МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра вычислительной техники

**Курсовая работа**

**по дисциплине «Программирование»**

**Тема: «Класс - структура данных в двоичном файле**»

Факультет: АВТФ Преподаватель: Менжулин С.А.

Группа: АВТ-342

Студенты: Долматов М.М.

Тема: 4. Класс - структура данных в двоичном файле (начальный балл – 4,5)

Вариант: 4.1

Новосибирск 2024

**Оглавление**

[**ОПИСАНИЕ ЗАДАНИЯ** 3](#_Toc186191666)

[**СТРУКТУРНОЕ ОПИСАНИЕ ЗАДАНИЯ** 4](#_Toc186191667)

[**Описание класса StringRecord** 5](#_Toc186191668)

[**Описание класса BinaryFile** 7](#_Toc186191669)

[**Формат файла:** 8](#_Toc186191670)

[**Реализация действий** 10](#_Toc186191671)

[**ФУНКЦИОНАЛЬНОЕ ОПИСАНИЕ ЗАДАНИЯ** 11](#_Toc186191672)

[**Описание класса StringRecord** 12](#_Toc186191673)

[**Описание класса BinaryFile** 18](#_Toc186191674)

[**Функциональное описание перегрузки операторов ввода/вывода для класса BinaryFile** 31](#_Toc186191675)

[**Функция log(const string& message)** 32](#_Toc186191676)

[**ОПИСАНИЕ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА** 33](#_Toc186191677)

[**ОПИСАНИЕ РАБОТЫ ПРОГРАММЫ НА КОНТРОЛЬНЫХ ПРИМЕРАХ** 42](#_Toc186191678)

[Как это работает: 42](#_Toc186191679)

[Оценка производительности: 43](#_Toc186191680)

[ВЫВОД 44](#_Toc186191681)

[СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ 46](#_Toc186191682)

[ПРИЛОЖЕНИЕ А 47](#_Toc186191683)

**ОПИСАНИЕ ЗАДАНИЯ**

4. Класс - структура данных в двоичном файле (начальный балл – 4,5)

Класс двоичного файла, производный от fstream. Двоичный файл содержит заданную структуру данных. Программа должна представлять собой простой текстовый редактор, использующий структуру данных для промежуточного хранения редактируемого файла. Должны быть реализованы операции создания и заполнения двоичного файла из заданного текстового и сохранения содержимого двоичного файла в текстовый.

Программа должна реализовывать указанные выше действия. Протестировать структуру данных. Программа тестирования должна содержать меню, обеспечивающее выбор операций.

1. Класс – двоичный файл, производный от fstream. Файл содержит массив указателей на строки, представленные записями переменной длины: целый счетчик и последовательность символов строки (без 0-ограничителя). Формат файла: в начале – файла, размерность массива указателей, текущее количество указателей, адрес (смещение) массива указателей в файле.

**СТРУКТУРНОЕ ОПИСАНИЕ ЗАДАНИЯ**

Проект представляет собой инструмент для работы с двоичными файлами, которые содержат строки переменной длины. Главная цель проекта — упростить процесс хранения, редактирования и обработки строк в специальном формате, позволяющем экономить место и работать с данными быстрее. В основе проекта лежат два ключевых класса: StringRecord и BinaryFile, каждый из которых реализует свои функции для управления данными.

**Общий функционал проекта:**

Проект обеспечивает:

1. Управление строками (StringRecord):
   * Каждая строка представлена объектом класса StringRecord, который содержит:
     + Длину строки (length).
     + Данные строки в формате уникального указателя (unique\_ptr<char[]>), что обеспечивает безопасное управление памятью.
   * Методы позволяют сериализовать (сохранять) и десериализовать (загружать) строки, а также преобразовывать их в формат std::string.
2. Работа с файлами (BinaryFile):
   * Сохраняет строки в двоичном файле.
   * Управляет массивом указателей (pointers), где хранятся смещения каждой строки относительно начала файла.
   * Обеспечивает функции для добавления, редактирования, удаления и сортировки строк.

Основные возможности проекта:

1. Хранение данных  
   Строки переменной длины сохраняются в бинарном формате, где каждый байт записывается точно так, как нужно для экономии пространства. Это отличается от текстового формата, где используются дополнительные символы вроде пробелов или переносов строк.
2. Обработка строк  
   Проект позволяет добавлять новые строки, редактировать существующие, удалять ненужные и даже менять их порядок. Все операции выполняются с учётом структуры файла, чтобы сохранить корректность данных.
3. Сортировка строк  
   Есть функции для сортировки строк в файле — по возрастанию или убыванию. Это удобно, если нужно организовать данные, например, в алфавитном порядке.
4. Работа с текстовыми файлами  
   Проект поддерживает преобразование данных из бинарного файла в обычный текстовый формат и обратно. Это позволяет, например, открыть данные в текстовом редакторе или загрузить их из текстового файла.
5. Эффективность  
   За счёт работы с указателями внутри файла и организации записей в памяти проект обеспечивает быстрый доступ к данным. Вместо того чтобы считывать весь файл, он быстро находит нужное место, используя массив смещений (указателей).

Процесс работы выглядит так:

1. Пользователь открывает файл через объект класса BinaryFile.
2. Добавляет строки, редактирует их, сортирует или удаляет, а затем сохраняет изменения.
3. При необходимости, данные можно экспортировать в текстовый файл или импортировать из него.

Что умеет делать проект:

– Добавление записей:  
Можно добавлять новые строки в конец файла или в любое конкретное место. Например, если вы добавили строку "Привет", она сохранится в бинарном формате.

– Редактирование строк:  
Любую строку в файле можно изменить. Например, если была строка "Hello", её можно заменить на "Hi".

– Удаление записей:  
Если строка больше не нужна, её можно удалить, а все остальные записи автоматически подстроятся.

– Сортировка:  
Строки можно упорядочить — например, от меньшей к большей длине или в алфавитном порядке.

– Загрузка и сохранение:  
Все изменения сохраняются в файл, чтобы их можно было использовать позже. Также можно загрузить данные из существующего файла и продолжить работу.

– Просмотр записей:  
Проект позволяет выводить содержимое файла на экран, чтобы увидеть, какие строки уже сохранены.

– Работа с текстовыми файлами:  
Удобно, если нужно быстро подготовить данные в текстовом редакторе, а потом загрузить их в бинарный файл.

**Описание класса StringRecord**

Класс StringRecord используется для работы с отдельными строками.

Основные поля:

* int length: длина строки в байтах.
* unique\_ptr<char[]> data: указатель на строку. Используется умный указатель для безопасного управления памятью (без утечек).

Основные методы:

1. Конструкторы:
   * StringRecord() — создаёт пустую строку.
   * StringRecord(const string& str) — создаёт объект на основе строки std::string.
2. Сохранение и загрузка:
   * void save(ostream& os) const: сохраняет строку в поток (например, файл).
   * void load(istream& is): загружает строку из потока.
3. Прочие методы:
   * string to\_string() const: возвращает строку в формате std::string.
   * Перегруженные операторы:
     + ostream& operator<<(ostream& os, const StringRecord& record) — для вывода строки в поток.
     + istream& operator>>(istream& is, StringRecord& record) — для ввода строки из потока.
4. Запрет копирования, разрешение перемещения:
   * Конструктор копирования и оператор присваивания запрещены.
   * Реализованы конструктор перемещения и оператор перемещения, чтобы избежать проблем с дублированием указателей.

Реализация StringRecord:

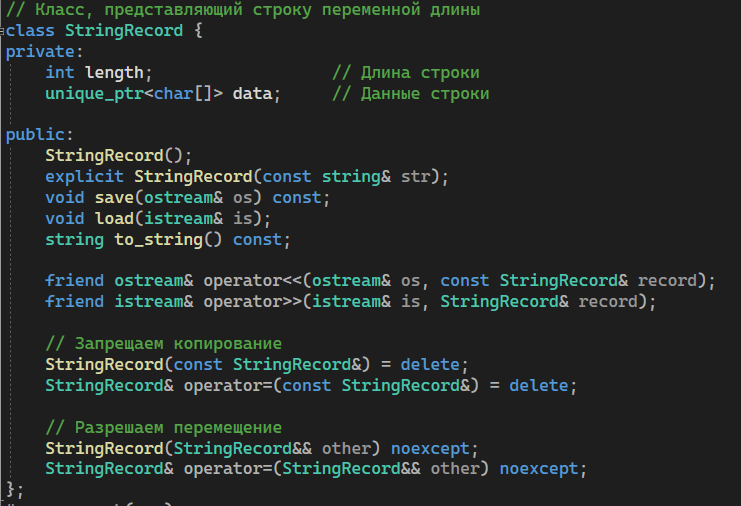


Рис. 1.2. Реализация StringRecord

**Описание класса BinaryFile**

Класс BinaryFile управляет файлами, содержащими массив строк.

Основные поля:

* int array\_size: размер массива указателей (смещений).
* int current\_count: текущее количество строк в файле.
* size\_t pointer\_array\_offset: смещение массива указателей относительно начала файла.
* vector<StringRecord> records: массив строк, которые находятся в оперативной памяти.
* vector<size\_t> pointers: массив смещений строк в файле.

Основные методы:

1. Конструктор и деструктор:
   * BinaryFile(const string& filename, ios::openmode mode):
     + Открывает файл в режиме чтения, записи или обоих.
     + Инициализирует массив указателей (pointers).
   * ~BinaryFile():
     + Закрывает файл и освобождает ресурсы.
2. Добавление, удаление, редактирование записей:
   * void add\_record(const string& str):
     + Добавляет строку в конец файла.
     + Обновляет массив указателей.
   * void insert\_record(size\_t index, const string& str):
     + Вставляет строку на указанную позицию.
     + Все последующие строки сдвигаются.
   * void delete\_record(size\_t index):
     + Удаляет строку по указанному индексу.
     + Массив указателей обновляется.
   * void edit\_record(size\_t index, const string& str):
     + Изменяет строку на указанную.
3. Сохранение и загрузка:
   * void save():
     + Сохраняет текущие строки и массив указателей в файл.
   * void load():
     + Загружает строки и массив указателей из файла.
4. Сортировка:
   * void insertion\_sort(bool ascending):
     + Сортирует строки методом вставки.
     + Параметр ascending определяет порядок сортировки.
   * void sort\_records(bool ascending):
     + Общая сортировка строк.
5. Работа с текстовыми файлами:
   * void load\_from\_text(const string& text\_filename):
     + Загружает строки из текстового файла в бинарный.
   * void save\_to\_text(const string& text\_filename):
     + Сохраняет строки из бинарного файла в текстовый.
6. Отображение строк:
   * void display\_records() const:
     + Выводит все строки из файла в консоль.

Реализация BinaryFile:

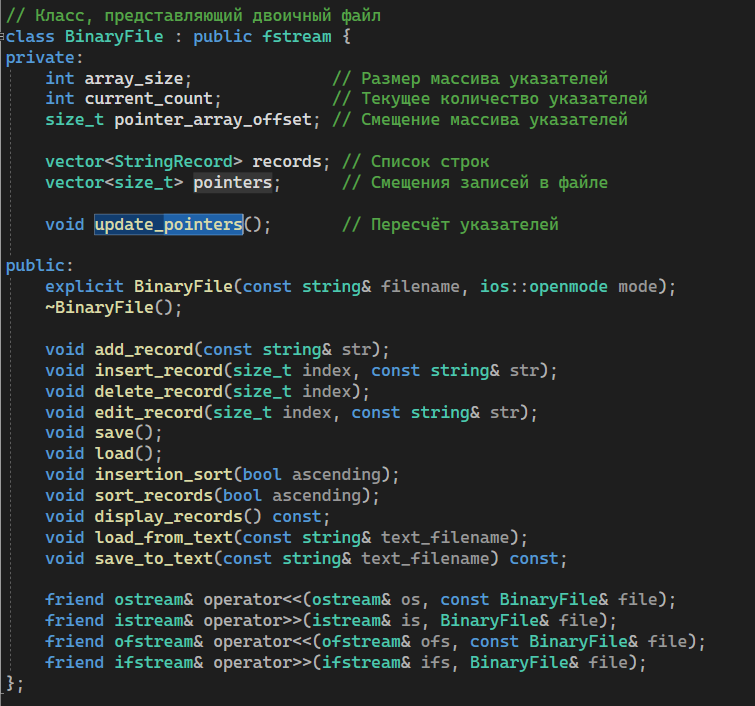


Рис. 1.2. Реализация BinaryFile

**Формат файла:**

Шапка файла:

* Размерность массива указателей (array\_size): Целое число, которое указывает, сколько указателей (смещений) будет храниться в массиве. Например, если в файле 10 строк, то array\_size = 10.
* Текущее количество указателей (current\_count): Целое число, которое указывает, сколько на данный момент записей существует в файле. Например, если в файле 3 строки, то current\_count = 3.
* Адрес (смещение) массива указателей в файле (pointer\_array\_offset): Целое число, которое указывает на смещение начала массива указателей в файле. Например, если массив указателей начинается с 16-го байта, то pointer\_array\_offset = 16.

Массив указателей (pointers):

* Содержит смещения строк в файле. Каждое значение в массиве указывает на позицию, с которой начинается соответствующая строка в файле.
* Массив указателей представляет собой вектор vector<size\_t> pointers, где каждый элемент — это смещение типа size\_t.

Строки переменной длины (StringRecord):

* Для каждой строки в файле сначала записывается её длина, затем сама строка.
* Длина строки (length): Целое число (например, 5), указывающее количество символов в строке.
* Данные строки (data): Массив символов, который хранит саму строку. Строка сохраняется в двоичном виде.

