

7- Практическое задание — работы со стандартной библиотекой шаблонов и файлового ввода-вывода

Table of Contents

Цель	2
Основные технологии	2
1. Система Управления Сотрудниками с использованием STL.....	5
2. Система Управления Инвентарем.....	9
3. Чтение и Запись Данных Сотрудников в Файл.....	12
4. Личный Каталог Библиотеки	15
Самостоятельная задача: Система управления библиотекой	17

Цель

Цель данной лабораторной работы - обеспечить студентов практическими навыками в области объектно-ориентированного программирования на языке C++. Студенты научатся создавать классы и объекты, использовать наследование и полиморфизм, работать со стандартной библиотекой шаблонов для эффективного управления данными. Они также освоят основы файлового ввода-вывода для сохранения и загрузки состояния программы. Через ряд задач, от простых до более сложных, студенты разовьют умение интегрировать эти концепции для создания полноценных приложений.

Основные технологии

STL (Стандартная Библиотека Шаблонов): Коллекция обобщённых классов шаблонов и функций, которая включает в себя алгоритмы, контейнеры, итераторы и другие компоненты. В этой задаче мы сосредоточимся на использовании vector, list и map.

1- Vector: Динамический массив, позволяющий хранить элементы в непрерывном блоке памяти.

```
#include <iostream>
#include <vector>
#include "locale.h"

int main() {
    setlocale(LC_ALL, "Russian");

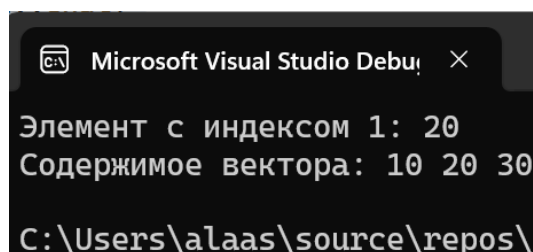
    // Создание вектора для хранения целых чисел
    std::vector<int> numbers;

    // Добавление элементов в вектор
    numbers.push_back(10);
    numbers.push_back(20);
    numbers.push_back(30);

    // Доступ к элементам с использованием оператора []
    std::cout << "Элемент с индексом 1: " << numbers[1] << std::endl;

    // Итерация по вектору с использованием цикла range-based for
    std::cout << "Содержимое вектора: ";
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```



2. List: Двусвязный список, обеспечивающий быстрое вставку и удаление элементов.

```
#include <iostream>
#include <list>
#include "locale.h"

int main() {
    setlocale(LC_ALL, "Russian");
```

```

// Создание списка для хранения целых чисел
std::list<int> numbers;

// Добавление элементов в список
numbers.push_back(10); // Добавление в конец списка
numbers.push_back(20);
numbers.push_front(5); // Добавление в начало списка

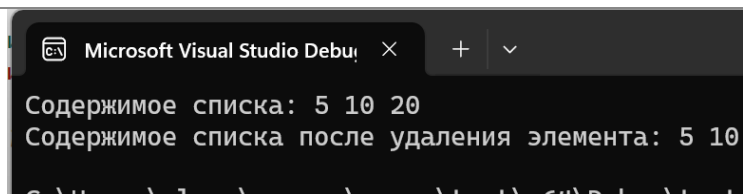
// Итерация по списку с использованием итератора для доступа к элементам
std::cout << "Содержимое списка: ";
for (auto it = numbers.begin(); it != numbers.end(); ++it) {
    std::cout << *it << " ";
}
std::cout << std::endl;

// Удаление элемента из списка
numbers.remove(20); // Удаление всех вхождений элемента со значением 20

// Повторная итерация по списку для вывода его содержимого после удаления элемента
std::cout << "Содержимое списка после удаления элемента: ";
for (int num : numbers) {
    std::cout << num << " ";
}
std::cout << std::endl;

return 0;
}

```



```

Microsoft Visual Studio Debug Console
Содержимое списка: 5 10 20
Содержимое списка после удаления элемента: 5 10
C:\Users\alaa\source\repos\test\test\Debug\test

```

3. Map: Ассоциативный массив, который хранит элементы в виде пар ключ-значение.

