Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Кафедра Электронных вычислительных машин

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

К ЛАБОРАТОРНЫМ РАБОТАМ

по курсу

«Объектно-ориентированное программирование»

(часть 2)

для студентов факультета ФРЭ, ФКП

Минск – 2018

# **Введение**

Основная цель настоящих методических указаний – дать материал для лабораторных занятий по изучаемым темам дисциплины. В указаниях по каждой изучаемой теме приводится список практических заданий для выполнения.

**Темы лабораторных работ:**

1. Генерация и обработка исключений.
2. Потоки. Состояние потока.
3. Организация работы с файлами. Текстовые файлы.
4. Организация работы с файлами. Бинарные файлы. Файлы произвольного доступа.
5. Класс-контейнеры и класс-итераторы.
6. Класс алгоритма.
7. Классы-контейнеры и классы-итераторы STL.
8. Игра на С++.

# Лабораторная работа №1

**Тема работы**: Генерация и обработка исключений

**Цель работы:** Понять, как обрабатываются исключения. Использовать try, throw и catch для отслеживания, индикации и обработки исключений.

**Теоретические материалы.**

**Исключения.**

*Исключения* – это ошибки, возникающие во время работы программы. Они могут вызваны множеством различных обстоятельств, таких, как выход за пределы памяти, ошибка недопустимым значением или использование индекса, выходящего за пределы вектора.

**Для чего нужны исключения?**

Зачем нам понадобился еще один механизм обработки ошибок? Рассмотрим, как обрабатывались ошибки в прошлом. Программы на С имели привычку сообщать об ошибке возвращением установленного значения из функции, в которой она произошла. Например, функции работы с дисковыми файлами часто возвращают NULL или 0, чтобы показать, что произошла ошибка. Каждый раз при вызове функции происходит проверка возвращаемых значений:

if (somefunc()== ERROR\_RETURN\_VALUE)

//обработка ошибки или вызов обработчика ошибок

else

//нормальная работа

if (anotherfunc()==NULL)

//обработка ошибки или вызов обработчика ошибок

else

//нормальная работа

if (thirdfunc()==0)

//обработка ошибки или вызов обработчика ошибок

else

//нормальная работа

Одна из проблем данного подхода заключается в том, что каждый вызов такой функции должен проверяться программой. Окружение каждой функции if, добавление выражений обработки ошибок или вызова обработчиков приводит к тому, что код разрастается, как на дрожжах, становится запутанным, читается с трудом.

Проблема еще более усугубляется при использовании классов, так как ошибки могут возникать и при отсутствии явных вызовов функций. Например, представим себе, что какое-либо приложение определяет объекты класса:

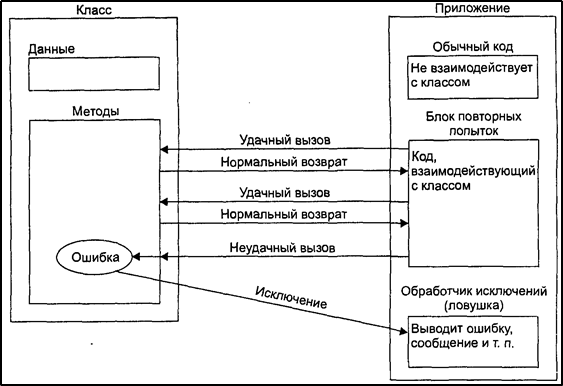
someClass obj1, obj2, obj3;

Как объяснить программе, что произошла ошибка в конструкторе класса? Последний вызывается неявным образом, поэтому никакого возвращаемого значения от него не дождешься.

Еще больше краски сгущаются, когда в приложении используются библиотеки классов. Библиотека классов - это нечто, сделанное посторонними людьми, но используемое вами, программистами, купившими ее, в вашей программе. Поэтому очень сложно отследить ошибки и установить какие-то значения, которые сообщали об ошибках, - реализация библиотечных классов всегда остается неизвестной для программиста. Собственно говоря, проблема выявления ошибок, возникающих в глубинах библиотечных функций, - это, возможно, одна из самых глобальных задач, решаемых с помощью исключений. В конце этого раздела мы обязательно вернемся к этому вопросу.

**Синтаксис исключений.**

Представим себе приложение, создающее объекты какого-либо класса и работающее с ними. В нормальной ситуации вызова методов не приводят ни к каким ошибкам. Но иногда в программе возникает ошибка, которую обнаруживает метод. Он формирует приложение о случившемся. При использовании исключений это событие называется генерацией исключительной ситуации. В приложении при этом имеется отдельная секция кода, в которой содержатся операции по обработке ошибок. Этот код называют *обработчиком исключительных ситуаций* или *улавливающим блоком*. В нем отлавливаются исключения, сгенерированные методами. Любой код приложения, использующий объекты класса, заключается в *блок повторных попыток*. Соответственно, ошибки, возникшие в последнем, будут пойманы улавливающим блоком. Код, который не имеет никакого отношения к классу, не включается в блок повторных попыток. На рисунке 1.1. показана схема, описанная выше.



**Рисунок 1.1.** – Механизм исключений

Механизм исключений использует три новых слова С++: ***catch***, ***throw*** и ***try***. Кроме того, нужно создать новый тип сущности, называемый классом exception. Программа XSYNTAX – это еще не нормальная программа, а только скелет, приводимый для того, чтобы показать синтаксис.

**Листинг 1.1.** Скелет программы XSYNTAX

// xsyntax.cpp

// эта программа не работает!

////////////////////////////////////////////////////////////////

Class AClass //просто класс

{

public:

Class AnError //класс exception

{

};

void Func() //какой-то метод

{

if( /\*условие ошибки\*/)

throw AnError(); //генерировать исключение

}

};

////////////////////////////////////////////////////////////////

int main() //приложение как бы

{

try //блок повторных попыток

{

AClass obj1; //взаимодействие с объектами AClass

Obj1.Func(); //тут может возникнуть ошибка

}

catch(AClass::AnError) //обработчик ошибок

{ //(улавливающий блок)

}

return 0;

}

Мы начинаем работу с создания класса AClass, представляющего собой некий класс, в котором могут возникнуть ошибки. Класс исключений, AnError, определён в общедоступной части класса AClass. Методы последнего подвергаются тщательному обследованию на предмет выявления ошибок. Если таковые обнаруживаются, мы с помощью ключевого слова throw и конструктора класса исключений производим генерацию исключительной ситуации:

throw AnError(); //’throw’ и конструктор класса AnError

В main() мы заключаем все выражения, каким-либо образом относящиеся к AClass, в блок повторных попыток. Если какие-то из них приводят к тому, что возникают подозрения на наличие ошибки в методе AClass, то генерируется исключение, и управление программой переходит к улавливающему блоку, который следует сразу же за блоком повторных попыток.

**Простой пример исключения.**

Наконец, посмотрим на какую-нибудь реально работающую программу, использующую идею исключений.

**Листинг 1.2.** Программа XSTAK

//xstak.cpp

//Демонстрация механизма исключений

#include <iostream>

using namespace std;

cons tint MAX=3; //в стеке максимум 3 целых числа

////////////////////////////////////////////////////////////////

class Stack

{

private:

int st[MAX]; //стек: целочисленный массив

int top; //индекс вершины стека

public:

class Range //класс исключений для Stack

{ //внимание: тело класса пусто

};

//--------------------------------------------------------------

Stack() //конструктор

{

top=-1;

}

//--------------------------------------------------------------

void push(int var)

{

if(top>=MAX-1) //если стек заполнен

throw Range(); //генерировать исключение

st[++top]=var; //внести число в стек

}

//--------------------------------------------------------------

int pop()

{

if(top<0) //если стек пуст

throw Range(); //исключение

return st[top--]; //взять число из стека

}

};

////////////////////////////////////////////////////////////////

int main()

{

Stack s1;

try

{

s1.push(11);

s1.push(22);

s1.push(33);

//s1.push(44) //Опаньки! Стек заполнен

cout<<”1: “<<s1.pop()<<endl;

cout<<”2: “<<s1.pop()<<endl;

cout<<”3: “<<s1.pop()<<endl;

cout<<”4: “<<s1.pop()<<endl;

//Опаньки! Стек пуст

catch(Stack::Range) //обработчик

{

cout<<”Исключение: Стек переполнен или пуст”<<endl;

}

cout<< “Приехали сюда после захвата исключения (или нормального выхода”<<endl;

return 0;

}

Имейте в виду, стек сделан специально таким маленьким, чтобы проще было привести пример возникновения исключительной ситуации, внеся в него слишком много элементов.

Рассмотрим части этой программ, относящиеся к исключениям. Их четыре. В спецификации класса имеется класс исключений. Также там имеются выражения, генерирующие исключения. В main() есть часть кода, которая может привести к исключительным ситуациям, - это блок повторных попыток. Есть код, в котором эти ситуации обрабатываются, - улавливающий блок.

**Описание класса исключений.**

В нашей программе класс исключений описывается внутри класс Stack:

сlass Range

{ //Тело класса пусто!

};

Как видите, тело класса пусто, а потому его объекты не обладают ни данными, ни методами. Все, что нам на самом деле нужно в этой простенькой программе, это имя класса Range. Оно используется для связывания выражения генерации исключения с улавливающим блоком (классу не обязательно быть пустым, как мы увидим позднее).

