис С. Цифровая схемотехника и архитектура компьютера. — М.: ДМК Пресс, 2021.
5. Рихтер Д. Windows для профессионалов. Создание эффективных Win32-приложений. — СПб.: Питер, 2001.
6. Microsoft Learn. Windows API: файловые функции и работа с каталогами.
7. Microsoft Learn. Inline Assembler. Документация Microsoft Visual C++.
8. ГОСТ 19.701-90. Единая система программной документации. Схемы алгоритмов, программ, данных и систем.

Приложение А. Листинг программы

Ниже приведён фрагмент листинга основных файлов программы. Полный проект состоит из нескольких модулей, подключающих общий заголовочный файл main.h.

```
/* main.h */
#ifndef FILE_MANAGER_H
#define FILE_MANAGER_H

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>
#include <direct.h>

#define MAX_PATH_LEN 1024
#define MAX_FILES 1000

typedef struct {
    char name[MAX_PATH_LEN];
    unsigned long long size;
    FILETIME modified_time;
    int is_directory;
} FileInfo;

void clear_input_buffer(void);
void read_line(char* buffer, int size);
void make_path(char* result, const char* dir, const char* name);
void make_search_mask(char* result, const char* dir);
int load_directory(const char* path, FileInfo files[], int*
count);
void list_directory(const char* path);
void search_files(const char* path);
void sort_and_show_files(const char* path);
void copy_file_menu(void);
void move_file_menu(void);
void change_directory(char* current_path);
int string_length_asm(const char* str);
void print_program_info(void);
void print_menu(void);
#endif

/* main.c */
#include "main.h"

int main(void) {
```

```

char current_path[MAX_PATH_LEN];
int choice;

if (_getcwd(current_path, MAX_PATH_LEN) == NULL) {
    printf("Error: cannot get current directory.\n");
    return 1;
}

print_program_info();

do {
    print_menu();
    if (scanf("%d", &choice) != 1) {
        printf("Input error.\n");
        clear_input_buffer();
        continue;
    }
    clear_input_buffer();

    switch (choice) {
        case 1: list_directory(current_path); break;
        case 2: search_files(current_path); break;
        case 3: sort_and_show_files(current_path); break;
        case 4: copy_file_menu(); break;
        case 5: move_file_menu(); break;
        case 6: change_directory(current_path); break;
        case 0: printf("Program finished.\n"); break;
        default: printf("Wrong option. Try again.\n"); break;
    }
} while (choice != 0);

return 0;
}

/* asm_utils.c */
#include "main.h"

int string_length_asm(const char* str) {
    int len = 0;
    __asm {
        mov esi, str
        xor ecx, ecx
    count_loop:
        mov al, [esi]
        cmp al, 0
        je end_count
        inc esi
        inc ecx
        jmp count_loop
    }
}

```

```

        end_count:
            mov len, ecx
        }
        return len;
    }

void print_program_info(void) {
    const char* title = "SIMPLE FILE MANAGER";
    int length = string_length_asm(title);
    printf("Program name: %s\n", title);
    printf("Program name length calculated using __asm: %d\n",
length);
}

/* directory_ops.c */
int load_directory(const char* path, FileInfo files[], int*
count) {
    WIN32_FIND_DATA find_data;
    HANDLE hFind;
    char search_mask[MAX_PATH_LEN];

    *count = 0;
    make_search_mask(search_mask, path);
    hFind = FindFirstFileA(search_mask, &find_data);

    if (hFind == INVALID_HANDLE_VALUE) {
        printf("Error: cannot open directory: %s\n", path);
        return 0;
    }

    do {
        if (strcmp(find_data.cFileName, ".") == 0 ||
            strcmp(find_data.cFileName, "..") == 0) {
            continue;
        }
        if (*count >= MAX_FILES) break;

        strncpy(files[*count].name, find_data.cFileName,
MAX_PATH_LEN - 1);
        files[*count].name[MAX_PATH_LEN - 1] = '\0';
        files[*count].is_directory =
            (find_data.dwFileAttributes &
FILE_ATTRIBUTE_DIRECTORY) != 0;
        files[*count].size =
            ((unsigned long long)find_data.nFileSizeHigh << 32) |
            find_data.nFileSizeLow;
        files[*count].modified_time = find_data.ftLastWriteTime;
        (*count)++;
    } while (FindNextFileA(hFind, &find_data));
}