```

#include <iostream>
#include <map>
#include <string>
#include "locale.h"

int main() {
    setlocale(LC_ALL, "Russian");

    // Создание карты, где ключ - строка, а значение - int
    std::map<std::string, int> ageMap;

    // Добавление пар ключ-значение в карту
    ageMap["Иван"] = 25;
    ageMap["Елена"] = 30;
    ageMap["Алексей"] = 28;

    // Поиск элемента в карте по ключу и вывод его значения
    std::string name = "Елена";
    if (ageMap.find(name) != ageMap.end()) {
        std::cout << "Возраст " << name << ": " << ageMap[name] << std::endl;
    }
    else {
        std::cout << name << " не найден в карте." << std::endl;
    }

    // Итерация по карте и вывод всех пар ключ-значение
    std::cout << "Весь список возрастов:" << std::endl;
    for (const auto& pair : ageMap) {
        std::cout << pair.first << ": " << pair.second << std::endl;
    }

    return 0;
}

```

}

```

Microsoft Visual Studio Debug Console
Возраст Елена: 30
Весь список возрастов:
Алексей: 28
Елена: 30
Иван: 25
C:\Users\alaaas\source\repos\t

```

В C++ контейнеры `vector`, `list` и `map` представляют собой различные структуры данных, каждая из которых имеет свои особенности и предназначена для решения определенных задач. Рассмотрим их подробнее, чтобы понять различия и ситуации, в которых предпочтительно использовать тот или иной контейнер вместо обычных массивов или матриц.

Vector (Вектор):

Вектор — это динамический массив, который может изменять свой размер в процессе выполнения программы. Векторы обеспечивают быстрый доступ к элементам по индексу, как и массивы, но при этом позволяют добавлять и удалять элементы в конце контейнера без необходимости заново выделять память или перемещать все элементы, как это требуется при работе с обычными массивами.

Преимущества использования вектора по сравнению с массивом:

- 1- Динамическое изменение размера.
- 2- Удобство использования, включая методы добавления, удаления элементов и доступа к элементам.
- 3- Возможность использования в качестве массива с автоматическим управлением памятью.

List (Список)

Список в C++ — это двусвязный список, позволяющий эффективное добавление и удаление элементов как в начале, так и в середине или конце списка. По сравнению с вектором, списки обеспечивают более быструю вставку и удаление, но медленнее предоставляют доступ к элементам, так как для доступа к элементу требуется проход по ссылкам от начала или конца списка.

Преимущества использования списка по сравнению с массивом:

- 1- Быстрая вставка и удаление элементов в любом месте списка.
- 2- Возможность изменения размера без перевыделения памяти для всей структуры.

Map (Ассоциативный массив)

Map - это ассоциативный массив, который хранит данные в виде пар ключ-значение. Каждый ключ уникален, и элементы в map автоматически сортируются по ключам. Это позволяет быстро находить значение по ключу, но добавление, удаление и доступ к элементам может быть медленнее, чем в векторах, из-за необходимости поддерживать упорядоченность элементов.

Преимущества использования map по сравнению с массивом:

- Быстрый поиск элементов по ключу.
- Автоматическая сортировка элементов по ключу.
- Уникальность ключей, что обеспечивает однозначное соответствие между ключом и значением.

Почему выбирать их вместо массива или матрицы?

- 1- Динамичность размера: в отличие от статических массивов, размер `vector`, `list` и `map` может изменяться в процессе выполнения программы.
- 2- Удобство работы: Стандартные методы для добавления, удаления и доступа к элементам упрощают разработку и уменьшают вероятность ошибок.
- 3- Специализированные операции: каждый из этих контейнеров предоставляет уникальные возможности (быстрая вставка/удаление, быстрый поиск по ключу, автоматическая сортировка), которые могут быть критически важны для определенных алгоритмов или задач.

Выбор между `vector`, `list`, `map` и обычными массивами зависит от конкретных требований к производительности, типу хранимых данных и операций, которые вы планируете с ними выполнять.

Различие между `list` и `vector` в C++

Различие между `list` и `vector` в C++ заключается в их внутренней структуре и производительности различных операций. `list` представляет собой двусвязный список, что позволяет эффективно вставлять и удалять элементы в любой части списка, включая начало, середину и конец, без необходимости перемещения других элементов. Однако доступ к элементам списка происходит последовательно, что делает операцию поиска относительно медленной по сравнению с вектором. С другой стороны, `vector` работает как динамический массив, обеспечивая быстрый доступ к элементам по индексу благодаря непрерывному размещению в памяти. Но при добавлении или удалении элементов в начало или середину вектора, остальные элементы необходимо сдвигать, что может быть менее эффективно, особенно для больших объемов данных.

Когда предпочтительнее использовать `list` над `vector`:

- A. Когда требуется частое добавление и удаление элементов в середине коллекции.
- B. Когда порядок элементов важен, и вы часто выполняете операции, которые могут нарушить этот порядок в массиве или векторе.

Когда предпочтительнее использовать `vector` над `list`:

- A. Когда важен быстрый доступ к элементам по индексу.
- B. Когда добавление элементов происходит в основном в конец коллекции, что минимизирует необходимость сдвига элементов.
- C. Когда память и производительность являются критически важными, так как вектор обычно использует память более эффективно за счет непрерывного размещения данных.

Выбор между `list` и `vector` должен основываться на специфике задачи и требованиях к производительности операций, которые вы собираетесь наиболее часто выполнять с коллекцией.

1. Система Управления Сотрудниками с использованием STL

Цель задачи: создать простую систему управления сотрудниками для демонстрации использования `vector`, `list` и `map` из Стандартной Библиотеки Шаблонов (STL).

Шаги Реализации

Шаг 1: Определение Класа Сотрудника

Начнем с определения базового класса `Employee`, который будет содержать общие атрибуты и методы для сотрудников.