**Генерация исключения.**

В классе Stack исключение возникает, когда приложение пытается извлечь значение из пустого стека или занести значение в заполненный до отказа стек. Чтобы сообщить приложению о том, что оно выполнило недопустимую операцию с объектом класса Stack, методы этого класса проверяют указанные условия с использованием if и генерируют исключение, если эти условия выполняются. В программе XSTAK исключение генерируется в двух местах, оба раза с помощью выражения throw Range();.

Второе слово этого незамысловатого предложения неявно запускает конструктор класса Range для создания, соответственно, объекта этого класса. Что касается первого слова, то оно передает управление программой обработчику ошибок (мы вскоре его увидим).

**Блок повторных попыток (Try-блок).**

Все выражения в main(), в которых могут произойти ошибки, то есть выражения, манипулирующие данными объектов класса Stack, заключены в фигурные скобки, перед которыми стоит ключевое слово try:

try

{

//код, работающий с объектами, в которых могут произойти ошибки

}

Это просто часть обычного кода программы. Такое могло бы понадобиться вне зависимости от того, используется механизм исключений или нет. В блок повторных попыток включаются только выражения, которым по долгу службы приходится иметь дело с классом Stack. Более того, в программе может быть несколько таких try-блоков, чтобы к этому классу был доступ из разных мест.

**Обработчик ошибок (улавливающий блок).**

Код, в котором содержатся операции по обработке ошибок, заключается в фигурные скобки и начинается с волшебного слова catch. Имя класса исключений должно включать в себя наименование класса, в котором он живёт. В данном примере это Stack::Range.

сatch(Stack::Range)

{

//код-обработчик ошибок

}

Такая конструкция называется обработчиком прерываний. Она должна следовать непосредственно за блоком повторных попыток. Что касается приведенного примера, то в нем обработчик ошибок занимается только выводом на экран сообщения об ошибках, чтобы пользователь знал, почему его программа перестала работать. Управление программой «проваливается» на дно обработчиком ошибок, так что программа может продолжать работу прямо с этого места. Обработчик, правда, может передать управление совсем в другое место, а еще чаще он поступает совсем просто: завершает работу программы.

**Последовательность событий.**

Давайте подведем итоги и покажем последовательность действий программы при возникновении ошибки.

1. Код нормально выполняется вне блока повторных попыток.

2. Управление переходит в блок повторных попыток.

3. Какое-то выражение в этом блоке приводит к возникновению ошибки в методе.

4. Метод генерирует исключение.

5. Управление переходит к обработчику ошибок (улавливающему блоку), следующему сразу за блоком повторных попыток.

Вот и все. Обратите внимание, каким прозрачным и четким получился конечный алгоритм. В любом выражении блока повторных попыток может произойти ошибка, но у нас все под контролем, не нужно беспокоиться о получении возвращаемого значения для каждого из них, потому что это делается автоматически. В данном конкретном примере сознательно написано два ошибочных выражения. Первое:

s1.push(44); //в стек занесено слишком много элементов

приводит к исключительной ситуации (если в примере убрать комментарии перед ним), а второе:

cout<<”4:”<<s1.pop()<<endl; //извлечение из пустого стека

приводит к исключительной ситуации, если закомментировать первое выражение. Попробуйте сделать и так, и так. В обоих случаях выведено одно и то же сообщение:

Исключение: Стек переполнен или пуст

**Варианты заданий:**

1. Лабораторная работа выполняется в Visual Studio 2010-2017 версия.

2. Соблюдение основных принципов ООП и правил хорошего тона (не использовать глобальные переменные и т.д.).

***Общая часть задания:***

Реализовать шаблонный класс операций над динамическими элементами данных, включающий операции: добавление, удаление, поиск, сортировка, просмотр.

***По вариантная часть задания:***

|  |  |
| --- | --- |
| **№ задания** | **Задание к лабораторной работе** |
| **1** | Проверить обработку исключительных ситуаций: выделение памяти, выход за пределы массива, проверка ввода. Реализовать однонаправленный список с использованием сортировки Шелла. |
| **2** | Проверить обработку исключительных ситуаций: освобождение памяти, удаление из пустого множества, проверка ввода. Реализовать двунаправленный список с использованием сортировки пузырьком. |
| **3** | Проверить обработку исключительных ситуаций: выделение памяти, сортировка пустого множества, проверка ввода. Реализовать однонаправленный стек с использованием сортировки вставками. |
| **4** | Проверить обработку исключительных ситуаций: освобождение памяти, просмотр пустого множества, проверка ввода. Реализовать двунаправленный стек с использованием быстрой сортировки. |
| **5** | Проверить обработку исключительных ситуаций: выделение памяти, неудачный поиск, проверка ввода. Реализовать однонаправленное кольцо с использованием сортировки Шелла. |
| **6** | Проверить обработку исключительных ситуаций: освобождение памяти, выход за пределы массива, проверка ввода. Реализовать двунаправленное кольцо с использованием сортировки пузырьком. |
| **7** | Проверить обработку исключительных ситуаций: выделение памяти, удаление из пустого множества, проверка ввода. Реализовать бинарное дерево с использованием сортировки вставками. |
| **8** | Проверить обработку исключительных ситуаций: освобождение памяти, сортировка пустого множества, проверка ввода. Реализовать динамический массив(вектор) с использованием быстрой сортировки. |
| **9** | Проверить обработку исключительных ситуаций: выделение памяти, просмотр пустого множества, проверка ввода. Реализовать однонаправленный список с использованием сортировки Шелла. |
| **10** | Проверить обработку исключительных ситуаций: освобождение памяти, неудачный поиск, проверка ввода. Реализовать двунаправленный список с использованием сортировки пузырьком. |
| **11** | Проверить обработку исключительных ситуаций: выделение памяти, выход за пределы массива, проверка ввода. Реализовать однонаправленный стек с использованием сортировки вставками. |
| **12** | Проверить обработку исключительных ситуаций: освобождение памяти, удаление из пустого множества, проверка ввода. Реализовать двунаправленный стек с использованием быстрой сортировки. |
| **13** | Проверить обработку исключительных ситуаций: выделение памяти, сортировка пустого множества, проверка ввода. Реализовать однонаправленное кольцо с использованием сортировки Шелла. |
| **14** | Проверить обработку исключительных ситуаций: освобождение памяти, просмотр пустого множества, проверка ввода. Реализовать двунаправленное кольцо с использованием сортировки пузырьком. |
| **15** | Проверить обработку исключительных ситуаций: выделение памяти, неудачный поиск, проверка ввода. Реализовать бинарное дерево с использованием сортировки вставками. |
| **16** | Проверить обработку исключительных ситуаций: освобождение памяти, выход за пределы массива, проверка ввода. Реализовать динамический массив(вектор) с использованием быстрой сортировки. |
| **17** | Проверить обработку исключительных ситуаций: выделение памяти, удаление из пустого множества, проверка ввода. Реализовать однонаправленный список с использованием сортировки Шелла. |
| **18** | Проверить обработку исключительных ситуаций: освобождение памяти, сортировка пустого множества, проверка ввода. Реализовать двунаправленный список с использованием сортировки пузырьком. |
| **19** | Проверить обработку исключительных ситуаций: выделение памяти, просмотр пустого множества, проверка ввода. Реализовать однонаправленный стек с использованием сортировки вставками. |
| **20** | Проверить обработку исключительных ситуаций: освобождение памяти, неудачный поиск, проверка ввода. Реализовать двунаправленный стек с использованием быстрой сортировки. |
| **21** | Проверить обработку исключительных ситуаций: выделение памяти, выход за пределы массива, проверка ввода. Реализовать однонаправленное кольцо с использованием сортировки Шелла. |
| **22** | Проверить обработку исключительных ситуаций: освобождение памяти, удаление из пустого множества, проверка ввода. Реализовать двунаправленное кольцо с использованием сортировки пузырьком. |
| **23** | Проверить обработку исключительных ситуаций: выделение памяти, сортировка пустого множества, проверка ввода. Реализовать бинарное дерево с использованием сортировки вставками. |
| **24** | Проверить обработку исключительных ситуаций: освобождение памяти, просмотр пустого множества, проверка ввода. Реализовать динамический массив (вектор) с использованием быстрой сортировки. |
| **25** | Проверить обработку исключительных ситуаций: выделение памяти, освобождение памяти, просмотр пустого множества, проверка ввода. Реализовать однонаправленный список с использованием сортировки Шелла. |

# Лабораторная работа №2

**Тема работы**: Потоки. Состояние потока

**Цель работы:** Познакомиться с потоками. Овладеть навыками работы с потоками ввода/вывода.

**Теоретические материалы.**

**Потоковые классы.**

*Поток* – это общее название, как ни странно, потока данных. В С++ поток представляет собой объект некоторого класса. Именно поэтому вы могли встретить в листингах потоковые объекты cin и cout. Разные потоки предназначены для представления разных видов данных. Например, класс ifstream олицетворяет собой поток данных от входного дискового файла.

**Преимущества потоков.**

Программисты на С могут удивиться, какие же преимущества дает использование потоковых классов для ввода/вывода вместо традиционных функций С printf(), scanf(), для файлов – fprintf(), fscanf().

Одним из аргументов в пользу потоков является простота использования. Если вам приходилось когда-нибудь использовать символ управления форматом %d в printf(), вы оцените это. Ничего подобного в потоках вы не встретите, ибо каждый объект сам знает, как он должен выглядеть на экране. Это избавляет программиста от одного из основных источников ошибок.

Другой причиной является то, что можно перегружать стандартные операторы и функции вставки (**<<**) и извлечения (**>>**) для работы с создаваемыми классами. Это позволяет работать с собственными классами как со стандартными типами, что, опять же, делает программирование проще и избавляет от множества ошибок, не говоря уж об эстетическом удовлетворении.