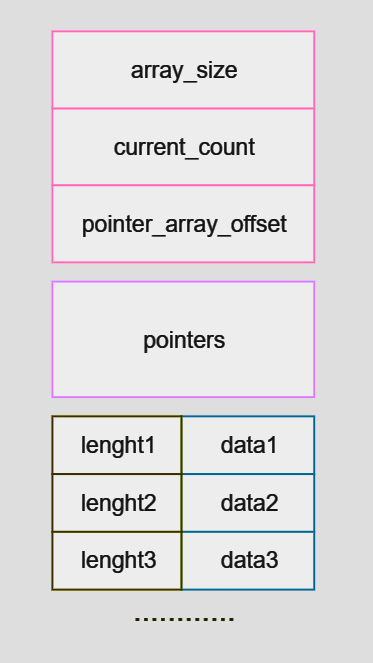


Рис. 1.3. Структура двоичного файла

**Реализация действий**

* Создание бинарного файла: Шапка файла записывается, массив указателей и строки добавляются по мере их ввода.
* Чтение бинарного файла: Cначала читается шапка файла, затем массив указателей, и по смещениям извлекаются строки.
* Сохранение содержимого в текстовый файл: Чтение всех строк из бинарного файла и запись их в текстовый файл в привычном формате.

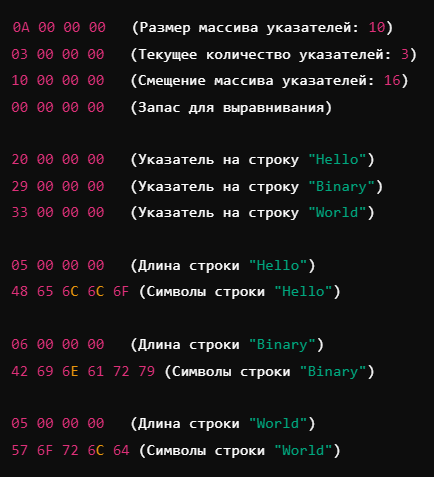


Рис. 1.4. Структура двоичного файла на примере

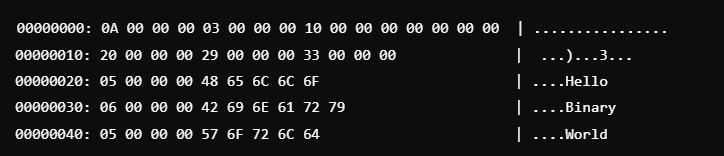


Рис. 1.5. Расшифрованный двоичный файл

Краткий пример использования, в дальнейшем мы рассмотрим все аспекты подробнее:

1. Вы создаёте объект BinaryFile и указываете имя файла. Если файл ещё не существует, он создаётся.
2. Добавляете строки, например: "Hello", "Binary", "World".
3. По желанию сортируете строки в алфавитном порядке.
4. Сохраняете данные в файл, и они остаются там до следующего открытия.
5. Позже вы можете открыть этот файл, удалить ненужную строку, добавить новую или отредактировать существующую.

**ФУНКЦИОНАЛЬНОЕ ОПИСАНИЕ ЗАДАНИЯ**

Программа начинается с подключения необходимых библиотек и определения директивы препроцессора #define \_CRT\_SECURE\_NO\_WARNINGS, которая отключает предупреждения о небезопасных функциях в Visual Studio. Библиотека BinaryFile.h включает в себя все необходимые классы для работы с бинарным файлом, а также используется библиотека conio.h для реализации функции \_getch() для ввода символов с клавиатуры, а windows.h — для очистки консоли с помощью system("cls").

В функции menu() создается умный указатель на объект BinaryFile, что позволяет автоматизировать управление памятью. Программа работает в цикле, показывая пользователю меню с различными вариантами действий, такими как создание нового файла, загрузка, управление записями, сортировка данных, а также сохранение и загрузка из текстовых файлов.

Основное меню содержит несколько пунктов, каждый из которых соответствует определенному действию, например:

* Создание нового двоичного файла,
* Загрузка и сохранение двоичных файлов,
* Управление записями в файле (добавление, удаление, редактирование),
* Сортировка данных по возрастанию или убыванию,
* Работа с текстовыми файлами (загрузка и сохранение).

Программа использует механизм try-catch для обработки ошибок, таких как некорректный ввод данных или проблемы при работе с файлами. Для каждого пункта меню реализованы соответствующие функции, и все действия логируются с помощью функции log(). В случае возникновения ошибки выводится сообщение об ошибке, и программа возвращается в меню.

Программа завершает свою работу, когда пользователь выбирает пункт выхода из меню, и в конце вызывается menu(), что запускает весь процесс.

**Описание класса StringRecord**

* 1. Конструктор по умолчанию StringRecord::StringRecord():
* Этот конструктор инициализирует объект StringRecord с нулевыми значениями для всех его переменных.
* Переменная length (длина строки) инициализируется значением 0, что означает, что строка еще не задана.
* Переменная data (указатель на данные строки) инициализируется значением nullptr, что означает отсутствие выделенной памяти для строки.
* Этот конструктор создается для случаев, когда необходимо создать пустой объект StringRecord без начальных данных.



Рис. 2.1. Конструктор по умолчанию StringRecord::StringRecord()

Описание:

* length(0) — устанавливает длину строки равной 0.
* data(nullptr) — указывает на то, что память для строки еще не выделена (пустая строка).
  1. Конструктор с параметром StringRecord::StringRecord(const string& str):
* Этот конструктор используется для инициализации объекта StringRecord значением строки типа string.
* В процессе работы конструктора:
  + Переменная length получает значение, равное размеру строки str. Размер строки извлекается с помощью функции str.size(), и для хранения этого значения используется тип int.
  + Переменной data выделяется динамическая память для хранения символов строки. Для этого используется make\_unique<char[]>, что создает массив символов необходимого размера.
  + Функция copy() копирует все символы из строки str в выделенную память, начиная с адреса, полученного через data.get().

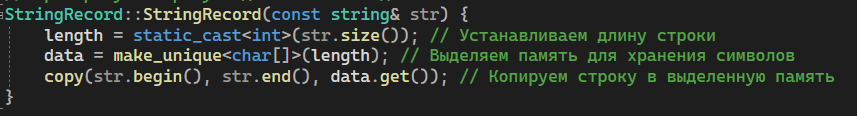


Рис. 2.2. Конструктор с параметром StringRecord::StringRecord(const string& str):

Описание:

* length = static\_cast<int>(str.size()) — вычисляет длину строки и сохраняет её в переменной length.
* data = make\_unique<char[]>(length) — выделяет динамическую память под массив символов.
* copy(str.begin(), str.end(), data.get()) — копирует все символы строки str в выделенную память.
  1. Перемещающий конструктор StringRecord::StringRecord(StringRecord&& other) noexcept:
* Этот конструктор используется для перемещения данных из другого объекта типа StringRecord. Это полезно, когда объект больше не будет использоваться, и его ресурсы можно передать другому объекту без лишних копий.
* В процессе работы конструктора:
  + Переменная length получает значение длины строки из другого объекта other.
  + Указатель data перемещается с помощью move(), что означает, что указатель теперь будет ссылаться на память объекта other.
  + После перемещения данных из other, длина у объекта other обнуляется, а указатель data становится nullptr.

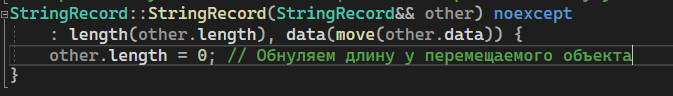


Рис. 2.3. Перемещающий конструктор StringRecord::StringRecord(StringRecord&& other) noexcept:

Описание:

* length(other.length) — переносит длину строки из исходного объекта other.
* data(move(other.data)) — перемещает указатель на данные из объекта other в текущий объект. Используется move() для оптимизации, чтобы не делать копию.
* other.length = 0 — обнуляет длину строки в объекте other, так как данные теперь принадлежат текущему объекту.
  1. Перемещающий оператор присваивания StringRecord& StringRecord::operator=(StringRecord&& other) noexcept:
* Оператор перемещающего присваивания используется для перемещения данных из одного объекта типа StringRecord в другой. Он выполняет те же операции, что и перемещающий конструктор, но применяется в случае, когда объект уже существует и ему присваиваются данные из другого объекта.
* В процессе работы оператора:
  + Сначала проверяется, не происходит ли самоприсваивание (когда объект присваивает сам себе). Если это так, операция пропускается.
  + Данные из объекта other перемещаются в текущий объект. Длина строки length копируется, а данные, хранимые в динамически выделенной памяти, перемещаются с помощью move().
  + У объекта other сбрасываются значения: его длина обнуляется, а указатель на данные становится nullptr.
  + В конце возвращается ссылка на текущий объект \*this, чтобы поддерживать цепочку присваиваний.

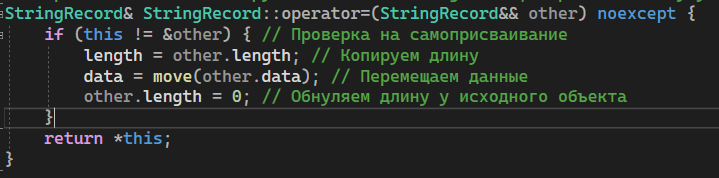


Рис. 2.4. Перемещающий оператор присваивания StringRecord& StringRecord::operator=(StringRecord&& other) noexcept:

Описание:

* if (this != &other) — проверка на самоприсваивание, чтобы избежать ненужных операций, если объект присваивает сам себе.
* length = other.length — копирует длину строки из объекта other.
* data = move(other.data) — перемещает указатель на данные из объекта other в текущий объект, используя move().
* other.length = 0 — сбрасывает длину строки у исходного объекта, так как данные теперь принадлежат текущему объекту.
  1. Метод save void StringRecord::save(ostream& os) const:
* Этот метод сохраняет данные объекта StringRecord в поток (например, в файл).
* Сначала записывается длина строки с помощью os.write(), используя reinterpret\_cast<const char\*>(&length) для преобразования целого числа в указатель на байтовые данные.
* Затем записываются сами данные строки, хранящиеся в массиве, с помощью os.write(data.get(), length).
* Метод используется для сериализации объекта в бинарный формат, позволяя сохранить строку и ее длину в поток.

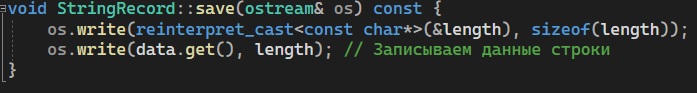


Рис. 2.5. Метод save void StringRecord::save(ostream& os) const

Описание:

* os.write(reinterpret\_cast<const char\*>(&length), sizeof(length)) — записывает длину строки в поток в бинарном виде.
* os.write(data.get(), length) — записывает символы строки в поток.
  1. Метод load void StringRecord::load(istream& is):
* Этот метод загружает данные из потока в объект StringRecord.
* Сначала считывается длина строки с помощью is.read(), после чего выделяется память для данных с помощью make\_unique<char[]>(length).
* Затем из потока считываются сами данные строки.
* Метод используется для десериализации объекта из бинарного формата.

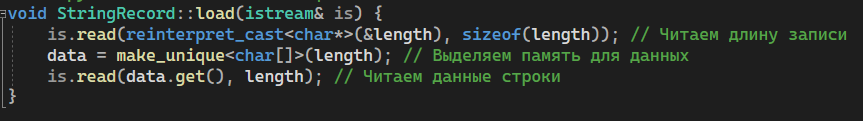


Рис. 2.6. Метод load void StringRecord::load(istream& is):

Описание:

* is.read(reinterpret\_cast<char\*>(&length), sizeof(length)) — считывает длину строки из потока.
* data = make\_unique<char[]>(length) — выделяет память для строки.
* is.read(data.get(), length) — считывает данные строки в выделенную память.
  1. Метод to\_string string StringRecord::to\_string() const:
* Этот метод преобразует объект StringRecord в обычную строку string, возвращая строковое представление данных.
* Он использует массив символов, хранящийся в объекте, и преобразует его в строку с помощью конструктора класса string, который принимает указатель на массив символов и его размер.

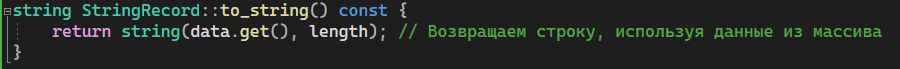


Рис. 2.7. Метод to\_string string StringRecord::to\_string() const

Описание:

* string(data.get(), length) — создает строку string из массива символов, на который указывает data, используя длину строки length как размер массива.
  1. Оператор вывода (ostream& operator<<(ostream& os, const StringRecord& record)):
* Предназначен для вывода содержимого объекта StringRecord в поток, например, в консоль или текстовый файл.
* Использует метод to\_string(), чтобы получить строковое представление записи. Это позволяет корректно отображать данные, хранящиеся в объекте, в виде обычной строки.
* Возвращает поток ostream, чтобы поддерживать цепочку операций вывода (например, cout << record1 << record2).

Описание:

* При вызове оператора << данные объекта StringRecord преобразуются в строку с помощью метода to\_string() и отправляются в указанный поток.
* Это делает объект удобным для работы в текстовом формате, поскольку позволяет отображать содержимое записи так же, как обычную строку.
  1. Оператор ввода (istream& operator>>(istream& is, StringRecord& record)):
* Предназначен для ввода данных в объект StringRecord из текстового потока, например, из консоли или текстового файла.
* Считывает строку из потока и создает новый объект StringRecord на основе считанных данных.
* Использует промежуточную строковую переменную для хранения данных, считанных из потока, после чего выполняет инициализацию объекта с помощью конструктора StringRecord.

Описание работы:

* При вызове оператора >> строка считывается из потока (до символа новой строки).
* На основе считанных данных создается новый объект StringRecord, который заменяет содержимое текущего объекта.
* Оператор возвращает поток istream, что позволяет выполнять цепочку операций ввода (например, cin >> record1 >> record2).