```

#include <iostream>
#include <vector>
#include <list>
#include <map>
#include <string>
#include "locale.h"

class Employee {
public:
    int id;
    std::string name;
    std::string department;

    Employee(int id, std::string name, std::string department) : id(id), name(name), department(department) {}

    virtual void display() {
        std::cout << "ID: " << id << ", Name: " << name << ", Department: " << department << std::endl;
    }
};

```

Шаг 2: Расширение Класса Сотрудника

Далее, создадим два производных класса: FullTimeEmployee и PartTimeEmployee, добавляя специфические атрибуты и методы.

```

class FullTimeEmployee : public Employee {
public:
    double salary;

    FullTimeEmployee(int id, std::string name, std::string department, double salary)
        : Employee(id, name, department), salary(salary) {}

    void display() override {
        Employee::display();
        std::cout << "Salary: " << salary << std::endl;
    }
};

class PartTimeEmployee : public Employee {
public:
    double hourlyRate;

    PartTimeEmployee(int id, std::string name, std::string department, double hourlyRate)
        : Employee(id, name, department), hourlyRate(hourlyRate) {}

    void display() override {
        Employee::display();
        std::cout << "Hourly Rate: " << hourlyRate << std::endl;
    }
};

```

Шаг 3: Использование STL Контейнеров

Используем vector для хранения списка сотрудников, list для управления названиями отделов, и map для ассоциации названий отделов с их сотрудниками.