Но неужели потоковый ввод/вывод так важен при работе в среде с графическим интерфейсом пользователя, такой, как, например, Windows, где непосредственный вывод текста на экран не используется вообще? Неужели в наше время еще нужны потоки С++? Оказывается, да, нужны. Потому что это лучший способ записывать данные в файл, лучший способ организации данных в памяти для последующего использования при вводе/выводе текста в окошках и других элементах графического интерфейса пользователя (GUI).

**Иерархия потоковых классов**

Потоковые классы имеют довольно сложную иерархическую структуру. На рис. 2.1 показана организация важнейших из них.

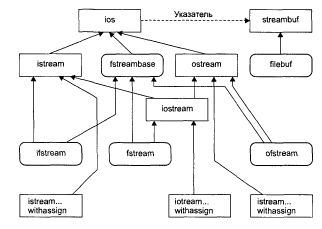


Рис. 2.1 - Иерархия потоковых классов

Мы уже встречались с использованием некоторых потоковых классов. Операция извлечения >> является методом класса istream, операция вставки << - методом класса ostream. Оба этих класса являются наследниками ios. Объект cout, представляющий собой стандартный выходной поток, который обычно выводит на экран данные, является предопределенным объектом класса ostream\_withassign, являющегося наследником класса ostream. Аналогично, cin - объект, istream\_withassign, наследника istream.

Классы, используемые для вывода данных на экран и ввода с клавиатуры, описаны в заголовочном файле IOSTREAM, который мы всегда подключали в наших примерах из предыдущих глав. Классы, используемые для ввода/вывода файлов, объявленны в файле FSTREAM. На рис. 2.1 видно, в каких заголовочных файлах что хранится. (К сказанному можно добавить, что некоторые манипуляторы описаны в IOMANIP, а некоторые классы для работы с объектами в “в памяти” определены в STRSTREAM.) В качестве полезного упражнения можно распечатать эти заголовочные файлы и проследить взаимоотношения различных классов. Файлы хранятся в директории INCLUDE вашего компилятора. При изучении хранящихся в этих файлах классов можно найти ответы на многие вопросы, касающиеся потоков.

На рис. 2.1 видно, что класс ios является базовым для всей иерархии. Он содержит множество констант и методов, общих для операций ввода/вывода любых видов. Некоторые из них, такие, как флаги форматирования showpoint и fixed. В классе ios есть также указатель на класс streambuf, значением которого является адрес текущего буфера памяти. С помощью буфера производится обмен данными (запись или чтение). Кроме того, ios содержит ситуации вам не нужно заботиться о классе streambuf, обращение к которому из других классов происходит автоматически.

Классы istream и ostream являются наследниками ios и предназначены для ввода и вывода соответственно. Класс istream содержит такие функции, как get(), getline(), read() и перегружаемую операцию извлечения (>>), тогда как класс ostream содержит функции put(), write() и перегружаемую операцию вставки (<<).

Класс iostream - наследник одновременно классов iostream и ostream (пример множественного наследования). Его производные классы могут использоваться при работе с такими объектами, как дисковые файлы, которые могут быть открыты одновременно для записи и чтения. Три класса - istream\_withassign, ostream\_withassign и iostream\_withassign - являются наследниками istream, ostream и iostream соответственно. Они добавляют к этим классам операторы присваивания.

Приведенный ниже свод классов может показаться довольно абстрактным. Его можно пока пропустить, чтобы вернуться к нему при осуществлении каких-либо конкретных действия с использованием потоков.

**Класс ios**

Ios является дедушкой всех потоковых классов и обладает большинством особенностей, без которых работа с потоками была бы невозможно. Три его главных направления - это флаги форматирования, флаги ошибок и режим работы с файлами. Сначала мы обратимся к рассмотрению флагов форматирования, затем флагов состояния ошибки. Рассмотрение режима работы с файлами мы отложим до тех пор, пока не познакомимся с дисковыми файлами.

**Флаги форматирования**

Флаги форматирования - это набор определенный enum (перечисляемого типа) в классе ios. Они работают переключателями, определяющими различные форматы и способы ввода/вывода. Мы не будем обсуждать подробно каждый флаг, так как одни мы уже использовали, а названия других говорят сами за себя. В таблице 2.1 приводится полный список флагов форматирования.

Таблица 2.1 - Флаги форматирования класса ios

|  |  |
| --- | --- |
| **Флаги** | **Значение** |
| skipws | Пропуск пробелов при вводе |
| left | Выравнивание по левому краю |
| right | Выравнивание по правому краю |
| internal | Заполнение между знаком или основанием числа и самим числом |
| dec | Перевод в десятичную форму |
| oct | Перевод в восьмеричную форму |
| hex | Перевод в шестнадцатеричную форму |
| boolalpha | Перевод логического 0 и 1 соответственно в “false” и “true” |
| showbase | Выводить индикатор основания системы счисления (0 для восьмеричной, 0х для шестнадцатеричной) |
| showpoint | Показывать десятичную точку при выводе |
| uppercase | Переводить в верхний регистр буквы Х, Е и буквы шестнадцатеричной системы счисления (ABCDEF) (по умолчанию - в нижнем регистре) |
| showpos | Показывать “+” перед положительными целыми числами |
| scientific | Экспоненциальный вывод чисел с плавающей запятой |
| fixed | Фиксированный вывод чисел с плавающей запятой |
| unitbuf | Сброс потоков после вставки |
| stdio | сброс stdout, sterror после вставки |

Есть несколько способов установки флагов форматирования, для каждого свои. Так как они являются компонентами класса ios, обычно к ним обращаются посредством написания имени класса и оператора явного задания (например ios::skipws). Все без исключения флаги могут быть выставлены с помощью методов setf() и unsetf(). Взгляните на следующий пример:

cout.setf (ios::letf); //выравнивание текста по левому краю

cout >> “Этот текст выровнен по левому краю”

cout.unsetf(ios::left); //вернуться к прежнему форматированию

Многие флаги могут быть установлены с помощью манипуляторов (для самостоятельного рассмотрения).

**Варианты заданий:**

1. Лабораторная работа выполняется в Visual Studio 2010-2017 версия.

2. Соблюдение основных принципов ООП и правил хорошего тона (не использовать глобальные переменные и т.д.).

***Общая часть задания:***

Реализовать классы, представляющие из себя определенные сущности, в которых необходимо реализовать перегрузку операторов ввода и вывода, а также обработку ввода. В классе должны быть обязательно реализованы поля (не менее 7), представляющие из себя числовые и текстовые данные.

Использовать динамические данные для полей, в которых это обоснованно.

***По вариантная часть задания:***

|  |  |
| --- | --- |
| **№ задания** | **Задание к лабораторной работе** |
| **1** | Транспортное средство |
| **2** | Хозяйственные товары |
| **3** | Продукты питания |
| **4** | Одежда |
| **5** | Обувь |
| **6** | Бытовая техника |
| **7** | Фильмография |
| **8** | Мебель |
| **9** | Смурфики |
| **10** | Ягоды |
| **11** | Компьютерная периферия |
| **12** | Актёры |
| **13** | Домашние животные |
| **14** | Фрукты |
| **15** | Картинг |
| **16** | Волейбол |
| **17** | Дикие животные |
| **18** | Американский футбол |
| **19** | Часы |
| **20** | Овощи |
| **21** | Строительные материалы |
| **22** | Боулинг |
| **23** | Детские игрушки |
| **24** | Ресторан |
| **25** | Бижутерия |

# Лабораторная работа №3

**Тема работы**: Организация работы с файлами. Текстовые файлы.

**Цель работы:** Научиться создавать, читать, записывать и обновлять файлы. Овладеть обработкой файлов последовательного доступа.

**Теоретические материалы.**

Файлы позволяют пользователю считывать большие объемы данных непосредственно с диска, не вводя их с клавиатуры. Существуют два основных типа файлов: ***текстовые*** и ***двоичные***. В данной лабораторной работе речь пойдет про первый тип файлов.

**Потоковый ввод/вывод дисковых файлов**

Большинству программ требуется сохранять данные на диске и считывать их. Работа с дисковыми файлами и подразумевает наличие еще одного набора специальных классов: ifstream для ввода и ostream для вывода. Класс fstream осуществляет и ввод, и вывод. Объекты этих классов могут быть ассоциированы с дисковыми файлами, а их методы - использоваться для обмена данными с ними.

Вернемся к рис. 2.1. На нем видно, что класс ifstream является наследником класса ifstream, ofstream - класс ostream, fstream - класса iostream. Эти родительские классы, в свою очередь, являются наследниками класса ios. Такая иерархия вполне логична - классы, ориентированные на работу с файлами, могут использовать методы более общих классов. К тому же файловые классы используют принцип множественного наследования, будучи наследниками еще и класса fstreambase. Этот класс содержит объект класса filebuf, являющегося файловым буфером, а также ассоциированные методы, унаследованные от более общего класса streambuf. Обычно программисту не приходится заботиться от этих буферных классах.

Классы ifstream, ofstream и fstream объявлены в файле FSTREAM.

Программисты на С наверняка обратят внимание на то, что подход к дисковому вводу/выводу на С++ оказывается совсем иным. Старые функции языка С, такие, как fread() и fwrite() в С++ конечно, работают, но они не так хорошо вписываются в концепцию объектно-ориентированной среды программирования. Новый подход, предлагаемый С++, гораздо прозрачнее и проще в использовании. Кстати говоря, остерегайтесь случайного перемешивания старых функций С с потоками С++. Они не всегда в хороших отношениях друг с другом, хотя есть возможность заставить их жить дружно.