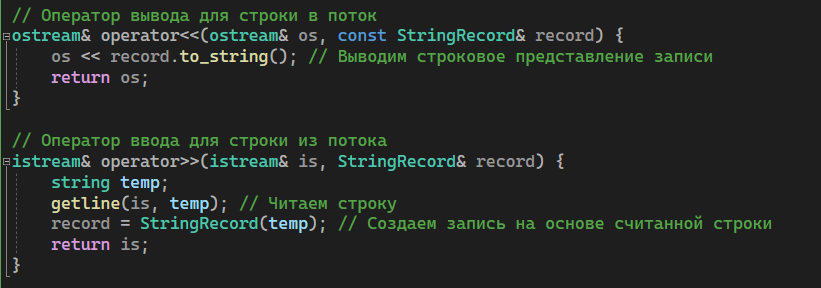


Рис. 2.8. Операторы ввода и вывода

Общее заключение функционального описания для класса StringRecord:

Класс StringRecord реализует эффективное управление строками переменной длины, с использованием динамически выделенной памяти для хранения данных. Он поддерживает создание, сохранение и загрузку строк в бинарном формате, а также их преобразование в строковое представление.

Основные функции класса включают:

* Конструкторы: Конструктор по умолчанию и конструктор с параметром для инициализации строки. Также реализованы конструктор перемещения и перемещающий оператор присваивания для эффективного управления памятью и ресурсами.
* Методы сериализации и десериализации: Методы save и load обеспечивают сохранение данных в поток и их восстановление из потока, что позволяет сериализовать и десериализовать объект в бинарном формате.
* Метод преобразования в строку: Метод to\_string предоставляет строковое представление данных, что позволяет удобно работать с объектами класса в текстовом формате.
* Операторы ввода и вывода: Операторы << и >> позволяют легко выводить и вводить строки из потока, предоставляя удобный интерфейс для работы с объектами класса StringRecord.

Реализация этих методов и операторов позволяет классу StringRecord эффективно управлять строками, минимизируя затраты памяти и времени, а также обеспечивая удобную работу с данными в различных форматах.

**Описание класса BinaryFile**

* 1. Конструктор BinaryFile::BinaryFile(const string& filename, ios::openmode mode)

**Назначение:**

* Конструктор класса BinaryFile предназначен для открытия бинарного файла с заданным именем и режимом доступа. Он инициализирует объект, открывает файл и, если файл открыт для чтения, загружает данные в объект.

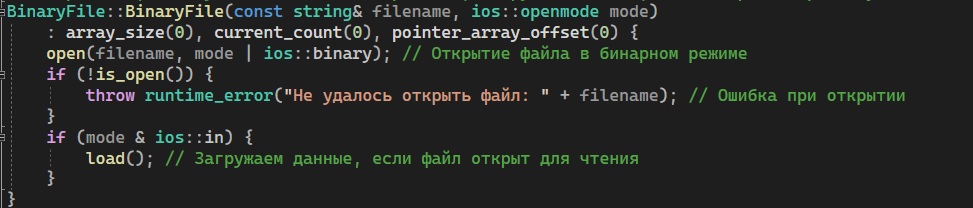


Рис. 3.1. Конструктор BinaryFile::BinaryFile(const string& filename, ios::openmode mode)

**Описание работы:**

1. **Инициализация переменных-членов:**
   * array\_size(0) — инициализация размера массива указателей на 0, т.к. изначально массив пуст.
   * current\_count(0) — инициализация текущего количества указателей на 0.
   * pointer\_array\_offset(0) — инициализация смещения для массива указателей, которое на момент создания объекта также равно 0.
2. **Открытие файла:**
   * Используется метод open(), который открывает файл в бинарном режиме с флагом ios::binary, добавляемым к переданному флагу доступа (например, для чтения или записи).
   * Если файл не удается открыть (проверка с помощью метода is\_open()), генерируется исключение типа runtime\_error, сообщающее о невозможности открытия файла с указанием его имени.
3. **Загрузка данных, если файл открыт для чтения:**
   * Если флаг ios::in (для чтения) присутствует в режиме, объект вызывает метод load(), чтобы загрузить данные из файла в структуру.
   1. Метод BinaryFile::add\_record(const string& str)

**Назначение:**

* Метод add\_record предназначен для добавления новой записи в объект класса BinaryFile. После добавления записи происходит обновление указателей и логирование операции.

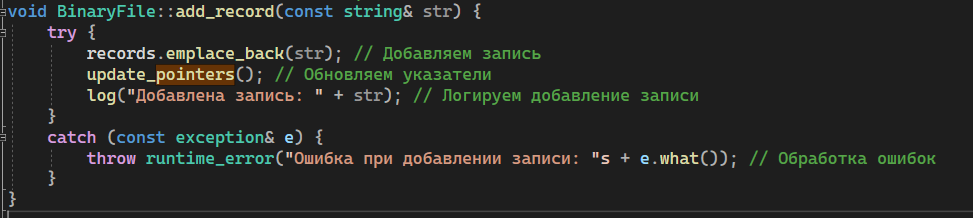


Рис. 3.2. Метод BinaryFile::add\_record(const string& str)

**Описание работы:**

1. **Добавление записи:**
   * Метод использует emplace\_back для добавления нового объекта StringRecord, который создается на основе переданной строки str.
   * records.emplace\_back(str) добавляет строку в вектор records, который хранит все строки в объекте BinaryFile.
2. **Обновление указателей:**
   * После добавления новой записи вызывается метод update\_pointers(), который пересчитывает и обновляет массив указателей на записи в файле. Это необходимо для поддержания корректных смещений и порядка данных.
3. **Логирование операции:**
   * В конце работы метода вызывается функция log(), которая записывает сообщение о добавлении новой записи в журнал (лог). Сообщение содержит строку, которая была добавлена.
4. **Обработка ошибок:**
   * Вся операция обернута в блок try-catch. В случае возникновения исключения (например, при добавлении записи или обновлении указателей), ловится стандартное исключение exception, и выбрасывается новое исключение типа runtime\_error, которое включает описание ошибки.
   1. Метод BinaryFile::insert\_record(size\_t index, const string& str)

**Назначение:**

* Метод insert\_record используется для вставки новой записи по заданному индексу в объект класса BinaryFile. После вставки обновляются указатели и выполняется логирование операции.

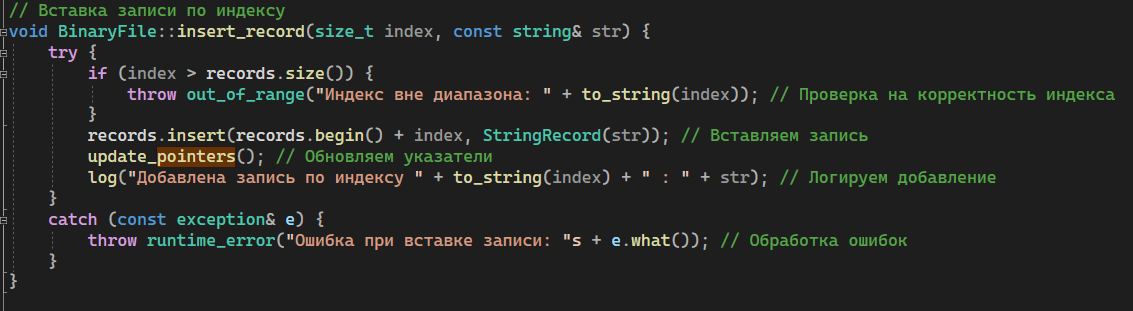


Рис. 3.3. Метод BinaryFile::insert\_record(size\_t index, const string& str)

**Описание работы:**

1. **Проверка корректности индекса:**
   * Метод начинает с проверки, что переданный индекс index находится в пределах допустимых значений. Если индекс больше размера вектора records, выбрасывается исключение out\_of\_range с сообщением об ошибке, которое включает недопустимый индекс.
2. **Вставка записи:**
   * Если индекс корректен, используется метод insert для вставки нового объекта StringRecord в вектор records. Запись создается на основе переданной строки str.
   * records.insert(records.begin() + index, StringRecord(str)) вставляет запись в нужную позицию в векторе.
3. **Обновление указателей:**
   * После успешной вставки записи вызывается метод update\_pointers(), который обновляет массив указателей, чтобы учесть изменения в списке записей.
4. **Логирование операции:**
   * После выполнения всех операций метод вызывает функцию log(), чтобы зафиксировать факт добавления записи в журнал. Сообщение включает индекс, по которому была добавлена запись, и саму строку.
5. **Обработка ошибок:**
   * Все действия обернуты в блок try-catch. В случае возникновения исключения (например, ошибки вставки или обновления указателей), перехватывается исключение exception, и выбрасывается новое исключение runtime\_error, которое включает описание произошедшей ошибки.
   1. Метод BinaryFile::delete\_record(size\_t index)

Назначение:

* Метод delete\_record используется для удаления записи по заданному индексу в объекте класса BinaryFile. После удаления записи обновляются указатели, и производится логирование операции.

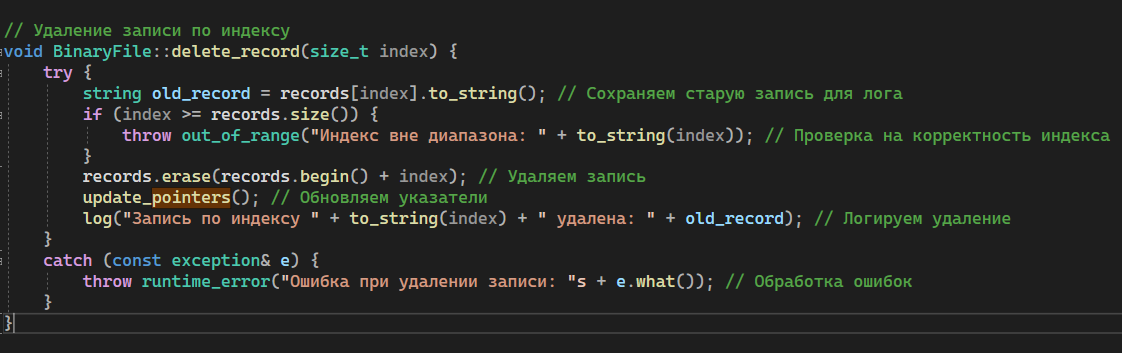


Рис. 3.4. Метод BinaryFile::delete\_record(size\_t index)

Описание работы:

1. Сохранение старой записи:
   * В начале метода происходит сохранение содержимого записи, которая будет удалена, в строковую переменную old\_record. Для этого используется метод to\_string() объекта записи по индексу index.
2. Проверка корректности индекса:
   * Далее, метод проверяет, находится ли переданный индекс index в допустимом диапазоне. Если индекс больше или равен размеру вектора records, выбрасывается исключение out\_of\_range с сообщением, указывающим на неверный индекс.
3. Удаление записи:
   * Если индекс корректен, то с помощью метода erase производится удаление записи из вектора records. Удаляется объект, находящийся по индексу index.
4. Обновление указателей:
   * После удаления записи вызывается метод update\_pointers(), который обновляет массив указателей, чтобы отразить изменения в списке записей.
5. Логирование операции:
   * После выполнения всех действий метод вызывает функцию log(), чтобы зафиксировать факт удаления записи. В журнал записывается индекс удаленной записи и ее содержимое, сохраненное в переменной old\_record.
6. Обработка ошибок:
   * Весь процесс обернут в блок try-catch. В случае возникновения исключений (например, ошибки удаления записи или обновления указателей), перехватывается исключение exception, и выбрасывается новое исключение runtime\_error, в котором описана произошедшая ошибка.
   1. Метод BinaryFile::edit\_record(size\_t index, const string& str)

**Назначение:**

* Метод edit\_record используется для редактирования существующей записи по заданному индексу в объекте класса BinaryFile. Он заменяет старую запись новой и обновляет указатели, а также логирует изменение.

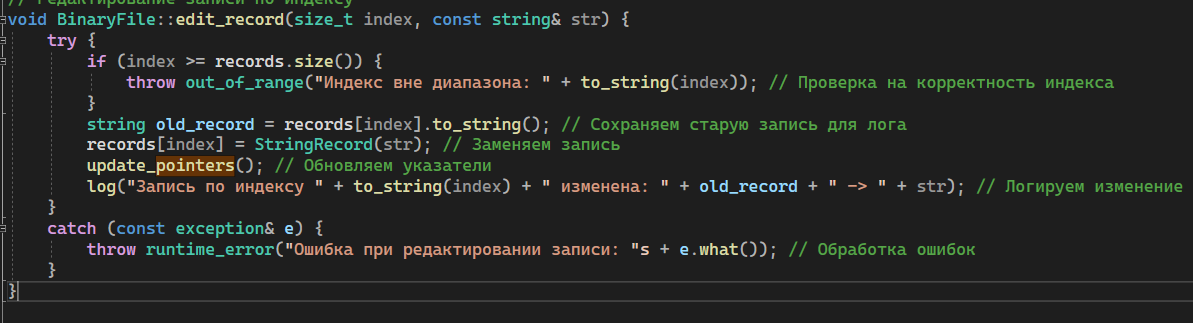


Рис. 3.5. Метод BinaryFile::edit\_record(size\_t index, const string& str)

**Описание работы:**

1. **Проверка корректности индекса:**
   * Сначала метод проверяет, что переданный индекс index не выходит за пределы вектора records. Если индекс больше или равен размеру вектора, выбрасывается исключение out\_of\_range, сообщающее о том, что индекс находится вне допустимого диапазона.
2. **Сохранение старой записи:**
   * Если индекс корректен, сохраняется старое содержимое записи в строковую переменную old\_record. Для этого используется метод to\_string() объекта записи по индексу index.
3. **Замена записи:**
   * Затем происходит замена старой записи на новую. Вектор records заменяет элемент с индексом index новым объектом класса StringRecord, который инициализируется строкой str.
4. **Обновление указателей:**
   * После замены записи вызывается метод update\_pointers(), который обновляет массив указателей, чтобы отражать изменения в списке записей.
5. **Логирование изменения:**
   * Далее вызывается метод log(), который записывает информацию об изменении в журнал. Логируемая информация включает индекс записи, старое значение записи и новое значение.
6. **Обработка ошибок:**
   * Весь процесс обернут в блок try-catch. В случае возникновения исключений (например, ошибок при изменении записи или обновлении указателей) перехватывается исключение exception, и выбрасывается новое исключение runtime\_error, в котором описана произошедшая ошибка.
   1. Метод BinaryFile::save()

Назначение:

* Метод save() сохраняет все данные в объекте класса BinaryFile в файл, открытый в бинарном режиме. Это включает в себя запись заголовка, массива указателей, а также всех строковых записей.