```

int main() {
    setlocale(LC_ALL, "Russian");
    std::vector<Employee*> employees;
    std::list<std::string> departments;
    std::map<std::string, std::vector<Employee*>> departmentEmployees;

    // Добавление сотрудников и отделов в контейнеры
    employees.push_back(new FullTimeEmployee(1, "Иван Иванов", "Разработка", 50000));
    employees.push_back(new PartTimeEmployee(2, "Петр Петров", "Маркетинг", 300));

    departments.push_back("Разработка");
    departments.push_back("Маркетинг");
}

```

```

for (auto& emp : employees) {
    departmentEmployees[emp->department].push_back(emp);
}

// Вывод информации о сотрудниках
for (auto& emp : employees) {
    emp->display();
}

// Очистка динамически выделенной памяти
for (auto& emp : employees) {
    delete emp;
}

return 0;
}

```

Объяснение Кода

- Класс Employee является базовым классом, содержащим общие атрибуты для всех сотрудников.
- Классы FullTimeEmployee и PartTimeEmployee наследуются от Employee и добавляют специфические атрибуты и переопределяют метод display для вывода своих дополнительных данных.
- Используем vector для хранения указателей на объекты сотрудников, что позволяет нам хранить объекты производных классов.
- list используется для управления списком отделов, а map - для ассоциации отделов с их сотрудниками, демонстрируя, как можно организовать данные в памяти.
- В конце программы происходит очистка памяти, выделенной для объектов сотрудников.

Этот пример демонстрирует основные принципы ООП, такие как наследование, а также использование контейнеров STL для организации и управления данными в программе.



Ключевое слово `auto` в C++ используется для автоматического определения типа переменной компилятором на основе её инициализации. Это удобно, когда тип выражения сложен или длинен, например, при работе с итераторами или когда тип выражения очевиден из контекста и его указание является избыточным.

Символ `&` после `auto` указывает, что переменная является ссылкой на элемент коллекции, а не копией. Это позволяет работать непосредственно с элементами коллекции, изменяя их внутри цикла, без необходимости копирования.

В данном контексте:

```

for (auto& emp : employees) {
    departmentEmployees[emp->department].push_back(emp);
}

```

Цикл проходит по коллекции `employees`, где каждый элемент `emp` представляет собой ссылку на объект в этой коллекции (предполагается, что `employees` - это коллекция указателей на объекты сотрудников). Это позволяет эффективно работать с элементами коллекции без их копирования. Внутри цикла для каждого сотрудника `emp` происходит обращение к его отделу (`emp->department`) и добавление указателя на этого сотрудника (`emp`) в соответствующий вектор в `departmentEmployees`, который ассоциирует отделы с сотрудниками.

Использование `auto&` значительно упрощает код и делает его более читаемым, особенно в ситуациях, когда типы данных сложны или когда необходимо избежать копирования объектов при итерации по коллекции.

Что изменится, если не использовать `auto`:

- Требуется точное знание типа: Вам нужно будет знать и указывать конкретный тип элементов в коллекции. Это может быть неудобно, особенно при работе со сложными или вложенными типами, такими как итераторы контейнеров STL или когда тип элементов может измениться в результате рефакторинга кода.
- Увеличение вероятности ошибок: Явное указание типа увеличивает вероятность ошибок, особенно если тип элементов коллекции изменится. Вам придется вручную обновлять тип в каждом цикле `for`, где используется эта коллекция.
- Снижение читаемости и удобства поддержки: Код становится менее гибким и менее адаптируемым к изменениям. `auto` делает код более читаемым и упрощает его поддержку, поскольку уменьшает количество мест, которые необходимо изменить при модификации типов данных.

В общем, использование `auto` рекомендуется, когда тип переменной может быть однозначно определен из контекста и когда явное указание типа не добавляет дополнительной ясности. Это упрощает код, делает его более адаптируемым к изменениям и уменьшает вероятность ошибок, связанных с несоответствием типов.

2. Система Управления Инвентарем

Цель задачи: создать систему управления инвентарем, которая использует map для отслеживания запасов продуктов.

Шаги Реализации

Шаг 1: Определение Класа Продукта

Начнем с создания базового класса Product, который будет содержать атрибуты, общие для всех продуктов.