**Форматированный файловый ввод/вывод**

При форматированном вводе/выводе числа хранятся на диске в виде серии символов. Таким образом, число 6,02 вместо того, чтобы храниться в виде четырехбайтного значения типа float или восьмибайтного double, хранится в виде последовательности символов ‘6’, ‘,’, ‘0’, ‘2’. Это странно с точки зрения экономии памяти, особенно в случае многоразрядных чисел, но зато легко применяется на практике и в некоторых ситуациях гораздо удобнее. Слава Богу, что хоть символы и строки хранятся в файлах в более или менее привычной форме.

**Запись данных**

Следующая программа демонстрирует запись символа, целого числа, числа типа double и двух объектов типа string в дисковый файл. Вывод на экран не производится. Приведем листинг программы FORMATO.

**Листинг 3.1.** Программа FORMATO

//formato.cpp

//Форматированный вывод в файл с использованием <<

#include <stream>

#include <iostream>

#include <string>

using namespace std;

int main()

{

char ch = ‘x’;

int j=77;

double d=6.02;

string str1 = “Kafka”; //строки без

string str2 = “Proust”; //пробелов

ofstream outfile(“fdata.txt”); //создать объект ofstream

outfile<<ch //вставить (записать) данные

<<j

<<’ ‘ //пробелы между числами

<<d

<<str1

<<’ ‘ //пробелы между строками

<<str2;

cout<< “Файл записан\n”;

return 0;

}

Здесь мы определили объект outfile в качестве компонента класса ofstream. В то же время мы инициализировали его файлом FDATA.TXT. Инициализация резервирует для дискового файла с данным именем различные ресурсы и получает доступ (или *открывает файл*) к нему. Если файл не существует, он создается. Если файл уже существует, он переписывается - новые данные в нем заменяют старые. Объект outfile ведет себя подобно cout из предыдущих программ, поэтому можно использовать операцию вставки (<<) для вывода переменных любого из стандартных типов в файл. Это так замечательно работает потому, что оператор вставки перегружен в классе ostream, который является родительским для ofstream.

Когда программа завершается, объект outfile вызывает свой деструктор, который закрывает файл, так что нам не приходится делать это явным образом.

Есть несколько потенциальных проблем с форматированным выводом в дисковые файлы. Во-первых, надо разделять числа (77 и 6,02, например) не числовыми символами. Поскольку они хранятся в виде последовательности символов, а не в виде полей фиксированной длины, это единственный шанс узнать при извлечении, где кончается одно и начинается другое число. Во-вторых, между строками должны быть разделители - по тем же причинам. Это подразумевает, что внутри строки не может быть пробелов. В этом примере для разделения данных мы использовали пробел в обоих случаях. Радует то, что символы не нуждаются в разделителях, хотя они и являются данными фиксированной длины.

Посмотреть на результаты работы программы FORMATO можно, открыв файл FDATA.TXT с помощью программы WORDPAD или команды DOS TYPE.

**Чтение данных**

Прочитать файл, созданный программой FORMATO, можно с использованием объекта ifstream, инициализированного именем файла. Файл автоматически открывается при создании объекта. Затем данные из него можно считать с помощью оператора извлечения (>>).

Приведем листинг программы FORMATI, которая считывает данные из файла FDATA.TXT.

**Листинг 3.2.** Программа FORMATI

//formati.cpp

//форматированное чтение из файла с помощью >>

#include <fstream> //для файлового ввода/вывода

#include <iostream>

#include <string>

using namespace std;

int main()

{

char ch;

int j;

double d;

string str1;

string str2;

ifstream infile(“fdata.txt”); //создать объект ifstream

//извлечь (прочесть) из него данные

infile>>ch>>j>>d>>str1>>str2;

cout<<ch<<endl //вывести данные

<<j<<endl

<<d<<endl

<<str1<<endl

<<str2<<endl;

return 0;

}

Объект ifstream, который мы назвали infile, действует примерно так же, как cin в предыдущих программах. Если мы корректно вставляли данные в файл, то извлечь их не составит никаких проблем. Мы просто сохраняем их в соответствующих переменных и выводим на экран. Результаты работы программы:

x

77

6.02

Kafka

Proust

Разумеется, числа приводятся обратно к своему двоичному представлению, чтобы с ними можно было работать в программе. Так, 77 сохраняется в переменной j типа int, это уже теперь не семерки символьного типа. 6,02 сохраняется в переменной типа double.

**Строки с пробелами**

Подход, продемонстрированный в последнем примере, не позволяет обрабатывать строки с char\*, содержащие пробелы. Для того чтобы эта ситуация изменилась, после каждой строки нужно писать специальный символ-ограничитель и использовать функцию getline() вместо оператора извлечения. Наша следующая маленькая программа OLINE выводит строки с пробелами. Посмотрите, как это сделано.

**Листинг 3.3.** Программа OLINE

//oline.cpp

//файловый вывод строк с пробелами

#unclude <fstream> //для операций

//файлового ввода/вывода

using namespace std;

int main()

{

ofstream outfile(“TEST.TXT”); //создать выходной файл

//отправить текст в файл

outfile<<”Приходи март. Я сызнова служу.\n”;

outfile<<”В несчастливом кружении событий \n”;

outfile<<”изменчивую прелесть нахожу \n”;

outfile<<”в смешеньи незначительных наитий. \n”;

return 0;

}

Когда вы запустите программу, строки стихотворения И. Бродского будут записаны в файл. Каждая строка заканчивается символом разделения строк (‘\n’). Обратите внимание, это строки типа char\*, а не объекты класса string. Многие потоковые операции гораздо легче производить с этим типом данных.

Чтобы извлечь строки из файла, мы создадим ifstream и станем читать строчку за строчкой функцией getline(), которая является методом класса istream. Эта функция считывает все символы, включая разделители, пока не дойдет до специального символа ‘\n’, затем помещает результат чтения в буфер, переданный ей в качестве аргумента. Его содержимое выводится на экран после считывания каждой строки.

**Листинг 3.4.** Программа ILINE

//iline.cpp

//Файловый ввод (извлечение из файла) строк

#include <fstream> //для файловых функций

#include <iostream>

using namespace std;

int main()

{

const int MAX=80; //размер буфера

char buffer[MAX]; //буфер символов

ifstream infile(“TEST.TXT”); //создать входной символ

while (!infile.eof() ) //цикл до EOF

{

infile.getline(buffer, MAX); //читает строку текста

cout<<buffer<<endl; //и выводит ее

}

return 0;

}

В результате работы программы на экране появятся все та же строфа из стихотворения, которую мы записали в файл TEST.TXT в процессе работы программы OLINE. Программе не дано знать, сколько всего строчек в тексте, поэтому она продолжает чтение до тех пор, пока не встретит признак окончания файла (EOF). Учтите, что эту программу нельзя применять для чтения произвольных файлов, - каждая строка текста должна заканчиваться символом ‘\n’. При попытке прочитать файл иной структуры программа зависнет.

**Определение признака конца файла (EOF)**

Итак, объекты порожденных из ios классов содержат флаги статуса ошибок, с помощью которых можно проверить результат выполнения операций. При чтении файла небольшими порциями, как в этом примере, мы рано или поздно наткнемся на условие окончания файла. Сигнал EOF посылается в программу операционной системой, когда больше нет данных для чтения. В программе ILINE это условие встречалось в выражении:

while (!infile.eof() ) //пока не EOF

Это все замечательно, но надо учитывать, что, проверяя конкретный флаг признака окончания файла, мы не проверяем все остальные флаги ошибок. А failbit и badbit тоже могут возникнуть при работе программы, хотя это случается довольно редко. Чтобы проверять все, что можно, мы изменим условие цикла:

while (infile.good) //пока нет ошибок

Можно также проверять поток напрямую. Любой потоковый объект, например infile, имеет значение, которое может тестироваться на предмет выявления наиболее распространенных ошибок, включая EOF. Если какое-либо из условий ошибки имеет значение true, объект возвращает ненулевое значение. Это значение на самом деле является указателем, но возвращаемый “адрес” никакой смысловой нагрузки не несет - он просто должен быть нулем либо не нулем. Поэтому запишем еще один вариант цикла while:

while(infile) //пока нет ошибок

На вид это выражение, конечно, очень простое, но непосвященным, скорее всего, будет непонятно, что оно делает.

**Ввод/вывод символов**

Функции put() и get(), являющиеся методами ostream и istream соответственно, могут быть использованы для ввода и вывода единичных символов. В программе OCHAR строка выводится именно посимвольно.

**Листинг 3.5.** Программа OCHAR

//ochar.cpp

//Посимвольный файловый вывод

#include <fstream> //для файловых функций

#include <iostream>

#include <string>

using namespace std;

int main()

{

string str = “Время - великий учитель, но, увы, ”

“оно убивает своих учеников. Берлиоз”;

ostream outfile(“TEST.TXT”); //Cоздать выходной файл

for(int j=0; j<str.size(); j++) //каждый символ

outfile.put(str[j]); //записывать в файл

cout<< “Файл записан\n”;

return 0;

}

В этой программе объект ofstream создается так же, как в программе OLINE. Длина объекта класса string по имени str находится с помощью метода size(), а символы выводятся в цикле for функцией put(). Афоризм Гектора Берлиоза (композитора XIX века) записывается в файл TEST.TXT. Считываем и выводим этот файл с помощью программы ICHAR.