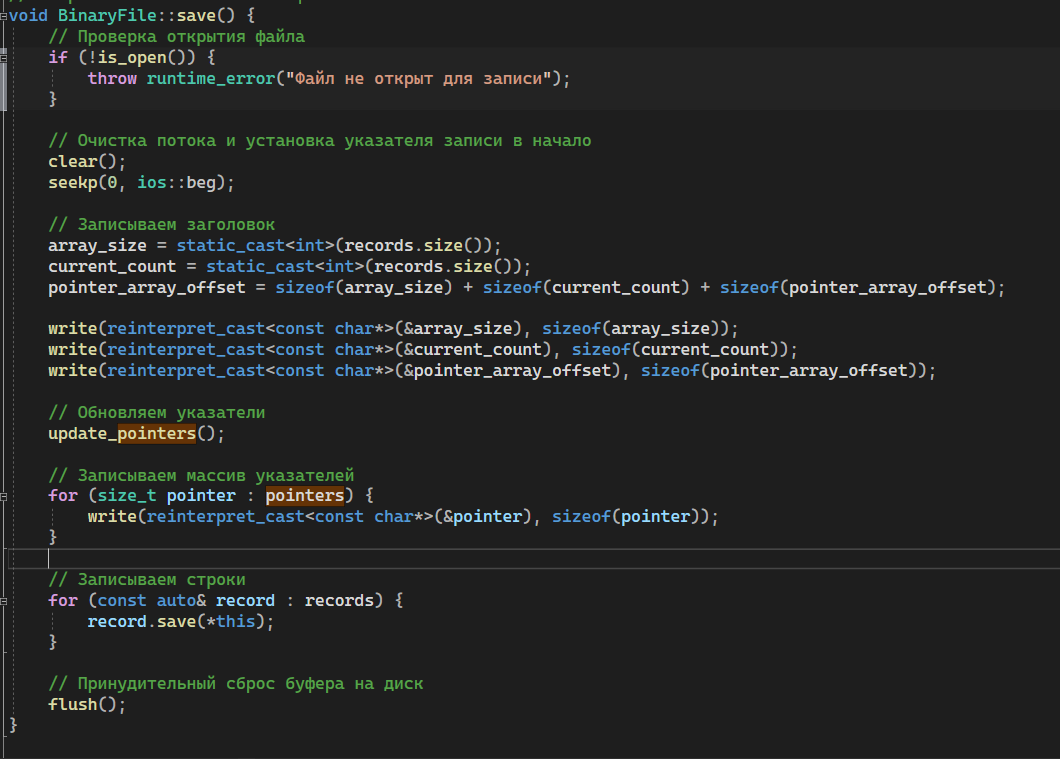


Рис. 3.6. Метод BinaryFile::save()

Описание работы:

1. Проверка открытия файла:
   * Сначала метод проверяет, открыт ли файл для записи с помощью is\_open(). Если файл не открыт, выбрасывается исключение runtime\_error с сообщением о том, что файл не открыт для записи.
2. Очистка потока и установка указателя записи в начало:
   * Далее метод очищает поток и устанавливает указатель записи в начало файла с помощью clear() и seekp(0, ios::beg). Это гарантирует, что запись в файл будет начинаться с самого начала.
3. Запись заголовка:
   * Записываются три целых числа: размер массива записей (array\_size), текущее количество записей (current\_count), и смещение массива указателей (pointer\_array\_offset). Эти значения устанавливаются перед записью данных в файл.
   * Эти три переменные записываются с помощью функции write, которая записывает данные в бинарном формате.
4. Обновление указателей:
   * Вызов метода update\_pointers() обновляет массив указателей, чтобы они соответствовали новым данным. Указатели, которые будут записаны в файл, представляют собой смещения, указывающие на начало каждой строки в файле.
5. Запись массива указателей:
   * Массив указателей записывается в файл с помощью цикла for, где каждый указатель записывается в бинарном формате с помощью write.
6. Запись строковых записей:
   * Для каждой записи в векторе records вызывается метод save для записи строковых данных в файл. Каждая запись сохраняется в файл в бинарном формате.
7. Принудительный сброс буфера на диск:
   * После завершения записи всех данных вызывается метод flush(), чтобы принудительно сбросить буфер записи на диск и завершить операцию записи в файл.

Метод save() сохраняет все данные из объекта BinaryFile в файл. Сначала записываются заголовочные данные, затем обновляются указатели и записываются сами записи. Этот метод обеспечивает сохранение всех структурированных данных в бинарном формате. После завершения записи на диск вызывается сброс буфера для сохранения данных. Также предусмотрена обработка ошибок, если файл не был открыт для записи.

* 1. Метод BinaryFile::load()

**Назначение:**

* Метод load() загружает данные из бинарного файла в объект класса BinaryFile. Он восстанавливает все записи и указатели, сохраненные ранее в файл, и инициализирует внутренние структуры объекта.



Рис. 3.7. Метод BinaryFile::load()

**Описание работы:**

1. **Очистка текущих данных:**
   * Метод начинает с вызова clear(), чтобы удалить все текущие данные, хранящиеся в объекте, и подготовить объект к загрузке новых данных.
2. **Установка указателя на начало файла:**
   * Устанавливается указатель потока чтения на начало файла с помощью seekg(0, std::ios::beg), чтобы начать чтение данных с самого начала файла.
3. **Проверка на открытие файла:**
   * Если файл не был открыт для чтения, выбрасывается исключение std::runtime\_error, с сообщением об ошибке.
4. **Чтение заголовка:**
   * Сначала из файла читаются заголовочные данные:
     + array\_size — размер массива (количество записей),
     + current\_count — текущее количество записей,
     + pointer\_array\_offset — смещение массива указателей.
   * Для каждого из этих значений используется функция read(), и если чтение не удалось, выбрасывается исключение с соответствующим сообщением об ошибке.
5. **Чтение указателей:**
   * Массив указателей pointers ресайзится в соответствии с количеством записей (array\_size).
   * Из файла считываются указатели с помощью read(), и если чтение не удалось, выбрасывается исключение.
6. **Чтение записей:**
   * Вектор records очищается, и затем начинаются попытки загрузить записи.
   * В цикле для каждой записи вызывается метод load() для объекта StringRecord, который считывает данные из файла и добавляет их в коллекцию записей.
   * В случае ошибки при загрузке записи выбрасывается исключение с описанием ошибки.

Метод load() загружает все данные, включая заголовок, указатели и записи, из бинарного файла. Он сначала очищает текущие данные объекта, затем читает заголовочные и указательные данные, а в конце — самі строки. В случае любых ошибок во время чтения данных выбрасываются исключения с описанием проблемы, что обеспечивает надежную обработку ошибок при загрузке данных из файла.

* 1. Метод BinaryFile::update\_pointers()

**Назначение:**

* Метод update\_pointers() обновляет массив указателей на записи в бинарном файле. Каждый указатель представляет собой смещение на соответствующую запись в файле, что позволяет эффективно управлять позициями данных при их сохранении или загрузке.

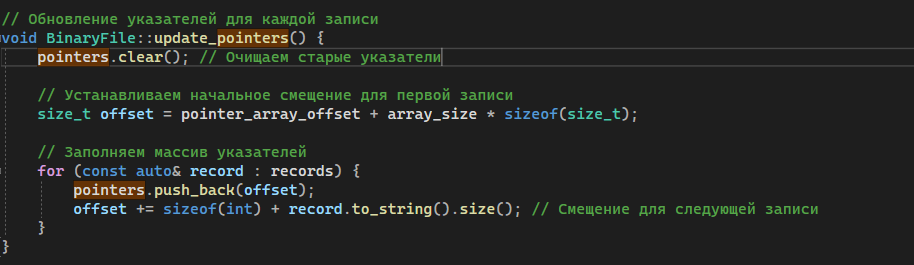


Рис. 3.8. Метод BinaryFile::update\_pointers()

**Описание работы:**

1. **Очистка старых указателей:**
   * В начале метода очищается вектор pointers с помощью clear(), чтобы удалить все предыдущие значения указателей, подготовив контейнер к новому заполнению.
2. **Установка начального смещения:**
   * Для первой записи устанавливается начальное смещение в переменной offset. Это смещение равняется pointer\_array\_offset + array\_size \* sizeof(size\_t).
   * pointer\_array\_offset — это смещение массива указателей в файле.
   * array\_size \* sizeof(size\_t) — количество байт, которое занимают все указатели в файле, учитывая их тип (size\_t).
3. **Заполнение массива указателей:**
   * Для каждой записи в векторе records вычисляется ее смещение и добавляется в массив pointers. Смещение для каждой записи увеличивается на величину, которая зависит от длины записи:
     + sizeof(int) — размер поля длины записи,
     + record.to\_string().size() — длина строки, хранящейся в записи (количество символов).
   * Таким образом, для каждой записи рассчитывается её смещение в файле относительно начала массива указателей.
   1. Метод BinaryFile::sort\_records(bool ascending)

**Назначение:**

* Метод sort\_records() выполняет сортировку записей в объекте BinaryFile в порядке, указанном пользователем — по возрастанию или убыванию. Сортировка выполняется с помощью алгоритма сортировки вставками (insertion sort), что позволяет эффективно работать с небольшими коллекциями данных.

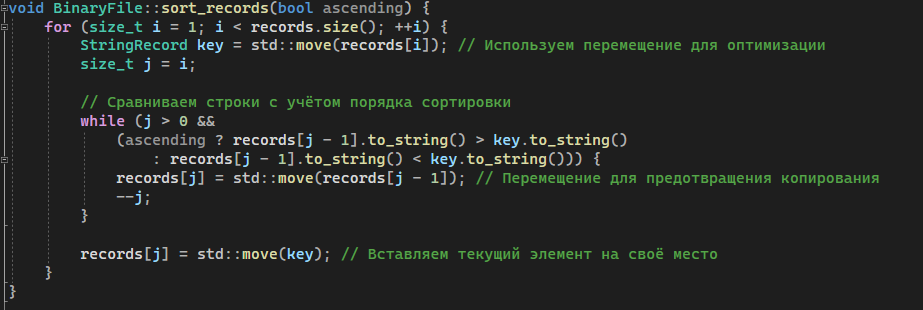


Рис. 3.9. Метод BinaryFile:: sort\_records()

**Описание работы:**

1. **Перебор элементов для сортировки:**
   * Метод итерирует по всем записям начиная с индекса 1 (i = 1), так как элемент на позиции 0 не требует сортировки.
2. **Ключ сортировки:**
   * Для каждого элемента создаётся временная переменная key, в которую переносится текущий элемент из коллекции records с использованием std::move(), что оптимизирует процесс перемещения данных вместо их копирования.
3. **Сравнение и сдвиг элементов:**
   * Внутри вложенного цикла while происходит сравнение строки текущей записи с предыдущей записью в зависимости от значения параметра ascending:
     + Если сортировка по возрастанию (ascending == true), то проверяется условие, при котором предыдущая строка должна быть больше текущей.
     + Если сортировка по убыванию (ascending == false), то проверяется условие, при котором предыдущая строка должна быть меньше текущей.
   * При удовлетворении условия записи сдвигаются на одну позицию влево, используя перемещение элементов, чтобы избежать излишнего копирования.
4. **Вставка текущего элемента:**
   * Когда элемент находит своё место в отсортированном подмножестве, он вставляется на свою позицию с использованием перемещения (std::move(key)).

Метод sort\_records() реализует сортировку записей в объекте BinaryFile по строковому значению записей. Для этого используется алгоритм сортировки вставками с оптимизацией через перемещение объектов, чтобы уменьшить затраты на копирование. Параметр ascending позволяет выбирать порядок сортировки (по возрастанию или убыванию). Этот метод полезен для упорядочивания данных в файле перед их сохранением или после загрузки.

* 1. Метод load\_from\_text(const string& text\_filename)

**Назначение:**

* Метод load\_from\_text() загружает данные из текстового файла в объект BinaryFile. Он читает записи из текстового файла и добавляет их в коллекцию records. Этот метод позволяет десериализовать данные, хранящиеся в текстовом формате, в бинарный файл.