```
#include <iostream>
#include <map>
#include <string>
#include "locale.h"

class Product {
public:
    int productID;
    std::string productName;
    double price;

    Product(int productID, std::string productName, double price)
        : productID(productID), productName(productName), price(price) {}

    virtual void display() {
        std::cout << "Product ID: " << productID << ", Name: " << productName << ", Price: " << price <<
std::endl;
    }
};
```

Шаг 2: Создание Производных Классов Продуктов

Создадим производные классы для различных категорий продуктов, например, Electronics и Clothing.

```
class Electronics : public Product {
public:
    Electronics(int productID, std::string productName, double price)
        : Product(productID, productName, price) {}

    void display() override {
        Product::display();
        std::cout << "Category: Electronics" << std::endl;
    }
};

class Clothing : public Product {
public:
    Clothing(int productID, std::string productName, double price)
        : Product(productID, productName, price) {}

    void display() override {
        Product::display();
        std::cout << "Category: Clothing" << std::endl;
    }
};
```

Шаг 3: Использование Map для Управления Инвентарем

Далее, мы используем map для ассоциации идентификаторов продуктов с объектами продуктов и их запасами.

```
int main() {
    setlocale(LC_ALL, "Russian");
    std::map<int, std::pair<Product*, int>> inventory;
```

```

// Добавление продуктов в инвентарь
inventory[1] = std::make_pair(new Electronics(1, "Smartphone", 500.0), 10);
inventory[2] = std::make_pair(new Clothing(2, "T-Shirt", 20.0), 50);

// Вывод инвентаря
for (const auto& item : inventory) {
    std::cout << "Stock: " << item.second.second << " ";
    item.second.first->display();
}