**Листинг 3.6.** Программа ICHAR

//ichar.cpp

//Посимвольный файловый ввод

#include <fstream>

#include <iostream>

using namespace std;

int main()

{

char ch; //символ для считывания

ifstream infile(“TEST.TXT”) //входной файл

while(infile) //читать до EOF или ошибки

{

infile.get(ch); //считать символ

cout<<ch; //и вывести его

}

cout<<endl;

return 0;

}

В этой программе используется функция get(). Чтение производится до признака окончания файла (или возникновения ошибки). Каждый прочитанный символ выводится с помощью cout, поэтому на экран в результате работы программы будет выведен весь афоризм.

Есть и другой способ читать символы из файла - использовать функцию rdbuf() (она является методом класса ios). Функция возвращает указатель на объект класса streambuf (или filebuf), ассоциированный с потоковым объектом. В этом объекте находится буфер символов, считанных из потока, поэтому можно использовать указатель на него в качестве объекта данных.

**Листинг 3.7.** Программа ICHAR2

//ichar2.cpp

//Файловый ввод символов

#include <fstream> //для файловых функций

#include <iostream>

using namespace std;

int main()

{

ifstream infile(“TEST.TXT”); //создать входной файл

cout<<infile.rdbuf(); //передать его буфер в cout

cout<<endl;

return 0;

}

Результат работы этой программы совпадает с ICHAR. Можно взять с полки пирожок за написание самой короткой программы чтения из файла. Обратите внимание на то, что функция rdbuf() сама знает, что следует прекратить работу при достижении EOF.

**Варианты заданий:**

1. Лабораторная работа выполняется в Visual Studio 2010-2017 версия.

2. Соблюдение основных принципов ООП и правил хорошего тона (не использовать глобальные переменные и т.д.).

***Общая часть задания:***

Реализовать класс текстовых файлов, с помощью которого и осуществлять взаимодействие с текстовыми файлами. В данном классе реализовать операции чтения и записи, а также реализовать класс согласно варианту, который будет являться элементом шаблона реализованного в лабораторной работе №1 с возможностью сортировки по каждому полю заданного класса (сортировка та же, что и в лабораторной работе №1). К тому же реализовать дополнительный функционал сортировки, который будет указан в задании по варианту.

Любую работу с данными реализовывать через обработку исключительных ситуаций и реализовать, где возможно, перегрузку операторов.

***По вариантная часть задания:***

|  |  |
| --- | --- |
| **№ задания** | **Задание к лабораторной работе** |
| **1** | Реализовать класс “Товар”. Поля класса:  ● наименование;  ● количество;  ● цена;  ● дата поступления.  Вывести в алфавитном порядке список товаров, хранящихся более количества дней введенных с клавиатуры, стоимость которых превышает введенное с клавиатуры значение. |
| **2** | Реализовать класс “Студент”. Поля класса:  ● ФИО;  ● номер группы;  ● средний балл;  ● доход на одного члена семьи.  Вывести список, в котором сначала идут студенты с доходом на одного члена семьи менее 2-ух базовых величин, затем остальные в порядке уменьшения среднего балла. |
| **3** | Реализовать класс “Рейс автобуса”. Поля класса:  ● номер рейса;  ● тип автобуса;  ● пункт назначения;  ● время отправления;  ● время прибытия на конечный пункт.  Вывести информацию о рейсах, которыми можно воспользоваться для прибытия в пункт назначения раньше заданного времени. |
| **4** | Реализовать класс “АТС”. Поля класса:  ● дата разговора;  ● код и название города;  ● время разговора;  ● тариф;  ● номер телефона в этом городе;  ● номер телефона абонента.  Вывести информацию позже введенной даты и с продолжительностью не менее введенного значения. |
| **5** | Реализовать класс “Сотрудник”. Поля класса:  ● ФИО;  ● табельный номер;  ● количество проработанных часов за месяц;  ● почасовой тариф.  Вывести в алфавитном порядке список сотрудников, проработавших в месяц менее введенного с клавиатуры количества часов. |
| **6** | Реализовать класс “Участник соревнований”. Поля класса:  ● ФИО;  ● игровой номер;  ● возраст;  ● рост;  ● вес.  Вывести информацию в алфавитном порядке о самой молодой и рослой команде. |
| **7** | Реализовать класс “Книга”. Поля класса:  ● регистрационный номер книги;  ● автор;  ● название;  ● год издания;  ● издательство;  ● количество страниц.  Вывести список книг с фамилиями авторов в алфавитном порядке, изданных после заданного года. |
| **8** | Реализовать класс “Библиотека”. Поля класса:  ● регистрационный номер библиотеки;  ● количество книг в библиотеке;  ● жанры книг, которые есть в библиотеке;  ● количество книг взятых на прочтение;  ● количество книг взятых для чтения в зале;  ● количество посещений в месяц зала.  Вывести список книг по жанрам: находящихся на данный момент в библиотеке, взятых на прочтение и взятых на чтение в зал. |
| **9** | Реализовать класс “Продукция”. Поля класса:  ● наименование;  ● количество;  ● номер цеха.  Для заданного цеха необходимо вывести количество выпущенных изделий по каждому наименованию в порядке убывания количества. |
| **10** | Реализовать класс “Кондитерская”. Поля класса:  ● наименование товаров;  ● количество товара;  ● срок годности товара.  Для заданного срока годности необходимо вывести количество выпущенных изделий по каждому наименованию в порядке убывания количества. |
| **11** | Реализовать класс “Сотрудник”. Поля класса:  ● ФИО;  ● номер отдела;  ● должность;  ● дата начала работы.  Вывести список сотрудников в порядке убывания по номеру отдела. |
| **12** | Реализовать класс “Зарплата”. Поля класса:  ● ФИО;  ● наименование отдела;  ● размер зарплаты.  Вывести в алфавитном порядке сотрудников заданного отдела, у которых зарплата ниже введенного значения. |
| **13** | Реализовать класс “ Изделие”. Поля класса:  ● категория (A, B, C);  ● ФИО сборщика;  ● наименование цеха;  ● количество изделий по категории.  Вывести общее количество и информацию об изделиях определенной категории, собранных рабочими введенного с клавиатуры цеха. |
| **14** | Реализовать класс “Абонент”. Поля класса:  ● ФИО абонента;  ● год установки телефона;  ● номер телефона.  Вывести в алфавитном порядке список абонентов с установленными телефонами с заданного года. |
| **15** | Реализовать класс “Игрушка”. Поля класса:  ● наименование;  ● количество;  ● цена;  ● возрастные границы.  Вывести в алфавитном порядке список игрушек, которые подпадают в определенные возрастные границы. |
| **16** | Реализовать класс “Студент”. Поля класса:  ● ФИО;  ● номер группы;  ● оценки по 5 экзаменам;  ● результаты 5 зачетов.  Вывести в алфавитном порядке список неуспевающих студентов, в заданной группе. |
| **17** | Реализовать класс “Обувь”. Поля класса:  ● артикул;  ● наименование;  ● количество;  ● цена.  Вывести в алфавитном порядке список обуви, цена которого менее введенного значения. |
| **18** | Реализовать класс “Исполнитель”. Поля класса:  ● ФИО;  ● год рождения;  ● название страны;  ● класс музыкального инструмента;  ● занятое место.  Вывести список исполнителей по классам инструментов в порядке занятых мест. |
| **19** | Реализовать класс “Художник”. Поля класса:  ● ФИО;  ● год рождения;  ● название страны;  ● направление в искусстве;  ● количество произведений.  Вывести в алфавитном порядке список художников по направлению в искусстве в порядке уменьшения существующих произведений. |
| **20** | Реализовать класс “Мультфильм”. Поля класса:  ● название мультфильма;  ● год выпуска;  ● название страны;  ● количество частей;  ● рейтинг.  Вывести в алфавитном порядке список мультфильмов по количеству частей в порядке уменьшений рейтинга. |
| **21** | Реализовать класс “Студент”. Поля класса:  ● ФИО;  ● факультет;  ● название изучаемых дисциплин;  ● курс;  ● семестр;   * количество лабораторных работ.   Вывести список студентов по количеству не сданных лабораторных, в порядке увеличения курса и номера семестра по каждой дисциплине. |
| **22** | Реализовать класс “Лекарства”. Поля класса:  ● наименование препарата;  ● страна изготовитель;  ● срок годности;  ● рейтинг продаваемости препарата (от 1 до 10);  ● кол-во проданных препаратов по каждому наименованию.  Вывести препараты рейтинг, которых мал и срок годности подходит к концу. А также проанализировать вероятности продажи этих препаратов, за 1 месяц, до конца срока годности и вывести в % соотношении. |
| **23** | Реализовать класс “Пиццерия”. Поля класса:  ● наименование пиццы;  ● продукты необходимые для каждой пиццы;  ● кол-во каждого из продуктов для каждой отдельной пиццы;  ● кол-во проданных пицц в день;  ● кол-во оставшихся продуктов для пиццы.  Вывести в алфавитном порядке список продуктов, которые испортятся за 1 день, если купят меньше пиц. |
| **24** | Реализовать класс “Преподаватель”. Поля класса:  ● ФИО;  ● кол-во групп;  ● кол-во лабораторных работ по предмету;  ● название предметов;  ● кол-во выданных лабораторных работ.  Вывести в алфавитном порядке название предметов в зависимости от кол-ва не выданных лабораторных работ в каждой группе. |
| **25** | Реализовать класс “Университет”. Поля класса:  ● название университете;  ● кол-во факультетов;  ● кол-во выпускников;  ● ФИО преподавателей.  Вывести в алфавитном порядке список преподавателей, которые вели у выпускников по каждому университету, и отобразить в процентном соотношении загруженность преподавателей на каждом факультете (чем больше студентов, тем больше загруженность) . |

# Лабораторная работа №4

**Тема работы**: Организация работы с файлами. Бинарные файлы. Файлы произвольного доступа.

**Цель работы:**  Поработать с бинарными файлами.

**Теоретические материалы.**

**Двоичный ввод/вывод.**

Форматированный файловый ввод/вывод чисел целесообразно использовать только при их небольшой величине и малом количестве. В противном случае, конечно, гораздо эффективнее использовать двоичный ввод/вывод, при котором числа хранятся таким же образом, как в ОП компьютера, а не в виде символьных строк. Целочисленные значения занимают 4 байта, тогда как текстовая версия числа, например “12345”, занимает 5 байт. Значения типа float также всегда занимают 4 байта. А форматированная версия “6,02314е13” занимает 10 байт.