**Описание работы:**

1. **Открытие файла:**
   * Метод пытается открыть указанный текстовый файл для чтения с помощью ifstream.
   * Если файл не удаётся открыть, выбрасывается исключение с сообщением о невозможности открыть файл.
2. **Чтение записей:**
   * После открытия файла, текущие записи очищаются методом clear().
   * Считывание записей из файла происходит в цикле с использованием оператора ввода operator>> для класса StringRecord. Каждая считанная запись добавляется в контейнер records.
3. **Проверка ошибок:**
   * После завершения чтения файла проверяется состояние потока. Если при чтении произошла ошибка, выбрасывается исключение с сообщением о проблемах при чтении.
4. **Обновление указателей:**
   * После загрузки всех записей, обновляются указатели с помощью метода update\_pointers().
5. **Логирование:**
   * После успешной загрузки данных, в лог записывается сообщение о том, что данные были успешно загружены.
6. **Обработка исключений:**
   * В случае возникновения исключений в процессе загрузки, ошибка логируется, и выбрасывается исключение с подробным сообщением.
   1. Метод save\_to\_text(const string& text\_filename) const

**Назначение:**

* Метод save\_to\_text() сохраняет данные из объекта BinaryFile в текстовый файл. Все записи из контейнера records записываются в файл, используя поток вывода и оператор вывода operator<< для класса StringRecord.

**Описание работы:**

1. **Открытие файла:**
   * Метод пытается открыть указанный текстовый файл для записи с помощью ofstream.
   * Если файл не удаётся открыть, выбрасывается исключение с сообщением о невозможности открыть файл для записи.
2. **Запись записей:**
   * Для каждой записи в контейнере records выполняется операция записи в текстовый файл с помощью оператора <<, который записывает строковое представление записи в файл.
   * Каждая запись записывается на новой строке.
3. **Проверка ошибок:**
   * После завершения записи в файл проверяется состояние потока. Если при записи возникла ошибка, выбрасывается исключение с сообщением о проблемах при записи в файл.
4. **Логирование:**
   * После успешной записи данных в файл, в лог записывается сообщение о том, что данные были успешно сохранены в текстовый файл.
5. **Обработка исключений:**
   * В случае возникновения исключений в процессе записи, ошибка логируется, и выбрасывается исключение с подробным сообщением.



Рис. 3.10. Методы load\_from\_text() и save\_to\_text()

Методы load\_from\_text() и save\_to\_text() обеспечивают эффективную загрузку и сохранение данных между бинарным файлом и текстовым форматом. Первый метод выполняет десериализацию данных из текстового файла, а второй — сериализацию в текстовый файл. Оба метода включают обработку ошибок и логирование, что обеспечивает надёжность работы с файлами и позволяет эффективно отслеживать ошибки.

### **Функциональное описание перегрузки операторов ввода/вывода для класса BinaryFile**

#### 1. Оператор вывода для бинарного файла (в текстовый поток)

ostream& operator<<(ostream& os, const BinaryFile& file);

* Этот оператор выводит содержимое объекта BinaryFile в текстовый поток. Он записывает количество записей в объекте и затем последовательно выводит каждую запись.

1. **Вывод количества записей:**
   * Сначала выводится количество записей в объекте BinaryFile (размер вектора records).
2. **Вывод каждой записи:**
   * Для каждой записи в контейнере records вызывается оператор вывода operator<< для класса StringRecord, который выводит строковое представление записи.
3. **Возврат потока:**
   * После вывода всех данных поток возвращается, чтобы продолжить работу с ним.

#### 2. Оператор ввода для бинарного файла (из текстового потока)

istream& operator>>(istream& is, BinaryFile& file);

* Этот оператор позволяет загружать данные из текстового потока в объект BinaryFile. Он считывает количество записей и затем загружает каждую запись в контейнер records.

1. **Очистка текущих записей:**
   * Вначале очищается контейнер records, чтобы удалить старые данные.
2. **Чтение количества записей:**
   * Читается количество записей, которое ожидается в файле.
3. **Чтение каждой записи:**
   * В цикле для каждого числа, равного количеству записей, происходит считывание объекта StringRecord. Если чтение не удаётся, выбрасывается исключение.
4. **Возврат потока:**
   * После завершения загрузки всех записей поток возвращается.

#### 3. Оператор вывода в бинарный файл

ofstream& operator<<(ofstream& ofs, const BinaryFile& file);

* Этот оператор записывает содержимое объекта BinaryFile в бинарный файл. Он сначала записывает количество записей, а затем сохраняет каждую запись в бинарном формате.

1. **Запись количества записей:**
   * Сначала записывается количество записей в объекте BinaryFile, которое сохраняется в бинарном формате.
2. **Запись каждой записи:**
   * Каждая запись сохраняется с помощью метода save(), который записывает данные в файл.
3. **Возврат потока:**
   * После завершения записи всех данных поток возвращается.

#### 4. Оператор ввода из бинарного файла

ifstream& operator>>(ifstream& ifs, BinaryFile& file);

* Этот оператор позволяет загружать данные из бинарного файла в объект BinaryFile. Он сначала читает количество записей, а затем загружает каждую запись из бинарного файла.

1. **Очистка текущих записей:**
   * Вначале очищается контейнер records, чтобы удалить старые данные.
2. **Чтение количества записей:**
   * Считывается количество записей, которое должно быть в бинарном файле.
3. **Проверка корректности данных:**
   * Если количество записей в файле отрицательно, выбрасывается исключение.
4. **Чтение каждой записи:**
   * В цикле для каждой записи выполняется загрузка данных с помощью метода load(), который читает бинарные данные.
5. **Возврат потока:**
   * После завершения загрузки всех записей поток возвращается.

Перегрузка операторов ввода и вывода для класса BinaryFile позволяет работать с объектами этого класса как с текстовыми, так и с бинарными потоками. Для текстового потока операторы сначала выводят количество записей, а затем каждую запись по очереди. Для бинарных файлов операторы записывают данные в эффективном бинарном формате, обеспечивая сохранение количества записей и самих данных, а затем загружают эти данные в объект, восстанавливая все записи. В случае ошибок чтения или записи выбрасываются соответствующие исключения с подробными сообщениями.

**Функция log(const string& message)**

Функция log(const string& message) предназначена для записи сообщений в лог-файл с отметкой времени. В начале функции используется библиотека chrono для получения текущего времени. Метод chrono::system\_clock::now() получает текущее время в системных единицах, после чего оно преобразуется в формат time\_t с помощью chrono::system\_clock::to\_time\_t(now). Далее, для безопасной работы с временем в Windows, используется функция localtime\_s, которая преобразует time\_t в структуру tm, представляющую локальное время.

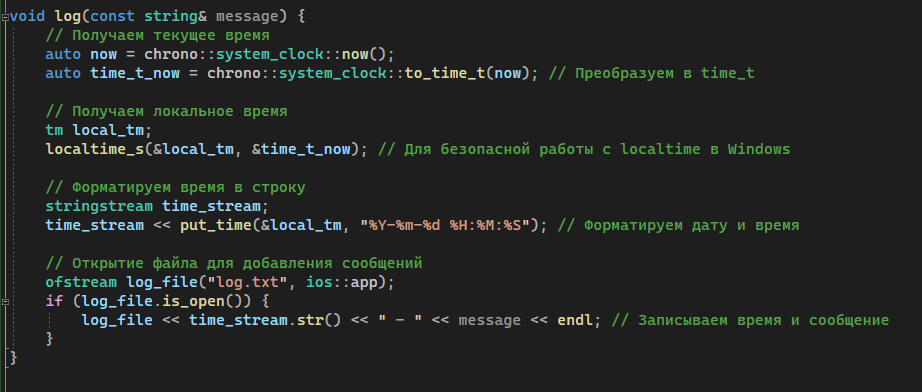


Рис. 4.1. Функция log(const string& message)

Затем с помощью stringstream и манипулятора put\_time формируется строка, содержащая отформатированное время в виде "Год-месяц-день Часы:Минуты:Секунды", что позволяет удобно отображать дату и время записи в логе.

После этого открывается файл log.txt в режиме добавления (ios::app), чтобы новые записи добавлялись в конец файла, а не перезаписывали существующие. Если файл успешно открыт, в него записывается строка, состоящая из текущего времени и самого сообщения. Запись завершается переводом строки.

Таким образом, функция log обеспечивает добавление сообщений с точной временной меткой в файл журнала для последующего анализа или отладки.

**ОПИСАНИЕ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА**

Для работы с файлом был создан пользовательский интерфейс, позволяющий удобно взаимодействовать с программой. Меню программы (Рис. 4.1) включает в себя следующие пункты:

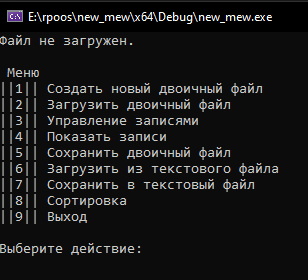


Рис. 5.1. Основной вид меню

Для вызова какого-либо пункта меню необходимо нажать соответствующую клавишу.

Рассмотрим функционал программы по пунктам:

Для начала нужно создать бинарный файл для дальнейшей работы, либо загрузить его, но так как проект новый, создадим файл 123:

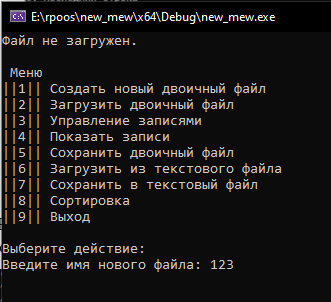


Рис. 5.2. Добавление элементов с помощью п.3

Далее рассмотрим п.3. Он самый важный так как именно через него и происходит основная выборочная работа с строками. Добавим несколько строк:

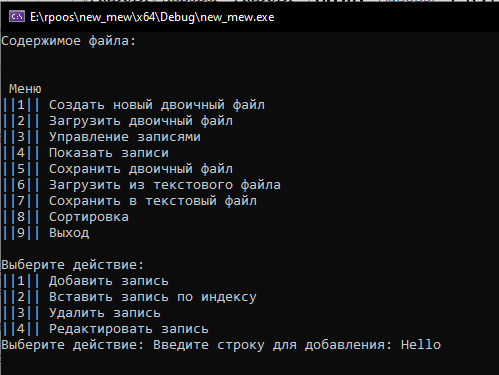


Рис. 5.3. Ввод строки

Все строки отображаются сверху в актуальном статусе и варианте файла, в котором мы находимся:

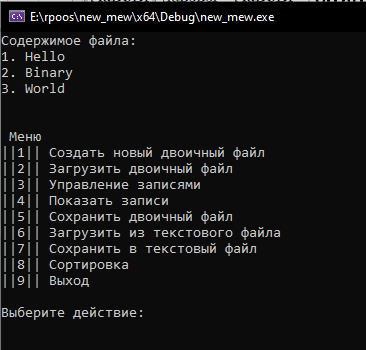


Рис. 5.4. Заполненный бинарный файл

Рассмотрим дальнейший функционал программы под п.3. Вставим, для примера, строку Nonono на место строки со вторым индексом:

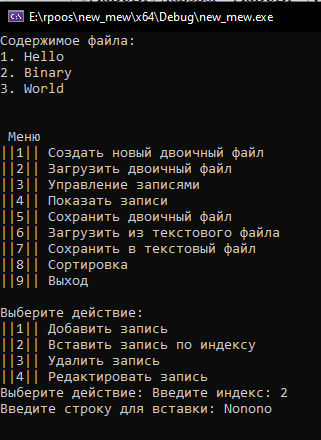


Рис. 5.5. Добавление элемента по индексу

Далее продолжим применять изменения через п.3. Удалим строку с первым индексом. Как видно на скриншотах, все прекрасно работает.

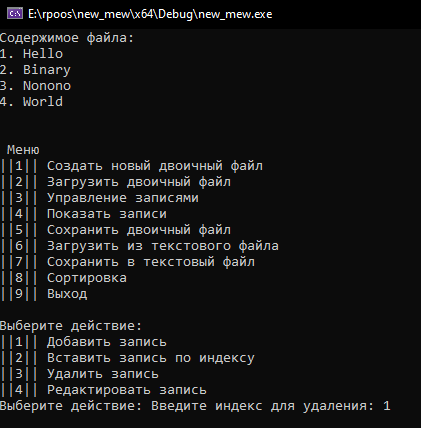


Рис. 5.6. Удаление элемента по индексу

Аналогично прошлым пунктам заменим, с помощью п.3, строку.

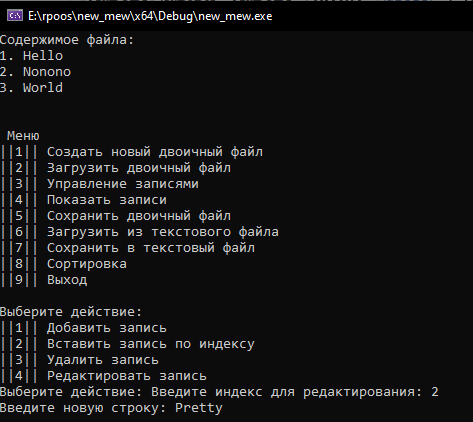


Рис. 5.7. Редактирование элемента по индексу

Имеет место быть отображение всего файла снизу, хоть он и отображается в актуальном формате, все равно в меню пункт обязан быть.

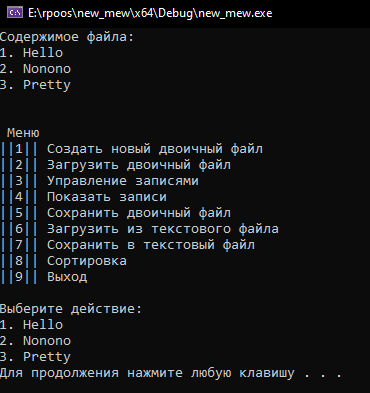


Рис. 5.8. Отображение всего файла по пункту

Пятый пункт заслуживает отдельного внимания, он выгружает из буфера временного хранения все строки что мы писали и редактировали прямиком в бинарный файл, простыми словами сохраняет его.