```

```

        FindClose(hFind);
        return 1;
    }

/* file_ops.c */
#include "main.h"

void copy_file_menu(void) {
    char source[MAX_PATH_LEN];
    char destination[MAX_PATH_LEN];

    printf("Enter source file path: ");
    read_line(source, MAX_PATH_LEN);

    printf("Enter destination file path: ");
    read_line(destination, MAX_PATH_LEN);

    if (CopyFileA(source, destination, FALSE)) {
        printf("File copied successfully.\n");
    }
    else {
        printf("File was not copied. Error code: %lu\n",
GetLastError());
    }
}

void move_file_menu(void) {
    char source[MAX_PATH_LEN];
    char destination[MAX_PATH_LEN];

    printf("Enter source file path: ");
    read_line(source, MAX_PATH_LEN);

    printf("Enter destination file path: ");
    read_line(destination, MAX_PATH_LEN);

    if (MoveFileExA(source, destination, MOVEFILE_COPY_ALLOWED |
MOVEFILE_REPLACE_EXISTING)) {
        printf("File moved successfully.\n");
    }
    else {
        printf("File was not moved. Error code: %lu\n",
GetLastError());
    }
}

```

```

/* input_utils.c */
#include "main.h"

void clear_input_buffer(void) {
    int c;
    while ((c = getchar()) != '\n' && c != EOF) {}
}

void read_line(char* buffer, int size) {
    if (fgets(buffer, size, stdin) != NULL) {
        buffer[strcspn(buffer, "\n")] = '\0';
    }
}

/* main.c */
#include "main.h"

int main(void) {
    setlocale(LC_ALL, "");
    char current_path[MAX_PATH_LEN];
    int choice;

    if (_getcwd(current_path, MAX_PATH_LEN) == NULL) {
        printf("Error: cannot get current directory.\n");
        return 1;
    }

    //print_program_info();

    do {
        print_menu();

        if (scanf("%d", &choice) != 1) {
            printf("Input error.\n");
            clear_input_buffer();
            continue;
        }

        clear_input_buffer();

        switch (choice) {
            case 1:
                list_directory(current_path);
                break;
            case 2:
                search_files(current_path);
                break;
            case 3:
                sort_and_show_files(current_path);

```

```

        break;
    case 4:
        copy_file_menu();
        break;
    case 5:
        move_file_menu();
        break;
    case 6:
        change_directory(current_path);
        break;
    case 0:
        printf("Program finished.\n");
        break;
    default:
        printf("Wrong option. Try again.\n");
        break;
    }
} while (choice != 0);

return 0;
}

/* menu.c */
#include "main.h"

void print_menu(void) {
    printf("\n===== Файловый менеджер =====\n");
    printf("1. Список файлов\n");
    printf("2. Поиск\n");
    printf("3. Сортировка\n");
    printf("4. Копирование\n");
    printf("5. Перемещение\n");
    printf("6. Изменение директории\n");
    printf("0. Выход\n");
    printf("\nВыберите действие: ");
}

/* path_utils.c */
#include "main.h"

void make_path(char* result, const char* dir, const char* name) {
    size_t len = strlen(dir);

    if (len > 0 && (dir[len - 1] == '\\\\' || dir[len - 1] == '/'))
    {
        snprintf(result, MAX_PATH_LEN, "%s%s", dir, name);
    }
    else {
        snprintf(result, MAX_PATH_LEN, "%s\\\\%s", dir, name);
    }
}