В следующем примере показано, как в бинарном виде массив целых чисел записывается в файл и читается из него. При этом используются две функции - write() (метод класса ofstream), а также read() (метод ifstream). Эти функции думают о данных в терминах байтов (тип char). Им все равно, как организованы данные, что они собой представляют, - они просто переносят байты из буфера в файл и обратно. Параметрами этих функций являются адрес буфера и его длина. Адрес должен быть вычислен с использованием reinterpret\_cast относительно типа char\*. Вторым параметром является длина в байтах (а не число элементов данных в буфере).

**Листинг программы 4.1.** Программа BINIO

//binio.cpp

//Двоичный ввод/вывод целочисленных данных

#include <fstream> //для файловых потоков

#include <iostream>

using namespace std;

const int MAX=100; //размер буфера

int buff[MAX]; //буфер для целых чисел

int main()

{

for(int j=0; j<MAX; j++) //заполнить буфер данными

buff[j]=j; //(0,1,2, ...)

//создать выходной поток

ofstream os(“edata.dat”, ios::binary);

//записать в него

os.write(reinterpret\_cast<char\*>(buff), MAX\*sizeof(int));

os.close(); //должен закрыть его

for(j=0; j<MAX; j++) //стереть буфер

buff[j]=0;

//создать входной поток

ifstream is (“edata.dat”, ios::binary);

//читать из него

is.read(reinterpret\_cast<char\*>(buff), MAX\*sizeof(int));

for(j=0; j<MAX; j++) //проверка данных

if(buff !=j)

{

cerr<<“Некорректные данные!\n”;

return 1;

}

cout<<”Данные корректны\n”;

return 0;

}

При работе с бинарными данными в качестве второго параметра write() и read() следует использовать ios::binary. Это необходимо по той причине, что текстовый режим, используемый по умолчанию, допускает несколько вольное обращение с данными. Например, специальный символ ‘\n’ занимает два байта (на самом деле это и есть два действия - перевод каретки и перевод строки). Это делает текст более удобным для чтения в DOS утилитами типа TYPE, но для бинарных данных такой подход не годится вовсе, так как любой байт, которому не повезло иметь ASCII-код 10, переводится двумя байтами. Аргумент ios::binary - типичный пример *бита состояния*.

**Оператор reinterpret\_cast**

Программа BINIO использует оператор reinterpret\_cast для того, чтобы буфер данных типа int выглядел для функций write() и read() как буфер типа char.

is.read(reinterpret\_cast<char\*>(buff), MAX\*sizeof(int));

reinterpret\_cast как бы говорит компилятору: “Я знаю, что тебе это не понравится, и все-таки я это сделаю”. Он изменяет тип данных в определенной области памяти, совершенно не задумываясь о том, имеет это смысл или нет. Поэтому вопрос целесообразности использования этого оператора остается целиком на совести программиста.

Можно использовать reinterpret\_cast для превращения указателей в данные типа int и обратно. Это небезопасное занятие, но порой оно необходимо.

**Закрытие файлов**

До сих пор не нужно было вручную закрывать файлы - это делалось автоматически при окончании работы с ними. Соответствующие объекты запускали деструкторы и закрывали ассоциированные файлы. Но в программе BINIO оба потока, входной is и выходной os, связаны с одним и тем же файлом EDATA.DAT, поэтому первый поток должен быть закрыт до того, как откроется второй. Для этого используется функция close().

Можно запускать эту функцию каждый раз при закрытии файла, не полагаясь на деструктор потока.

**Варианты заданий:**

1. Лабораторная работа выполняется в Visual Studio 2010-2017 версия.

2. Соблюдение основных принципов ООП и правил хорошего тона (не использовать глобальные переменные и т.д.).

***Общая часть задания:***

Реализовать класс бинарных файлов, с помощью которого и осуществлять взаимодействие с файлами. Данная лабораторная работа является модификацией лабораторной работы №3. Задание остаётся с лабораторной работы №3, но есть и некоторые изменения:

1. Работу с файлами осуществлять только через бинарные файлы.
2. Сортировку реализовать через файлы (в оперативной памяти в один момент времени может находится не более 2 объектов заданного класса (по варианту)), то есть считали 2 объекта из файла, сравнили их и изменение записали в файл, а после считывание другие 2 объекта.
3. Просмотр объектов реализовать поэлементно, то есть считывание объект из файла, выводить на экран, считываете следующий и т.д.
4. Новые/временные файлы не создавать, то есть взаимодействовать только с одним файлом.
5. Реализовать возможность просмотра/изменения конкретного объекта.
6. Обоснованное использование функций произвольного доступа.

# Лабораторная работа №5

**Тема работы**: Класс-контейнеры и класс-итераторы

**Цель работы:** Познакомиться с классами-итераторами и классами-контейнерами.

**Теоретические материалы.**

Контейнеры:

<http://ru.cppreference.com/w/cpp/container>

**Варианты заданий:**

1. Лабораторная работа выполняется в Visual Studio 2010-2017 версия.

2. Соблюдение основных принципов ООП и правил хорошего тона (не использовать глобальные переменные и т.д.).

***Общая часть задания:***

Реализовать класс-контейнер и класс-итератор согласно варианту.

Перемещение по контейнеру и изменение элементов контейнера осуществлять только через итераторы. Для итераторов предусмотреть возможность доступа к произвольному элементу контейнера. Элементом контейнера является класс, заданный по варианту в 3 лабораторной работе. Реализовать сортировку контейнера согласно варианту. Продемонстрировать работу заданных итераторов.

Любую работу с данными реализовывать через обработку исключительных ситуаций и реализовать, где возможно, перегрузку операторов для контейнера и итератора.

Для ассоциативных контейнеров либо ключ, либо значение должны быть уникальными, так что на свое усмотрение выбирайте уникальное поле.

***По вариантная часть задания:***

|  |  |
| --- | --- |
| **№ задания** | **Задание к лабораторной работе** |
| **1** | **Контейнер**: ассоциативный контейнер (ключ - объект, значение - число).  **Итераторы**: вывода и однонаправленный.  **Сортировка**: сортировка вставками (сортировка по ключу). |
| **2** | **Контейнер**: однонаправленный список.  **Итераторы**: ввода и однонаправленный.  **Сортировка**: сортировка Шелла. |
| **3** | **Контейнер**: двунаправленный список.  **Итераторы**: вывода и двунаправленный.  **Сортировка**: сортировка пузырьком. |
| **4** | **Контейнер**: однонаправленный стек.  **Итераторы**: ввода и обратный.  **Сортировка**: сортировка вставками. |
| **5** | **Контейнер**: двунаправленный стек.  **Итераторы**: вывода и двунаправленный.  **Сортировка**: быстрая сортировка. |
| **6** | **Контейнер**: однонаправленное кольцо.  **Итераторы**: ввода и однонаправленный.  **Сортировка**: сортировка Шелла. |
| **7** | **Контейнер**: двунаправленное кольцо.  **Итераторы**: вывода и двунаправленный.  **Сортировка**: сортировка пузырьком. |
| **8** | **Контейнер**: бинарное дерево.  **Итераторы**: ввода и однонаправленный.  **Сортировка**: сортировка вставками. |
| **9** | **Контейнер**: динамический массив (вектор).  **Итераторы**: вывода и обратный.  **Сортировка**: быстрая сортировка. |
| **10** | **Контейнер**: ассоциативный контейнер (ключ - число, значение - объект).  **Итераторы**: ввода и однонаправленный.  **Сортировка**: сортировка Шелла (сортировка по ключу). |
| **11** | **Контейнер**: однонаправленный список.  **Итераторы**: ввода и обратный.  **Сортировка**: сортировка Шелла. |
| **12** | **Контейнер**: двунаправленный список.  **Итераторы**: вывода и двунаправленный.  **Сортировка**: сортировка пузырьком. |
| **13** | **Контейнер**: однонаправленный стек.  **Итераторы**: ввода и однонаправленный.  **Сортировка**: сортировка вставками. |
| **14** | **Контейнер**: двунаправленный стек.  **Итераторы**: вывода и двунаправленный.  **Сортировка**: быстрая сортировка. |
| **15** | **Контейнер**: однонаправленное кольцо.  **Итераторы**: ввода и обратный.  **Сортировка**: сортировка Шелла. |
| **16** | **Контейнер**: двунаправленное кольцо.  **Итераторы**: вывода и двунаправленный.  **Сортировка**: сортировка пузырьком. |
| **17** | **Контейнер**: бинарное дерево.  **Итераторы**: ввода и однонаправленный.  **Сортировка**: сортировка вставками. |
| **18** | **Контейнер**: динамический массив (вектор).  **Итераторы**: вывода и однонаправленный.  **Сортировка**: быстрая сортировка. |
| **19** | **Контейнер**: ассоциативный контейнер (ключ - строка, значение - объект).  **Итераторы**: вывода и обратный.  **Сортировка**: сортировка пузырьком (сортировка по значению). |
| **20** | **Контейнер**: однонаправленный список.  **Итераторы**: ввода и однонаправленный.  **Сортировка**: сортировка Шелла. |
| **21** | **Контейнер**: двунаправленный список.  **Итераторы**: вывода и двунаправленный.  **Сортировка**: сортировка пузырьком. |
| **22** | **Контейнер**: однонаправленный стек.  **Итераторы**: ввода и обратный.  **Сортировка**: сортировка вставками. |
| **23** | **Контейнер**: двунаправленный стек.  **Итераторы**: вывода и обратный.  **Сортировка**: быстрая сортировка. |
| **24** | **Контейнер**: однонаправленное кольцо.  **Итераторы**: ввода и однонаправленный.  **Сортировка**: сортировка Шелла. |
| **25** | **Контейнер**: двунаправленное кольцо.  **Итераторы**: вывода и двунаправленный.  **Сортировка**: сортировка пузырьком. |