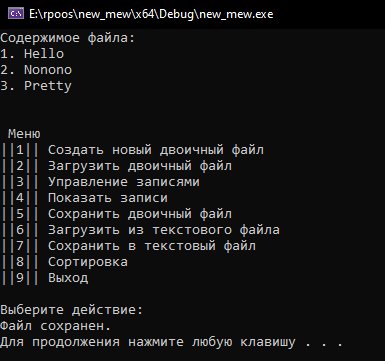


Рис. 5.9. Сохранение файла в бинарный

Перейдем к пункту 7. Все что у нас виднеется в бинарном файле мы можем выгрузить в текстовый файл, в котором все прекрасно будет отображаться. Ниже приведено два скриншота, где видно, что содержимое файла полностью перешло в текстовый файл. Кстати, его мы создали с этой же программы. Если не существует файла куда мы обращаемся, то он автоматически создается.

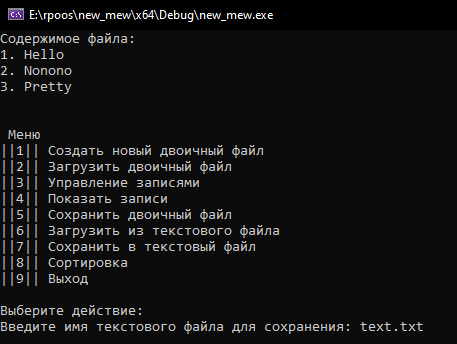


Рис. 5.9. Выгружаем все в текстовый файл

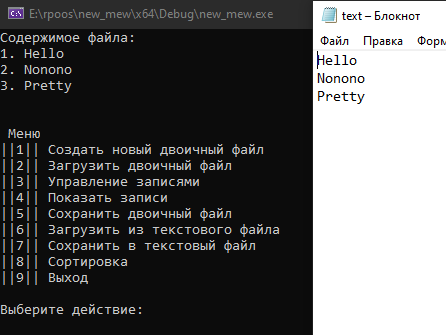


Рис. 5.10. Итог действий

Так же в моем проекте реализована лексикографическая сортировка. Она так же разбита на два под кейса и прекрасно сортирует то, что находится в нашем бинарном файле, а точнее в буфере.

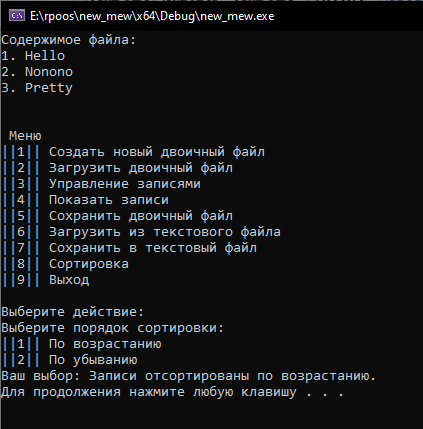


Рис. 5.11. Реализация сортировки

Так же, заранее изменив содержание текстового файла, для примера, можно его выгрузить в нашу программу, обработать и дальше выполнять действия.

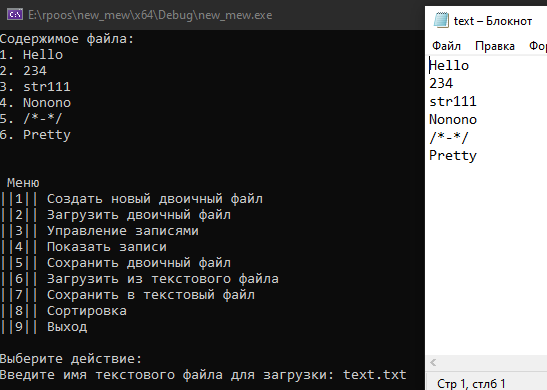


Рис. 5.12. Итоговый вид после выгрузки с текстового файла

В моем проекте так же есть log.txt, который отображает все действия, сохраняя при этом время и дату. Грубо говоря, история проекта записывается.

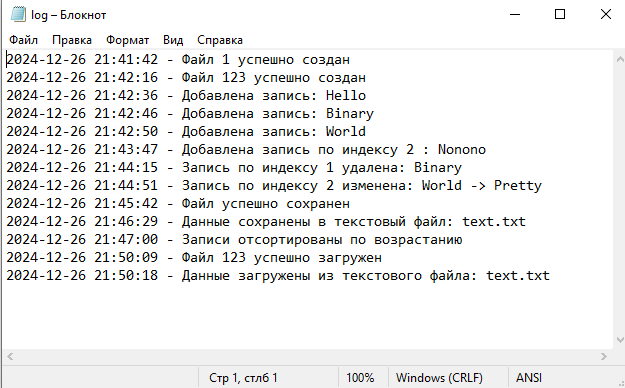


Рис. 5.12. Запись истории действий

**ОПИСАНИЕ РАБОТЫ ПРОГРАММЫ НА КОНТРОЛЬНЫХ ПРИМЕРАХ**

Для проверки скорости выполнения операций создадим несколько функций, которые будут отвечать за заполнение бинарного файла, а также направление в сортировку. Вызвать обе функции можно с помощью switch case в main.cpp.

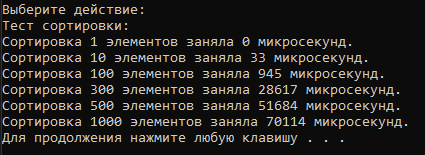


Рис. 6.1. Тестирование сортировки на i++ элементах

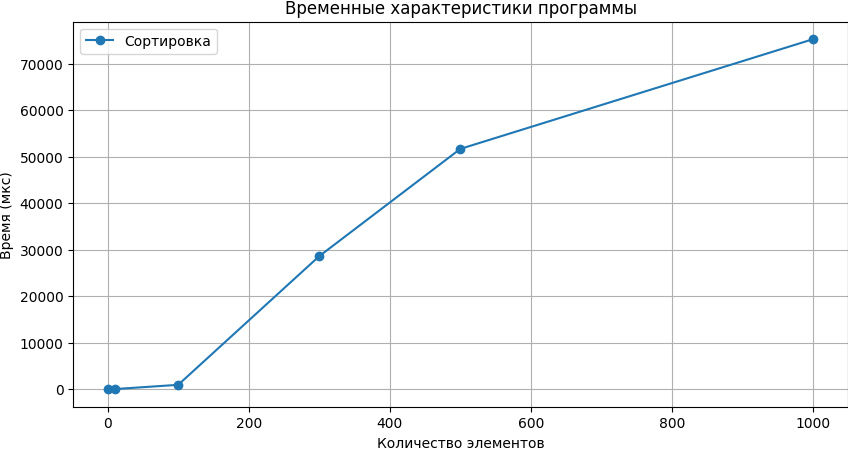


Рис. 6.2. График зависимости времени от количества при сортировке

Это **сортировка вставками** (Insertion Sort). Этот алгоритм работает следующим образом:

1. Он начинает с первого элемента и перебирает элементы с индексом 1 до конца массива.
2. Для каждого элемента (внутри цикла for на i), он перемещает элемент в "правильное" место среди предыдущих элементов с помощью внутреннего цикла while.
3. Внутренний цикл while сравнивает текущий элемент с предыдущими элементами и сдвигает их вправо, пока не находит правильное место для текущего элемента.
4. После того как подходящее место найдено, элемент вставляется в нужную позицию.

### Как это работает:

* **Внешний цикл** (for) отвечает за перебор всех элементов в коллекции начиная с 1-го (первый элемент по умолчанию считается отсортированным).
* **Внутренний цикл** (while) сравнивает текущий элемент с предыдущими элементами и сдвигает их вправо, если они не соответствуют порядку сортировки.
* Сравнение строк осуществляется с помощью метода to\_string(), что превращает объект StringRecord в строку для сравнения.

### Оценка производительности:

* Этот алгоритм имеет **время выполнения O(n²)** в худшем случае, что делает его менее эффективным для больших наборов данных по сравнению с другими алгоритмами сортировки (например, быстрой сортировкой или сортировкой слиянием). Однако, для небольших массивов или почти отсортированных данных сортировка вставками может быть достаточно быстрой.

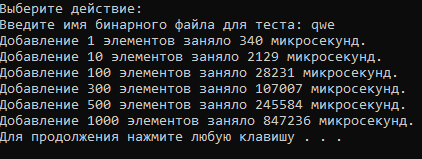


Рис. 6.3. Тестирование загрузки в бинарный файл

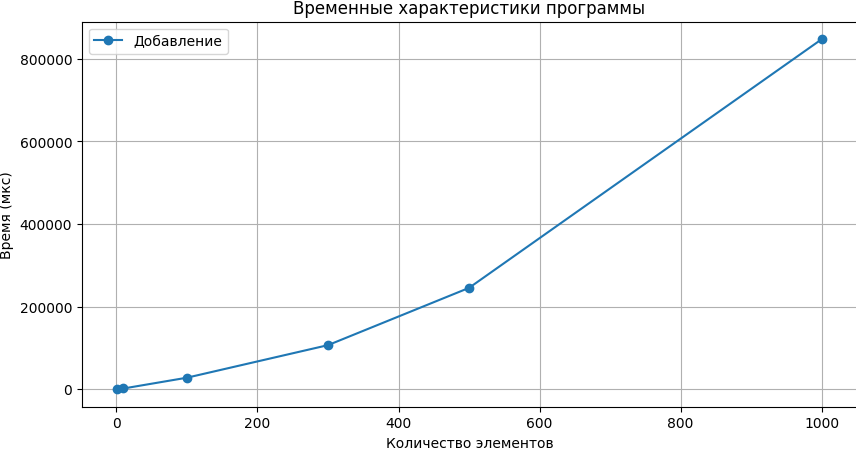


Рис. 6.4. График зависимости времени от добавления элементов

Время увеличивается с ростом числа добавляемых элементов. Это ожидаемо, потому что запись в файл — это операция, которая требует значительных затрат времени, особенно при большом объеме данных.

Наблюдается значительное увеличение времени с 100 до 1000 элементов. Это связано с тем, что файл с каждым добавлением становится больше, а система должна все больше времени затрачивать на запись в него.

# ВЫВОД

Разработка проекта продемонстрировала реализацию двоичного файла, производного от класса fstream, с поддержкой операций редактирования, сортировки и сохранения записей переменной длины. Работа над проектом позволила выявить сильные стороны подхода и области для оптимизации.

#### **1. Архитектура программы**

Проект представляет собой текстовый редактор с промежуточным хранением данных в двоичном файле. Данные организованы в виде массива указателей на строки переменной длины. Структура данных в файле включает:

* Заголовок файла, содержащий:
  + размерность массива указателей,
  + текущее количество записей,
  + смещение массива указателей.
* Массив указателей на записи.
* Сами записи, каждая из которых состоит из целого числа (длина строки) и последовательности символов строки.

Эта структура обеспечивает компактное представление данных и возможность быстрого доступа к любой записи.

#### **2. Функциональность программы**

Реализованы ключевые операции работы с двоичным файлом, включая:

1. **Создание и заполнение файла**: Возможность создать новый двоичный файл или загрузить уже существующий.
2. **Редактирование записей**: Добавление, удаление, изменение и вставка строк на произвольные позиции с автоматической перерасчетом указателей.
3. **Сохранение и загрузка данных**: Конвертация данных между двоичным и текстовым форматом.
4. **Сортировка**: Реализован алгоритм сортировки вставками, позволяющий упорядочить строки по возрастанию или убыванию.

#### **3. Производительность**

Проведено тестирование временных характеристик программы. Полученные результаты показывают:

* **Добавление строк в бинарный файл**:  
  С увеличением количества добавляемых строк наблюдается линейное увеличение времени операции. Например, добавление 1 строки занимает около 1493 микросекунд, а добавление 1000 строк — 871060 микросекунд. Это связано с необходимостью перерасчета указателей и сохранения структуры файла при каждом добавлении.
* **Сортировка записей**:  
  Время выполнения сортировки зависит от количества элементов и соответствует сложности алгоритма вставок (O(n2)O(n^2)O(n2)). Сортировка 1 строки выполняется практически мгновенно, в то время как для 1000 строк время значительно возрастает. Алгоритм может быть заменен на более эффективный для улучшения производительности.

#### **4. Удобство использования**

Пользовательский интерфейс реализован в виде текстового меню. Возможность выполнять операции через простой интерфейс делает программу доступной для пользователей с минимальными навыками работы с файлами. Использование unique\_ptr исключает утечки памяти при управлении объектами, что повышает надежность.

#### **5. Области для улучшений**

* **Оптимизация ввода-вывода**:  
  Уменьшение количества операций записи при добавлении строк и оптимизация работы с указателями может снизить затраты времени.
* **Улучшение алгоритмов**:  
  Использование более сложных алгоритмов сортировки, таких как быстрая сортировка или сортировка слиянием, позволит ускорить обработку больших объемов данных.
* **Расширение функциональности**:  
  Возможность поиска записей, по ключевым словам, группового редактирования и дополнительных форматов экспорта может сделать программу более универсальной.

**Заключение**

Проект успешно реализует задачи создания, редактирования и работы с бинарным файлом. Проведенное тестирование подтвердило работоспособность программы, а анализ временных характеристик выявил пути для дальнейшего улучшения производительности.

# СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

 Страуструп, Б. Программирование на C++ / Б. Страуструп. – М.: Издательство "Диалектика", 2013. – 1008 с.