// Очистка памяти
for (auto& item : inventory) {
    delete item.second.first;
}

return 0;
}

```

Объяснение Кода

- Класс Product служит базой для всех продуктов, содержит в себе основные атрибуты и метод display для отображения информации о продукте.
- Производные классы, такие как Electronics и Clothing, расширяют базовый класс, добавляя специфическую информацию, например, категорию продукта.
- Используя map, мы ассоциируем каждый продукт с его идентификатором и запасами, что позволяет нам эффективно управлять инвентарем.
- В конце, мы перебираем map для вывода информации о каждом продукте и его запасах, а затем освобождаем выделенную память.

Этот пример помогает понять, как можно использовать наследование для создания иерархии классов и map для эффективного управления данными в C++.



Давайте подробно разберем объявление `std::map<int, std::pair<Product*, int>> inventory;` и обсудим, почему именно такой тип данных может быть использован в определенных ситуациях.

- `std::map`: Это ассоциативный контейнер из стандартной библиотеки шаблонов (STL) в C++, который хранит элементы в виде пар ключ-значение. Каждый ключ в карте уникален, и элементы автоматически сортируются по ключам. Это обеспечивает быстрый доступ к значениям по ключам и эффективное добавление и удаление элементов с сохранением порядка сортировки.
- `<int, std::pair<Product*, int>>`: Это параметры шаблона для `std::map`, указывающие типы ключа и значения соответственно. В данном случае ключом является `int`, что может представлять, например, идентификатор продукта. Значением является пара `std::pair<Product*, int>`, где `Product*` — это указатель на объект `Product`, представляющий продукт, а `int` — дополнительное целочисленное значение, которое может, например, обозначать количество данного продукта на складе.
- Использование символа `*` после типа `Product` в контексте `std::pair<Product*, int>` указывает на то, что в паре используется указатель на объект типа `Product`, а не сам объект. В C++ указатели представляют собой переменные, содержащие адреса других переменных или объектов в памяти. В данном случае `Product*` означает указатель на объект `Product`. Указатели позволяют динамически выделять и освобождать память в куче во время выполнения программы. Это дает возможность создавать объекты, размер или количество которых заранее неизвестно или может изменяться. В контексте управления инвентарем это может быть полезно для создания новых объектов `Product`, когда они добавляются в инвентарь, и управления их жизненным циклом. Использование указателей на базовый

класс позволяет хранить в контейнерах объекты производных классов, сохраняя при этом возможность использования полиморфных функций. Если `Product` является базовым классом для различных типов продуктов, использование указателей на `Product` позволит эффективно работать с разными типами продуктов в единой структуре данных.

- `std::pair<Product*, int>`: Это стандартный контейнер STL, который хранит два значения, возможно, разных типов. В данном контексте он используется для группировки двух связанных данных — указателя на продукт и количества этого продукта. `std::pair` удобен для хранения таких связанных данных, поскольку обеспечивает простой доступ к каждому элементу пары через его члены `first` и `second`.

Почему используется именно такая структура

Эта структура данных выбрана по нескольким причинам:

- Организация данных: `std::map` с парой в качестве значения позволяет удобно организовать и связать различные типы данных, такие как объект продукта и его количество. Это облегчает управление инвентарем, где каждому продукту соответствует уникальный идентификатор и количество.
- Быстрый доступ и поиск: Благодаря уникальности ключей и автоматической сортировке `std::map` обеспечивает быстрый поиск продуктов по идентификатору, а также позволяет легко проверять наличие продукта в инвентаре.
- Гибкость и масштабируемость: Структура легко расширяется для включения дополнительных данных о продукте или изменения структуры данных продукта (например, замена указателя на продукт на умный указатель для более безопасного управления памятью).

Использование `std::map<int, std::pair<Product*, int>>` демонстрирует, как можно эффективно структурировать и управлять сложными наборами данных в приложениях C++, таких как системы управления инвентарем, где важны скорость доступа к данным и их логическая организация.

3. Чтение и Запись Данных Сотрудников в Файл

Цель задачи: расширить систему управления сотрудниками из Задачи 1, добавив функциональность для чтения и записи данных сотрудников в файл, используя базовые операции работы с файлами.

Основные технологии:

Файловый ввод/вывод: Механизмы C++ для чтения из файлов и записи в файлы, используя потоки `ifstream` и `ofstream`.

Определения:

- **ifstream:** Поток ввода из файла, используется для чтения данных из файла.
- **ofstream:** Поток вывода в файл, используется для записи данных в файл.

Шаги Реализации

Шаг 1: Расширение Класа Сотрудника

Для начала, мы используем класс `Employee` из Задачи 1, добавив методы для сериализации (преобразование в строку для записи в файл) и десериализации (преобразование строки из файла обратно в объект).

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include "locale.h"

class Employee {
public:
    int id;
    std::string name;
    std::string department;

    Employee() : id(0) {} // Для десериализации

    Employee(int id, std::string name, std::string department)
        : id(id), name(name), department(department) {}

    // Сериализация данных сотрудника в строку
    std::string serialize() const {
        return std::to_string(id) + ";" + name + ";" + department;
    }

    // Десериализация строки в данные сотрудника
    void deserialize(const std::string& data) {
        size_t pos1 = data.find(';');
        size_t pos2 = data.find(';', pos1 + 1);
        id = std::stoi(data.substr(0, pos1));
        name = data.substr(pos1 + 1, pos2 - pos1 - 1);
        department = data.substr(pos2 + 1);
    }
};
```

Шаг 2: Запись Списка Сотрудников в Файл

Теперь добавим функцию для записи списка сотрудников в файл. Эта функция будет принимать вектор сотрудников и имя файла, в который будет произведена запись.

```
void writeEmployeesToFile(const std::vector<Employee>& employees, const std::string& filename) {
    std::ofstream fileOut(filename);
    for (const auto& employee : employees) {
        fileOut << employee.serialize() << std::endl;
    }
    fileOut.close();
}
```

```
}

```

Шаг 3: Чтение Списка Сотрудников из Файла

Добавим функцию для чтения списка сотрудников из файла. Функция будет читать данные из файла, десериализовать строки в объекты Employee и добавлять их в вектор.

```
void readEmployeesFromFile(std::vector<Employee>& employees, const std::string& filename) {
    std::ifstream fileIn(filename);
    std::string line;
    while (std::getline(fileIn, line)) {
        Employee emp;
        emp.deserialize(line);
        employees.push_back(emp);
    }
    fileIn.close();
}
```

Шаг 4: Пример Использования

Продemonстрируем использование этих функций в main.