# Лабораторная работа №6

**Тема работы**: Класс-алгоритм

**Цель работы:** Познакомиться с классом-алгоритм, поработать с ним.

**Варианты заданий:**

1. Лабораторная работа выполняется в Visual Studio 2010-2017 версия.

2. Соблюдение основных принципов ООП и правил хорошего тона (не использовать глобальные переменные и т.д.).

***Общая часть задания:***

Реализовать класс алгоритма согласно варианту для контейнера и итератора реализованных в 5 лабораторной работе. Все алгоритмы должны быть реализованы собственноручно.

Перемещение по контейнеру и изменение элементов контейнера осуществлять только через итераторы. Для итераторов предусмотреть возможность доступа к произвольному элементу контейнера. Элементом контейнера является класс, заданный по варианту в 3 лабораторной работе.

Любую работу с данными реализовывать через обработку исключительных ситуаций и реализовать, где возможно, перегрузку операторов для контейнера и итератора.

Для ассоциативных контейнеров либо ключ, либо значение должны быть уникальными, так что на свое усмотрение выбирайте уникальное поле.

Алгоритмы в качестве параметров принимают итераторы на начало и конец диапазона (остальные параметры по необходимости).

Для реализации условия необходимо реализовать несколько функций, которые будут отвечать за свое условие и одну из функций передавать в качестве параметра в соответствующие реализации алгоритмов.

***По вариантная часть задания:***

|  |  |
| --- | --- |
| **№ задания** | **Задание к лабораторной работе** |
| **1** | **Контейнер**: ассоциативный контейнер (ключ - объект, значение - число).  **Итераторы**: вывода и однонаправленный.  **Алгоритмы**:  1. сортировка вставками (сортировка по ключу).  2. функция для проверки все ли элементы удовлетворяют определенному условию.  3. функция, которая копирует множество элементов, удовлетворяющих условию, в заданное место. |
| **2** | **Контейнер**: однонаправленный список.  **Итераторы**: ввода и однонаправленный.  **Алгоритмы**:  1. сортировка Шелла.  2. функция для подсчета количества элементов удовлетворяющих определенному условию.  3. функция, которая перемещает множество элементов, удовлетворяющих условию, в заданное место. |
| **3** | **Контейнер**: двунаправленный список.  **Итераторы**: вывода и двунаправленный.  **Алгоритмы**:  1. сортировка пузырьком.  2. функция для сравнения коллекций, которая возвращает итератор на первой различающийся позиции.  3. функция, которая присваивает заданное значение элементам, которые удовлетворяют условию. |
| **4** | **Контейнер**: однонаправленный стек.  **Итераторы**: ввода и обратный.  **Алгоритмы**:  1. сортировка вставками.  2. функция нахождения первого элемента, который удовлетворяет заданным критериям.  3. функция, которая удаляет элементы, удовлетворяющие условию. |
| **5** | **Контейнер**: двунаправленный стек.  **Итераторы**: вывода и двунаправленный.  **Алгоритмы**:  1. быстрая сортировка.  2. функция, которая определяет первую позицию вхождения последовательности.  3. функция, которая удаляет дубликаты. |
| **6** | **Контейнер**: однонаправленное кольцо.  **Итераторы**: ввода и однонаправленный.  **Алгоритмы**:  1. сортировка Шелла.  2. функция для проверки все ли элементы удовлетворяют определенному условию.  3. функция, которая копирует множество элементов, удовлетворяющих условию, в заданное место. |
| **7** | **Контейнер**: двунаправленное кольцо.  **Итераторы**: вывода и двунаправленный.  **Алгоритмы**:  1. сортировка пузырьком.  2. функция для подсчета количества элементов удовлетворяющих определенному условию.  3. функция, которая перемещает множество элементов, удовлетворяющих условию, в заданное место. |
| **8** | **Контейнер**: бинарное дерево.  **Итераторы**: ввода и однонаправленный.  **Алгоритмы**:  1. сортировка вставками.  2. функция для сравнения коллекций, которая возвращает итератор на первой различающийся позиции.  3. функция, которая присваивает заданное значение элементам, которые удовлетворяют условию. |
| **9** | **Контейнер**: динамический массив(вектор).  **Итераторы**: вывода и обратный.  **Алгоритмы**:  1. быстрая сортировка.  2. функция нахождения первого элемента, который удовлетворяет заданным критериям.  3. функция, которая удаляет элементы, удовлетворяющие условию. |
| **10** | **Контейнер**: ассоциативный контейнер (ключ - число, значение - объект).  **Итераторы**: ввода и однонаправленный.  **Алгоритмы**:  1. сортировка Шелла (сортировка по ключу).  2. функция, которая определяет первую позицию вхождения последовательности.  3. функция, которая удаляет дубликаты. |
| **11** | **Контейнер**: однонаправленный список.  **Итераторы**: ввода и обратный.  **Алгоритмы**:  1. сортировка Шелла.  2. функция для проверки все ли элементы удовлетворяют определенному условию.  3. функция, которая копирует множество элементов, удовлетворяющих условию, в заданное место. |
| **12** | **Контейнер**: двунаправленный список.  **Итераторы**: вывода и двунаправленный.  **Алгоритмы**:  1. сортировка пузырьком.  2. функция для подсчета количества элементов удовлетворяющих определенному условию.  3. функция, которая перемещает множество элементов, удовлетворяющих условию, в заданное место. |
| **13** | **Контейнер**: однонаправленный стек.  **Итераторы**: ввода и однонаправленный.  **Алгоритмы**:  1. сортировка вставками.  2. функция для сравнения коллекций, которая возвращает итератор на первой различающийся позиции.  3. функция, которая присваивает заданное значение элементам, которые удовлетворяют условию. |
| **14** | **Контейнер**: двунаправленный стек.  **Итераторы**: вывода и двунаправленный.  **Алгоритмы**:  1. быстрая сортировка.  2. функция нахождения первого элемента, который удовлетворяет заданным критериям.  3. функция, которая удаляет элементы, удовлетворяющие условию. |
| **15** | **Контейнер**: однонаправленное кольцо.  **Итераторы**: ввода и обратный.  **Алгоритмы**:  1. сортировка Шелла.  2. функция, которая определяет первую позицию вхождения последовательности.  3. функция, которая удаляет дубликаты. |
| **16** | **Контейнер**: двунаправленное кольцо.  **Итераторы**: вывода и двунаправленный.  **Алгоритмы**:  1. сортировка пузырьком.  2. функция для проверки все ли элементы удовлетворяют определенному условию.  3. функция, которая копирует множество элементов, удовлетворяющих условию, в заданное место. |
| **17** | **Контейнер**: бинарное дерево.  **Итераторы**: ввода и однонаправленный.  **Алгоритмы**:  1. сортировка вставками.  2. функция для подсчета количества элементов удовлетворяющих определенному условию.  3. функция, которая перемещает множество элементов, удовлетворяющих условию, в заданное место. |
| **18** | **Контейнер**: динамический массив(вектор).  **Итераторы**: вывода и однонаправленный.  **Алгоритмы**:  1. быстрая сортировка.  2. функция для сравнения коллекций, которая возвращает итератор на первой различающийся позиции.  3. функция, которая присваивает заданное значение элементам, которые удовлетворяют условию. |
| **19** | **Контейнер**: ассоциативный контейнер (ключ - строка, значение - объект).  **Итераторы**: вывода и обратный.  **Алгоритмы**:  1. сортировка пузырьком (сортировка по значению).  2. функция нахождения первого элемента, который удовлетворяет заданным критериям.  3. функция, которая удаляет элементы, удовлетворяющие условию. |
| **20** | **Контейнер**: однонаправленный список.  **Итераторы**: ввода и однонаправленный.  **Алгоритмы**:  1. сортировка Шелла.  2. функция, которая определяет первую позицию вхождения последовательности.  3. функция, которая удаляет дубликаты. |
| **21** | **Контейнер**: двунаправленный список.  **Итераторы**: вывода и двунаправленный.  **Алгоритмы**:  1. сортировка пузырьком.  2. функция для проверки все ли элементы удовлетворяют определенному условию.  3. функция, которая копирует множество элементов, удовлетворяющих условию, в заданное место. |
| **22** | **Контейнер**: однонаправленный стек.  **Итераторы**: ввода и обратный.  **Алгоритмы**:  1. сортировка вставками.  2. функция для подсчета количества элементов удовлетворяющих определенному условию.  3. функция, которая перемещает множество элементов, удовлетворяющих условию, в заданное место. |
| **23** | **Контейнер**: двунаправленный стек.  **Итераторы**: вывода и обратный.  **Алгоритмы**:  1. быстрая сортировка.  2. функция для сравнения коллекций, которая возвращает итератор на первой различающийся позиции.  3. функция, которая присваивает заданное значение элементам, которые удовлетворяют условию. |
| **24** | **Контейнер**: однонаправленное кольцо.  **Итераторы**: ввода и однонаправленный.  **Алгоритмы**:  1. сортировка Шелла.  2. функция нахождения первого элемента, который удовлетворяет заданным критериям.  3. функция, которая удаляет элементы, удовлетворяющие условию. |
| **25** | **Контейнер**: двунаправленное кольцо.  **Итераторы**: вывода и двунаправленный.  **Алгоритмы**:  1. сортировка пузырьком.  2. функция, которая определяет первую позицию вхождения последовательности.  3. функция, которая удаляет дубликаты. |