 C++: Основы работы с файлами [Электронный ресурс]. – Режим доступа: https://cppstudio.ru/articles/reading-and-writing-binary-files-in-c-plus-plus/

 Метанит. Основы работы с файлами в C++ [Электронный ресурс]. – Режим доступа: https://metanit.com/cpp/tutorial/2.9.php

 Хабр. Понимание и использование указателей и ссылок в C++ [Электронный ресурс]. – Режим доступа: https://habr.com/ru/post/343682/

 Хабр. Как работает динамическое выделение памяти в C++ [Электронный ресурс]. – Режим доступа: https://habr.com/ru/post/483420/

 Stack Overflow на русском [Электронный ресурс]. – Режим доступа: <https://ru.stackoverflow.com/>

 C++ форум [Электронный ресурс]. – Режим доступа: <https://cppforum.ru/>

 C++: Работа с потоками / П. Григорьев [Электронный ресурс]. – Режим доступа: https://cpp-home.ru/articles/streams/

# ПРИЛОЖЕНИЕ А

**Листинг кода программы:**

main.cpp

#define \_CRT\_SECURE\_NO\_WARNINGS // Отключает предупреждения о небезопасных функциях в Visual Studio

#include "BinaryFile.h" // Подключение заголовочного файла для работы с BinaryFile

#include <conio.h> // Для использования функции \_getch() для ввода символов

#include <windows.h> // Для использования функции system("cls") для очистки консоли

#include <locale.h> // Для установки локализации

#include <iostream> // Для ввода и вывода

#include <vector> // Для std::vector

#include <string> // Для std::string

#include <random> // Для std::random\_device, std::mt19937, std::uniform\_int\_distribution

#include <algorithm> // Для std::sort

#include <chrono> // Для измерения времени выполнения

using namespace std; // Упрощает использование элементов стандартного пространства имен

vector<string> generateRandomStrings(size\_t count, size\_t length = 10) {

vector<string> strings;

random\_device rd;

mt19937 gen(rd());

uniform\_int\_distribution<> dis('a', 'z');

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

string str;

for (size\_t j = 0; j < length; ++j) {

str += static\_cast<char>(dis(gen));

}

strings.push\_back(str);

}

return strings;

}

void testSortingPerformance() {

vector<size\_t> sizes = { 1, 10, 100, 300, 500, 1000 };

for (size\_t size : sizes) {

// Создаем вектор с числами от 1 до size

vector<int> data(size);

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

data[i] = i + 1; // Заполняем числами от 1 до size

}

// Создаем объект BinaryFile

BinaryFile bin\_file;

// Добавляем данные в объект BinaryFile

for (const auto& num : data) {

bin\_file.add\_record(to\_string(num)); // Добавляем числа как строки

}

auto start = chrono::high\_resolution\_clock::now();

// Сортируем данные с помощью вашей функции сортировки

bin\_file.sort\_records(true); // Сортировка по возрастанию

auto end = chrono::high\_resolution\_clock::now();

cout << "Сортировка " << size << " элементов заняла "

<< chrono::duration\_cast<chrono::microseconds>(end - start).count()

<< " микросекунд.\n";

}

}

void testInsertionPerformance(const string& filename) {

vector<size\_t> sizes = { 1, 10, 100, 300, 500, 1000 };

for (size\_t size : sizes) {

BinaryFile file(filename.c\_str(), ios::out | ios::trunc);

auto start = chrono::high\_resolution\_clock::now();

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

file.add\_record("Случайная запись " + to\_string(i));

}

auto end = chrono::high\_resolution\_clock::now();

file.save(); // Сохраняем изменения

cout << "Добавление " << size << " элементов заняло "

<< chrono::duration\_cast<chrono::microseconds>(end - start).count()

<< " микросекунд.\n";

}

}

// Функция для вывода меню пользователя

void printMenu()

{

// Печать пунктов меню

cout << "\n Меню \n";

cout << "||1|| Создать новый двоичный файл\n";

cout << "||2|| Загрузить двоичный файл\n";

cout << "||3|| Управление записями\n";

cout << "||4|| Показать записи\n";

cout << "||5|| Сохранить двоичный файл\n";

cout << "||6|| Загрузить из текстового файла\n";

cout << "||7|| Сохранить в текстовый файл\n";

cout << "||8|| Cортировка\n";

cout << "||9|| Выход\n\n";

cout << "Выберите действие:\n";

}

// Основная функция программы, обеспечивающая работу меню

void menu()

{

setlocale(LC\_ALL, "Rus"); // Устанавливает русскую локализацию для корректного отображения текста

unique\_ptr<BinaryFile> bin\_file; // Умный указатель на объект BinaryFile, для автоматического управления памятью

bool running = true; // Флаг для работы цикла основного меню

// Главный цикл программы

while (running)

{

system("cls"); // Очищает консоль перед отображением меню

// Если двоичный файл загружен, отображаем его содержимое

if (bin\_file)

{

try {

cout << "Содержимое файла:\n";

bin\_file->display\_records(); // Вызов метода для отображения записей из файла

cout << endl;

}

catch (const exception& e) {

// Обработка ошибок, связанных с отображением

cout << "Ошибка при отображении записей: " << e.what() << endl;

}

}

else

{

// Если файл не загружен, выводим сообщение

cout << "Файл не загружен.\n";

}

printMenu(); // Выводим меню пользователя

char choice = \_getch(); // Ожидаем выбора пользователя без необходимости нажимать Enter

try

{

switch (choice)

{

case '1': {

// Создание нового двоичного файла

cout << "Введите имя нового файла: ";

string filename;

cin >> filename;

// Проверяем, существует ли файл с таким именем

ifstream file\_check(filename);

if (file\_check.is\_open()) {

cout << "Ошибка: файл с таким именем уже существует.\n";

file\_check.close();

}

else {

try {

bin\_file = make\_unique<BinaryFile>(filename.c\_str(), ios::out | ios::trunc); // Создаем новый файл

cout << "Файл успешно создан.\n";

log("Файл " + filename + " успешно создан"); // Логируем удаление

}

catch (const exception& e) {

cout << "Ошибка при создании файла: " << e.what() << endl;

}

}

system("pause"); // Пауза перед возвратом в меню

break;

}

case '2': {

// Загрузка существующего двоичного файла

cout << "Введите имя файла для загрузки: ";

string filename;

cin >> filename;

try {

bin\_file = make\_unique<BinaryFile>(filename.c\_str(), ios::in | ios::out); // Загружаем файл

cout << "Файл загружен.\n";

log("Файл " + filename + " успешно загружен");

}

catch (const exception& e) {

cout << "Ошибка при загрузке файла: " << e.what() << endl;

}

system("pause");

break;

}

case '3': {

// Управление записями в файле

if (!bin\_file)

{

// Если файл не загружен, выводим сообщение

cout << "Сначала создайте или загрузите файл.\n";

system("pause");

break;

}

// Подменю для управления записями

cout << "||1|| Добавить запись\n"

<< "||2|| Вставить запись по индексу\n"

<< "||3|| Удалить запись\n"

<< "||4|| Редактировать запись\n"

<< "Выберите действие: ";

char sub\_choice = \_getch(); // Ввод подменю

try {

switch (sub\_choice)

{

case '1': {

// Добавление новой записи

cout << "Введите строку для добавления: ";

string record;

if (cin.peek() == '\n') {

cin.ignore();

}

getline(cin, record); // Считываем строку с пробелами

bin\_file->add\_record(record); // Добавляем запись в файл

cout << "Запись добавлена.\n";

break;

}

case '2': {

// Вставка записи по индексу

cout << "Введите индекс: ";

int index;

if (!(cin >> index)) {

throw runtime\_error("Ошибка ввода: требуется число");

}

cout << "Введите строку для вставки: ";

string record;

cin.ignore();

getline(cin, record);

bin\_file->insert\_record(index, record); // Вставляем запись

cout << "Запись вставлена.\n";

break;

}

case '3': {

// Удаление записи по индексу

cout << "Введите индекс для удаления: ";

int index;

if (!(cin >> index)) {

throw runtime\_error("Ошибка ввода: требуется число");

}

bin\_file->delete\_record(index); // Удаляем запись

cout << "Запись удалена.\n";

break;

}

case '4': {

// Редактирование записи

cout << "Введите индекс для редактирования: ";

int index;

if (!(cin >> index)) {

throw runtime\_error("Ошибка ввода: требуется число");

}

cout << "Введите новую строку: ";

string record;

cin.ignore();

getline(cin, record);

bin\_file->edit\_record(index, record); // Редактируем запись

cout << "Запись обновлена.\n";

break;

}

default:

cout << "Неверный выбор.\n";

break;

}

}

catch (const exception& e) {

// Обработка ошибок в управлении записями

cout << "Ошибка при работе с записями: " << e.what() << endl;

}

system("pause");

break;

}

case '4': {

// Показать все записи

if (!bin\_file)

{

cout << "Сначала создайте или загрузите файл.\n";

system("pause");

break;

}

try {

bin\_file->display\_records(); // Отображаем записи

}

catch (const exception& e) {

cout << "Ошибка при отображении записей: " << e.what() << endl;

}

system("pause");

break;

}

case '5': {

// Сохранение двоичного файла

if (!bin\_file)

{

cout << "Сначала создайте или загрузите файл.\n";

system("pause");

break;

}

try {

bin\_file->save(); // Сохраняем данные в файл

cout << "Файл сохранен.\n";

log("Файл успешно сохранен");

}

catch (const exception& e) {

cout << "Ошибка при сохранении файла: " << e.what() << endl;

}

system("pause");

break;

}

case '6': {

// Загрузка данных из текстового файла

if (!bin\_file) {

cout << "Сначала создайте или загрузите файл.\n";

system("pause");

break;

}

try {

cout << "Введите имя текстового файла для загрузки: ";

string text\_filename;

cin >> text\_filename;

bin\_file->load\_from\_text(text\_filename); // Загружаем текстовые данные

cout << "Загружено из текстового файла.\n";

}

catch (const exception& e) {

cout << "Ошибка при загрузке текстового файла: " << e.what() << endl;

}

system("pause");

break;

}

case '7': {

// Сохранение данных в текстовый файл

if (!bin\_file)

{

cout << "Сначала создайте или загрузите файл.\n";

system("pause");

break;

}

try {

cout << "Введите имя текстового файла для сохранения: ";

string text\_filename;

cin >> text\_filename;

bin\_file->save\_to\_text(text\_filename); // Сохраняем данные в текстовый файл

cout << "Сохранено в текстовый файл.\n";

}

catch (const exception& e) {

cout << "Ошибка при сохранении текстового файла: " << e.what() << endl;

}

system("pause");

break;

}

case '8': {

if (!bin\_file) {

cout << "Сначала создайте или загрузите файл.\n";

system("pause");

break;

}

// Подменю сортировки

cout << "Выберите порядок сортировки:\n"

<< "||1|| По возрастанию\n"

<< "||2|| По убыванию\n"

<< "Ваш выбор: ";

char sort\_choice = \_getch();

try {

if (sort\_choice == '1') {

bin\_file->sort\_records(true); // Сортировка по возрастанию

cout << "Записи отсортированы по возрастанию.\n";

log("Записи отсортированы по возрастанию");

}

else if (sort\_choice == '2') {

bin\_file->sort\_records(false); // Сортировка по убыванию

cout << "Записи отсортированы по убыванию.\n";

log("Записи отсортированы по убыванию");

}

else {

cout << "Неверный выбор.\n";

}

}

catch (const exception& e) {

cout << "Ошибка при сортировке: " << e.what() << endl;

}

system("pause");

break;

}

case '9':

// Выход из программы

running = false;

break;

case 'q': {

cout << "Тест сортировки:\n";

testSortingPerformance();

system("pause");

break;

}

case 'w': {

cout << "Введите имя бинарного файла для теста: ";

string filename;

cin >> filename;

testInsertionPerformance(filename);

system("pause");

break;

}

default:

// Обработка неверного ввода

cout << "Неверный ввод.\n";

system("pause");

break;

}

}

catch (const exception& e)

{

// Обработка ошибок верхнего уровня

cout << "Ошибка: " << e.what() << endl;

system("pause");

}

}

}

// Точка входа в программу

int main()

{

menu(); // Запускает меню

return 0; // Завершение программы с кодом успешного выполнения

}

BinaryFile.cpp

#include "BinaryFile.h"

// Конструктор по умолчанию для объекта StringRecord

// Инициализирует переменные length (длина строки) в 0 и data (указатель на данные) в nullptr

StringRecord::StringRecord() : length(0), data(nullptr) {}

// Конструктор с параметром для инициализации StringRecord строкой

// Преобразует строку в динамически выделенный массив символов

StringRecord::StringRecord(const string& str) {

length = static\_cast<int>(str.size()); // Устанавливаем длину строки

data = make\_unique<char[]>(length); // Выделяем память для хранения символов

copy(str.begin(), str.end(), data.get()); // Копируем строку в выделенную память

}

// Перемещающий конструктор для перемещения данных из другого объекта StringRecord

// Перемещает указатель на данные, а также сбрасывает длину у исходного объекта

StringRecord::StringRecord(StringRecord&& other) noexcept

: length(other.length), data(move(other.data)) {

other.length = 0; // Обнуляем длину у перемещаемого объекта

}

// Перемещающий оператор присваивания

// Перемещает данные из другого объекта StringRecord, сбрасывая длину у исходного объекта

StringRecord& StringRecord::operator=(StringRecord&& other) noexcept {

if (this != &other) { // Проверка на самоприсваивание

length = other.length; // Копируем длину

data = move(other.data); // Перемещаем данные

other.length = 0; // Обнуляем длину у исходного объекта

}

return \*this;

}

// Сохраняет запись в поток (файл)

void StringRecord::save(ostream& os) const {

os.write(reinterpret\_cast<const char\*>(&length), sizeof(length)); // Записываем длину записи

os.write(data.get(), length); // Записываем данные строки

}

// Загружает запись из потока (файла)

void StringRecord::load(istream& is) {

is.read(reinterpret\_cast<char\*>(&length), sizeof(length)); // Читаем длину записи

data = make\_unique<char[]>(length); // Выделяем память для данных

is.read(data.get(), length); // Читаем данные строки

}

// Преобразует запись в строку

string StringRecord::to\_string() const {

return string(data.get(), length); // Возвращаем строку, используя данные из массива