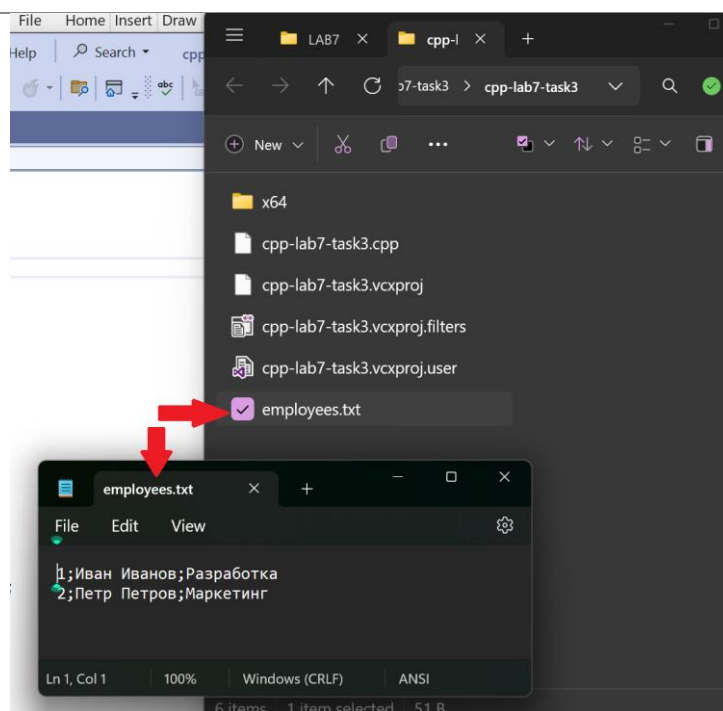
```
int main() {
    setlocale(LC_ALL, "Russian");
    std::vector<Employee> employees;
    employees.push_back(Employee(1, "Иван Иванов", "Разработка"));
    employees.push_back(Employee(2, "Петр Петров", "Маркетинг"));

    // Запись в файл
    writeEmployeesToFile(employees, "employees.txt");

    // Чтение из файла
    std::vector<Employee> loadedEmployees;
    readEmployeesFromFile(loadedEmployees, "employees.txt");

    // Вывод сотрудников, загруженных из файла
    for (const auto& employee : loadedEmployees) {
        std::cout << employee.id << " " << employee.name << " " << employee.department << std::endl;
    }

    return 0;
}
```



Объяснение Кода

- Класс Employee теперь имеет методы `serialize` и `deserialize` для преобразования объекта сотрудника в строку и обратно. Это позволяет нам легко сохранять и загружать данные сотрудников в файл и из файла.
- Функция `writeEmployeesToFile` записывает данные каждого сотрудника в файл, используя сериализованную строку.
- Функция `readEmployeesFromFile` читает строки из файла, десериализует их в объекты Employee и добавляет в вектор.
- В `main` демонстрируется процесс записи списка сотрудников в файл и последующее его чтение, с выводом данных сотрудников на экран.

Эта задача показывает, как можно работать с файловым вводом/выводом в C++, используя сериализацию и десериализацию данных для удобной работы с объектами.

4. Личный Каталог Библиотеки

Цель задачи: создать систему каталога личной библиотеки, которая использует операции ввода/вывода файлов и контейнеры STL для управления коллекцией книг.

Шаги Реализации

Шаг 1: Определение Класа Книги

Сначала определим класс Book, который будет содержать информацию о каждой книге.