```

```

    }
}

void make_search_mask(char* result, const char* dir) {
    size_t len = strlen(dir);

    if (len > 0 && (dir[len - 1] == '\\\\' || dir[len - 1] == '/'))
    {
        snprintf(result, MAX_PATH_LEN, "%s*", dir);
    }
    else {
        snprintf(result, MAX_PATH_LEN, "%s\\\\*", dir);
    }
}

/* search_ops.c */
#include "main.h"

void search_files_recursive(const char* path, const char*
pattern) {
    WIN32_FIND_DATA find_data;
    HANDLE hFind;
    char search_mask[MAX_PATH_LEN];

    make_search_mask(search_mask, path);
    hFind = FindFirstFileA(search_mask, &find_data);

    if (hFind == INVALID_HANDLE_VALUE) {
        return;
    }

    do {
        char full_path[MAX_PATH_LEN];
        int is_directory;

        if (strcmp(find_data.cFileName, ".") == 0 ||
strcmp(find_data.cFileName, "..") == 0) {
            continue;
        }

        make_path(full_path, path, find_data.cFileName);

        if (strstr(find_data.cFileName, pattern) != NULL) {
            printf("%s\n", full_path);
        }

        is_directory = (find_data.dwFileAttributes &
FILE_ATTRIBUTE_DIRECTORY) != 0;

```

```

        if (is_directory) {
            search_files_recursive(full_path, pattern);
        }

    } while (FindNextFileA(hFind, &find_data));

    FindClose(hFind);
}

void search_files(const char* path) {
    char pattern[MAX_PATH_LEN];

    printf("Введите имя файла: ");
    read_line(pattern, MAX_PATH_LEN);

    if (strlen(pattern) == 0) {
        printf("Файлы не найдены.\n");
        return;
    }

    printf("\nРезультаты поиска:\n");
    search_files_recursive(path, pattern);
}

/* sort_ops.c */
#include "main.h"

int compare_by_name(const void* a, const void* b) {
    const FileInfo* file_a = (const FileInfo*)a;
    const FileInfo* file_b = (const FileInfo*)b;
    return _stricmp(file_a->name, file_b->name);
}

int compare_by_size(const void* a, const void* b) {
    const FileInfo* file_a = (const FileInfo*)a;
    const FileInfo* file_b = (const FileInfo*)b;

    if (file_a->size < file_b->size) return -1;
    if (file_a->size > file_b->size) return 1;
    return 0;
}

int compare_by_time(const void* a, const void* b) {
    const FileInfo* file_a = (const FileInfo*)a;
    const FileInfo* file_b = (const FileInfo*)b;

    return CompareFileTime(&file_a->modified_time, &file_b->modified_time);
}

```

```

void sort_and_show_files(const char* path) {
    FileInfo* files = (FileInfo*)malloc(sizeof(FileInfo) *
MAX_FILES);
    int count = 0;
    int choice;

    if (files == NULL) {
        printf("Memory allocation error.");
        return;
    }

    if (!load_directory(path, files, &count)) {
        free(files);
        return;
    }

    printf("Sort by :");
    printf("1. Name");
    printf("2. Size");
    printf("3. Modification date");
    printf("Choose option: ");

    if (scanf("%d", &choice) != 1) {
        printf("Input error.");
        clear_input_buffer();
        free(files);

        return;
    }

    clear_input_buffer();

    switch (choice) {
    case 1:
        qsort(files, count, sizeof(FileInfo), compare_by_name);
        break;
    case 2:
        qsort(files, count, sizeof(FileInfo), compare_by_size);
        break;
    case 3:
        qsort(files, count, sizeof(FileInfo), compare_by_time);
        break;
    default:
        printf("Wrong option. Sorting by name will be used.");
        qsort(files, count, sizeof(FileInfo),
compare_by_name);
        break;
    }
}

```

```

    printf("Sorted files :");
    printf("-----");
    printf("-----");
    printf("%-35s | %12s | %-10s | %s", "Name", "Size", "Type",
"Modified");
    printf("-----");
    printf("-----");

    for (int i = 0; i < count; i++) {
        print_file_info(&files[i]);
    }

    free(files);
}

```