}

// Оператор вывода для строки в поток

ostream& operator<<(ostream& os, const StringRecord& record) {

os << record.to\_string(); // Выводим строковое представление записи

return os;

}

// Оператор ввода для строки из потока

istream& operator>>(istream& is, StringRecord& record) {

string temp;

getline(is, temp); // Читаем строку

record = StringRecord(temp); // Создаем запись на основе считанной строки

return is;

}

#pragma region BinaryFile

// Конструктор для бинарного файла

// Открывает файл с указанным именем и режимом, загружает данные, если открытие прошло успешно

BinaryFile::BinaryFile(const string& filename, ios::openmode mode)

: array\_size(0), current\_count(0), pointer\_array\_offset(0) {

open(filename, mode | ios::binary); // Открытие файла в бинарном режиме

if (!is\_open()) {

throw runtime\_error("Не удалось открыть файл: " + filename); // Ошибка при открытии

}

if (mode & ios::in) {

load(); // Загружаем данные, если файл открыт для чтения

}

}

// Деструктор, который закрывает файл

BinaryFile::~BinaryFile() {

close(); // Закрывает файл при уничтожении объекта

}

// Функция логирования, записывающая сообщения в файл log.txt

void log(const string& message) {

// Получаем текущее время

auto now = chrono::system\_clock::now();

auto time\_t\_now = chrono::system\_clock::to\_time\_t(now); // Преобразуем в time\_t

// Получаем локальное время

tm local\_tm;

localtime\_s(&local\_tm, &time\_t\_now); // Для безопасной работы с localtime в Windows

// Форматируем время в строку

stringstream time\_stream;

time\_stream << put\_time(&local\_tm, "%Y-%m-%d %H:%M:%S"); // Форматируем дату и время

// Открытие файла для добавления сообщений

ofstream log\_file("log.txt", ios::app);

if (log\_file.is\_open()) {

log\_file << time\_stream.str() << " - " << message << endl; // Записываем время и сообщение

}

}

// Добавление новой записи в конец файла

void BinaryFile::add\_record(const string& str) {

try {

records.emplace\_back(str); // Добавляем запись

update\_pointers(); // Обновляем указатели

log("Добавлена запись: " + str); // Логируем добавление записи

}

catch (const exception& e) {

throw runtime\_error("Ошибка при добавлении записи: "s + e.what()); // Обработка ошибок

}

}

// Вставка записи по индексу

void BinaryFile::insert\_record(size\_t index, const string& str) {

try {

if (index > records.size()) {

throw out\_of\_range("Индекс вне диапазона: " + to\_string(index)); // Проверка на корректность индекса

}

records.insert(records.begin() + index, StringRecord(str)); // Вставляем запись

update\_pointers(); // Обновляем указатели

log("Добавлена запись по индексу " + to\_string(index) + " : " + str); // Логируем добавление

}

catch (const exception& e) {

throw runtime\_error("Ошибка при вставке записи: "s + e.what()); // Обработка ошибок

}

}

// Удаление записи по индексу

void BinaryFile::delete\_record(size\_t index) {

try {

string old\_record = records[index].to\_string(); // Сохраняем старую запись для лога

if (index >= records.size()) {

throw out\_of\_range("Индекс вне диапазона: " + to\_string(index)); // Проверка на корректность индекса

}

records.erase(records.begin() + index); // Удаляем запись

update\_pointers(); // Обновляем указатели

log("Запись по индексу " + to\_string(index) + " удалена: " + old\_record); // Логируем удаление

}

catch (const exception& e) {

throw runtime\_error("Ошибка при удалении записи: "s + e.what()); // Обработка ошибок

}

}

// Редактирование записи по индексу

void BinaryFile::edit\_record(size\_t index, const string& str) {

try {

if (index >= records.size()) {

throw out\_of\_range("Индекс вне диапазона: " + to\_string(index)); // Проверка на корректность индекса

}

string old\_record = records[index].to\_string(); // Сохраняем старую запись для лога

records[index] = StringRecord(str); // Заменяем запись

update\_pointers(); // Обновляем указатели

log("Запись по индексу " + to\_string(index) + " изменена: " + old\_record + " -> " + str); // Логируем изменение

}

catch (const exception& e) {

throw runtime\_error("Ошибка при редактировании записи: "s + e.what()); // Обработка ошибок

}

}

// Сохранение всех записей в файл

void BinaryFile::save() {

// Проверка открытия файла

if (!is\_open()) {

throw runtime\_error("Файл не открыт для записи");

}

// Очистка потока и установка указателя записи в начало

clear();

seekp(0, ios::beg);

// Записываем заголовок

array\_size = static\_cast<int>(records.size());

current\_count = static\_cast<int>(records.size());

pointer\_array\_offset = sizeof(array\_size) + sizeof(current\_count) + sizeof(pointer\_array\_offset);

write(reinterpret\_cast<const char\*>(&array\_size), sizeof(array\_size));

write(reinterpret\_cast<const char\*>(&current\_count), sizeof(current\_count));

write(reinterpret\_cast<const char\*>(&pointer\_array\_offset), sizeof(pointer\_array\_offset));

// Обновляем указатели

update\_pointers();

// Записываем массив указателей

for (size\_t pointer : pointers) {

write(reinterpret\_cast<const char\*>(&pointer), sizeof(pointer));

}

// Записываем строки

for (const auto& record : records) {

record.save(\*this);

}

// Принудительный сброс буфера на диск

flush();

}

void BinaryFile::load() {

// Очищаем текущие данные

clear();

// Устанавливаем указатель на начало файла

seekg(0, std::ios::beg);

// Проверка на открытие файла

if (!is\_open()) {

throw std::runtime\_error("Не удалось открыть файл для загрузки.");

}

// Чтение заголовка

if (!read(reinterpret\_cast<char\*>(&array\_size), sizeof(array\_size))) {

throw std::runtime\_error("Ошибка при чтении array\_size.");

}

if (!read(reinterpret\_cast<char\*>(&current\_count), sizeof(current\_count))) {

throw std::runtime\_error("Ошибка при чтении current\_count.");

}

if (!read(reinterpret\_cast<char\*>(&pointer\_array\_offset), sizeof(pointer\_array\_offset))) {

throw std::runtime\_error("Ошибка при чтении pointer\_array\_offset.");

}

// Чтение указателей

pointers.resize(array\_size);

if (!read(reinterpret\_cast<char\*>(pointers.data()), array\_size \* sizeof(size\_t))) {

throw std::runtime\_error("Ошибка при чтении указателей");

}

// Чтение записей

records.clear();

try {

for (int i = 0; i < current\_count; ++i) {

StringRecord record;

record.load(\*this); // Загружаем запись

records.emplace\_back(std::move(record)); // Добавляем запись

}

}

catch (const std::exception& e) {

throw std::runtime\_error(std::string("Ошибка при загрузке записей: ") + e.what());

}

}

// Обновление указателей для каждой записи

void BinaryFile::update\_pointers() {

pointers.clear(); // Очищаем старые указатели

// Устанавливаем начальное смещение для первой записи

size\_t offset = pointer\_array\_offset + array\_size \* sizeof(size\_t);

// Заполняем массив указателей

for (const auto& record : records) {

pointers.push\_back(offset);

offset += sizeof(int) + record.to\_string().size(); // Смещение для следующей записи

}

}

// Вывод всех записей на экран

void BinaryFile::display\_records() const {

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

cout << i + 1 << ". " << records[i].to\_string() << endl; // Печатаем индекс и содержимое записи

}

}

void BinaryFile::sort\_records(bool ascending) {

for (size\_t i = 1; i < records.size(); ++i) {

StringRecord key = std::move(records[i]); // Используем перемещение для оптимизации

size\_t j = i;

// Сравниваем строки с учётом порядка сортировки

while (j > 0 &&

(ascending ? records[j - 1].to\_string() > key.to\_string()

: records[j - 1].to\_string() < key.to\_string())) {

records[j] = std::move(records[j - 1]); // Перемещение для предотвращения копирования

--j;

}

records[j] = std::move(key); // Вставляем текущий элемент на своё место

}

}

// Загрузка данных из текстового файла

void BinaryFile::load\_from\_text(const string& text\_filename) {

try {

ifstream text\_file(text\_filename); // Открываем текстовый файл

if (!text\_file.is\_open()) {

throw runtime\_error("Не удалось открыть текстовый файл: " + text\_filename);

}

records.clear(); // Очищаем текущие записи

StringRecord record;

while (text\_file >> record) {

records.emplace\_back(std::move(record)); // Добавляем запись из файла

}

if (text\_file.bad()) {

throw runtime\_error("Ошибка при чтении текстового файла");

}

update\_pointers(); // Обновляем указатели

log("Данные загружены из текстового файла: " + text\_filename); // Логируем

}

catch (const exception& e) {

log("Ошибка при загрузке из текстового файла: " + text\_filename + ". Причина: " + e.what());

throw runtime\_error("Ошибка при загрузке из текстового файла: "s + e.what());

}

}

// Сохранение записей в текстовый файл

void BinaryFile::save\_to\_text(const string& text\_filename) const {

try {

ofstream text\_file(text\_filename); // Открываем текстовый файл для записи

if (!text\_file.is\_open()) {

throw runtime\_error("Не удалось открыть текстовый файл для записи: " + text\_filename);

}

for (const auto& record : records) {

text\_file << record << endl; // Записываем каждую запись в файл

}

if (text\_file.bad()) {

throw runtime\_error("Ошибка при записи в текстовый файл");

}

log("Данные сохранены в текстовый файл: " + text\_filename); // Логируем

}

catch (const exception& e) {

log("Ошибка при сохранении в текстовый файл: " + text\_filename + ". Причина: " + e.what());

throw runtime\_error("Ошибка при сохранении в текстовый файл: "s + e.what());

}

}

// Перегрузка оператора вывода для бинарного файла

ostream& operator<<(ostream& os, const BinaryFile& file) {

os << file.records.size() << '\n'; // Выводим количество записей

for (const auto& record : file.records) {

os << record << '\n'; // Выводим каждую запись

}

return os;

}

// Перегрузка оператора ввода для бинарного файла

istream& operator>>(istream& is, BinaryFile& file) {

file.records.clear(); // Очищаем текущие записи

size\_t count;

is >> count; // Читаем количество записей

is.ignore(); // Игнорируем символ новой строки

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

StringRecord record;

if (is >> record) {

file.records.emplace\_back(move(record)); // Добавляем запись

}

else {

throw runtime\_error("Ошибка при чтении записи из текстового файла"); // Обработка ошибок

}

}

return is;

}

// Перегрузка оператора вывода в бинарный файл

ofstream& operator<<(ofstream& ofs, const BinaryFile& file) {

int count = static\_cast<int>(file.records.size());

ofs.write(reinterpret\_cast<const char\*>(&count), sizeof(count)); // Записываем количество записей

for (const auto& record : file.records) {

record.save(ofs); // Сохраняем каждую запись

}

return ofs;

}

// Перегрузка оператора ввода из бинарного файла

ifstream& operator>>(ifstream& ifs, BinaryFile& file) {

file.records.clear(); // Очищаем текущие записи

int count = 0;

ifs.read(reinterpret\_cast<char\*>(&count), sizeof(count)); // Читаем количество записей

if (count < 0) {

throw runtime\_error("Некорректное количество записей в бинарном файле");

}

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

StringRecord record;

record.load(ifs); // Загружаем запись

file.records.emplace\_back(move(record)); // Добавляем запись

}

return ifs;

}

#pragma endregion

BinaryFile.h

#ifndef BINARYFILE\_H

#define BINARYFILE\_H

#include <iostream>

#include <fstream>

#include <vector>

#include <string>

#include <iomanip>

#include <ctime>

#include <sstream>

#include <memory>

#include <stdexcept>

#include <chrono>

using namespace std;

// Упаковка структуры для точного соответствия бинарному формату файла

#pragma pack(push, 1)

void log(const string& message);

// Класс, представляющий строку переменной длины

class StringRecord {

private:

int length; // Длина строки

unique\_ptr<char[]> data; // Данные строки

public:

StringRecord();

explicit StringRecord(const string& str);

void save(ostream& os) const;

void load(istream& is);

string to\_string() const;

friend ostream& operator<<(ostream& os, const StringRecord& record);

friend istream& operator>>(istream& is, StringRecord& record);

// Запрещаем копирование

StringRecord(const StringRecord&) = delete;

StringRecord& operator=(const StringRecord&) = delete;

// Разрешаем перемещение

StringRecord(StringRecord&& other) noexcept;

StringRecord& operator=(StringRecord&& other) noexcept;

};

#pragma pack(pop)

// Класс, представляющий двоичный файл

class BinaryFile : public fstream {

private:

int array\_size; // Размер массива указателей

int current\_count; // Текущее количество указателей

size\_t pointer\_array\_offset; // Смещение массива указателей

vector<StringRecord> records; // Список строк

vector<size\_t> pointers; // Смещения записей в файле

void update\_pointers(); // Пересчёт указателей

public:

BinaryFile() = default;

explicit BinaryFile(const string& filename, ios::openmode mode);

~BinaryFile();

void add\_record(const string& str);

void insert\_record(size\_t index, const string& str);

void delete\_record(size\_t index);

void edit\_record(size\_t index, const string& str);

void save();

void load();

void sort\_records(bool ascending);

void display\_records() const;

void load\_from\_text(const string& text\_filename);

void save\_to\_text(const string& text\_filename) const;

friend ostream& operator<<(ostream& os, const BinaryFile& file);

friend istream& operator>>(istream& is, BinaryFile& file);

friend ofstream& operator<<(ofstream& ofs, const BinaryFile& file);

friend ifstream& operator>>(ifstream& ifs, BinaryFile& file);

};

#endif // BINARYFILE\_H