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include "locale.h"

class Book {
public:
    std::string title;
    std::string author;
    std::string genre;
    int year;

    Book(std::string title, std::string author, std::string genre, int year)
        : title(std::move(title)), author(std::move(author)), genre(std::move(genre)), year(year) {}

    // Сериализация данных книги в строку
    std::string serialize() const {
        return title + ";" + author + ";" + genre + ";" + std::to_string(year);
    }

    // Десериализация строки в данные книги
    void deserialize(const std::string& data) {
        size_t pos = 0;
        size_t next_pos = data.find(';', pos);
        title = data.substr(pos, next_pos - pos);

        pos = next_pos + 1;
        next_pos = data.find(';', pos);
        author = data.substr(pos, next_pos - pos);

        pos = next_pos + 1;
        next_pos = data.find(';', pos);
        genre = data.substr(pos, next_pos - pos);

        pos = next_pos + 1;
        year = std::stoi(data.substr(pos));
    }
};
```

Шаг 2: Функции Сохранения и Загрузки

Реализуем функции для сохранения и загрузки каталога книг.

```
void saveLibrary(const std::vector<Book>& library, const std::string& filename) {
    std::ofstream fileOut(filename);
    for (const auto& book : library) {
        fileOut << book.serialize() << std::endl;
    }
    fileOut.close();
}

void loadLibrary(std::vector<Book>& library, const std::string& filename) {
    std::ifstream fileIn(filename);
    std::string line;
    while (std::getline(fileIn, line)) {
        Book book("", "", "", 0);
        book.deserialize(line);
        library.push_back(book);
    }
}
```

```

    }
    fileIn.close();
}

```

Шаг 3: Тестирование Функций

Тестируем функции, добавляя книги в каталог и сохраняя/загружая их.

```

int main() {
    setlocale(LC_ALL, "Russian");
    // Создание каталога библиотеки с некоторыми книгами
    std::vector<Book> myLibrary = {
        Book("Война и мир", "Лев Толстой", "Роман", 1869),
        Book("Преступление и наказание", "Федор Достоевский", "Роман", 1866)
    };

    // Сохранение каталога библиотеки в файл
    saveLibrary(myLibrary, "library.txt");

    // Загрузка каталога библиотеки из файла
    std::vector<Book> loadedLibrary;
    loadLibrary(loadedLibrary, "library.txt");

    // Вывод каталога библиотеки, загруженного из файла
    for (const auto& book : loadedLibrary) {
        std::cout << "Название: " << book.title << ", Автор: " << book.author
                    << ", Жанр: " << book.genre << ", Год: " << book.year << std::endl;
    }

    return 0;
}

```

Объяснение Кода

- В классе Book определены атрибуты для названия книги, автора, жанра и года издания. Методы `serialize` и `deserialize` используются для преобразования данных книги в строку и обратно, что позволяет легко сохранять и загружать информацию о книге в файл и из файла.
- Функция `saveLibrary` принимает вектор книг и имя файла, в который будут записаны данные. Каждый объект книги сериализуется в строку и записывается в файл.
- Функция `loadLibrary` открывает файл с именем, переданным в качестве параметра, читает строки и десериализует их обратно в объекты `Book`, которые затем добавляются в вектор.
- В функции `main` происходит создание каталога библиотеки, сохранение его в файл, загрузка каталога из файла и вывод загруженных данных на экран.

Этот пример иллюстрирует, как можно работать с последовательностями объектов, использовать файловый ввод/вывод для сохранения и восстановления состояния программы, и как применять сериализацию для удобной работы с данными.

Самостоятельная задача: Система управления библиотекой

Ваша задача - создать простую систему управления библиотекой на C++. Система должна позволять добавлять новые книги в библиотеку, брать книги в займы, возвращать книги и выводить список всех книг в библиотеке. Также система должна сохранять состояние библиотеки между запусками программы, сохраняя и загружая данные о книгах в файл и из файла. Например, если пользователь попытается взять в займы книгу, которая уже кем-то взята, система должна сообщить ему об этом. Когда книга возвращается, система должна позволить пользователю взять её в займы. Вам следует создать разные классы, например класс для библиотеки и класс для книги, а также использовать вектор.