# Лабораторная работа №7

**Тема работы**: Класс-контейнеры и класс-итераторы STL

**Цель работы:** Познакомиться с классом-контейнером и классом-итератором STL

**Теоретические материалы.**

**Введение в STL.**

В STL содержится несколько основных сущностей. Три наиболее важные из них - контейнеры, алгоритмы и итераторы.

*Контейнер* - это способ организации хранения данных. В предыдущих лабораторных работах нам уже встречались некоторые контейнеры, такие, как стек, связный список, очередь. Еще один контейнер - это массив, но он настолько тривиален и популярен, что встроен в С++ и большинство других языков программирования. Контейнеры бывают самые разнообразные, и в STL включены наиболее полезные из них. Контейнеры STL подключаются к программе с помощью шаблонных классов, а значит, можно легко изменить тип хранимых в них данных.

Под *алгоритмами* в STL подразумевают процедуры, применяемые к контейнерам для обработки их данных различными способами. Например, есть алгоритмы сортировки, копирования, поиска и объединения. Алгоритмы представлены в STL в виде шаблонных функций. Однако они не являются методами классов-контейнеров. Наоборот, это совершенно независимые функции. На самом деле, одной из самых привлекательных черт STL является универсальность ее алгоритмов. Их можно использовать не только в объектах классов-контейнеров, но и в обычных массивах и даже в собственных контейнерах. (Контейнеры, тем не менее, содержат методы для выполнения некоторых специфических задач.)

*Итераторы* - это обобщение концепции указателей: они ссылаются на элементы контейнера. Их можно инкрементировать, как обычные указатели, и они будут ссылаться последовательно на все элементы контейнера. Итераторы - ключевая часто всего STL, поскольку они связывают алгоритмы с контейнерами. Их можно представить себе в виде кабеля, связывающего колонки вашей стереосистемы или компьютер с его периферией.

**Варианты заданий:**

1. Лабораторная работа выполняется в Visual Studio 2010-2017 версия.

2. Соблюдение основных принципов ООП и правил хорошего тона (не использовать глобальные переменные и т.д.).

***Общая часть задания:***

В данной лабораторной работе функционал остается с 6 лабораторной работы, но в качестве контейнеров и итераторов необходимо использовать контейнеры и итераторы библиотеки STL. Реализация алгоритмов остается с 6 лабораторной работы, но их надо соответствующе подстроить под контейнеры и итераторы библиотеки STL. Итератор использовать стандартный для контейнера по варианту.

***По вариантная часть задания:***

|  |  |
| --- | --- |
| **№ задания** | **Задание к лабораторной работе** |
| **1** | **Контейнер**: map (ключ - объект, значение - число).  **Алгоритмы**:  1. сортировка вставками (сортировка по ключу).  2. функция для проверки все ли элементы удовлетворяют определенному условию.  3. функция, которая копирует множество элементов, удовлетворяющих условию, в заданное место. |
| **2** | **Контейнер**: vector.  **Алгоритмы**:  1. сортировка Шелла.  2. функция для подсчета количества элементов удовлетворяющих определенному условию.  3. функция, которая перемещает множество элементов, удовлетворяющих условию, в заданное место. |
| **3** | **Контейнер**: list.  **Алгоритмы**:  1. сортировка пузырьком.  2. функция для сравнения коллекций, которая возвращает итератор на первой различающийся позиции.  3. функция, которая присваивает заданное значение элементам, которые удовлетворяют условию. |
| **4** | **Контейнер**: deque.  **Алгоритмы**:  1. сортировка вставками.  2. функция нахождения первого элемента, который удовлетворяет заданным критериям.  3. функция, которая удаляет элементы, удовлетворяющие условию. |
| **5** | **Контейнер**: stack.  **Алгоритмы**:  1. быстрая сортировка.  2. функция, которая определяет первую позицию вхождения последовательности.  3. функция, которая удаляет дубликаты. |
| **6** | **Контейнер**: queue.  **Алгоритмы**:  1. сортировка Шелла.  2. функция для проверки все ли элементы удовлетворяют определенному условию.  3. функция, которая копирует множество элементов, удовлетворяющих условию, в заданное место. |
| **7** | **Контейнер**: vector.  **Алгоритмы**:  1. сортировка пузырьком.  2. функция для подсчета количества элементов удовлетворяющих определенному условию.  3. функция, которая перемещает множество элементов, удовлетворяющих условию, в заданное место. |
| **8** | **Контейнер**: list.  **Алгоритмы**:  1. сортировка вставками.  2. функция для сравнения коллекций, которая возвращает итератор на первой различающийся позиции.  3. функция, которая присваивает заданное значение элементам, которые удовлетворяют условию. |
| **9** | **Контейнер**: deque.  **Алгоритмы**:  1. быстрая сортировка.  2. функция нахождения первого элемента, который удовлетворяет заданным критериям.  3. функция, которая удаляет элементы, удовлетворяющие условию. |
| **10** | **Контейнер**: set(ключ - число, значение - объект).  **Алгоритмы**:  1. сортировка Шелла (сортировка по ключу).  2. функция, которая определяет первую позицию вхождения последовательности.  3. функция, которая удаляет дубликаты. |
| **11** | **Контейнер**: stack.  **Алгоритмы**:  1. сортировка Шелла.  2. функция для проверки все ли элементы удовлетворяют определенному условию.  3. функция, которая копирует множество элементов, удовлетворяющих условию, в заданное место. |
| **12** | **Контейнер**: queue.  **Алгоритмы**:  1. сортировка пузырьком.  2. функция для подсчета количества элементов удовлетворяющих определенному условию.  3. функция, которая перемещает множество элементов, удовлетворяющих условию, в заданное место. |
| **13** | **Контейнер**: vector.  **Алгоритмы**:  1. сортировка вставками.  2. функция для сравнения коллекций, которая возвращает итератор на первой различающийся позиции.  3. функция, которая присваивает заданное значение элементам, которые удовлетворяют условию. |
| **14** | **Контейнер**: list.  **Алгоритмы**:  1. быстрая сортировка.  2. функция нахождения первого элемента, который удовлетворяет заданным критериям.  3. функция, которая удаляет элементы, удовлетворяющие условию. |
| **15** | **Контейнер**: deque.  **Алгоритмы**:  1. сортировка Шелла.  2. функция, которая определяет первую позицию вхождения последовательности.  3. функция, которая удаляет дубликаты. |
| **16** | **Контейнер**: stack.  **Алгоритмы**:  1. сортировка пузырьком.  2. функция для проверки все ли элементы удовлетворяют определенному условию.  3. функция, которая копирует множество элементов, удовлетворяющих условию, в заданное место. |
| **17** | **Контейнер**: queue.  **Алгоритмы**:  1. сортировка вставками.  2. функция для подсчета количества элементов удовлетворяющих определенному условию.  3. функция, которая перемещает множество элементов, удовлетворяющих условию, в заданное место. |
| **18** | **Контейнер**: vector.  **Алгоритмы**:  1. быстрая сортировка.  2. функция для сравнения коллекций, которая возвращает итератор на первой различающийся позиции.  3. функция, которая присваивает заданное значение элементам, которые удовлетворяют условию. |
| **19** | **Контейнер**: map(ключ - строка, значение - объект).  **Алгоритмы**:  1. сортировка пузырьком (сортировка по значению).  2. функция нахождения первого элемента, который удовлетворяет заданным критериям.  3. функция, которая удаляет элементы, удовлетворяющие условию. |
| **20** | **Контейнер**: list.  **Алгоритмы**:  1. сортировка Шелла.  2. функция, которая определяет первую позицию вхождения последовательности.  3. функция, которая удаляет дубликаты. |
| **21** | **Контейнер**: deque.  **Алгоритмы**:  1. сортировка пузырьком.  2. функция для проверки все ли элементы удовлетворяют определенному условию.  3. функция, которая копирует множество элементов, удовлетворяющих условию, в заданное место. |
| **22** | **Контейнер**: stack.  **Алгоритмы**:  1. сортировка вставками.  2. функция для подсчета количества элементов удовлетворяющих определенному условию.  3. функция, которая перемещает множество элементов, удовлетворяющих условию, в заданное место. |
| **23** | **Контейнер**: queue.  **Алгоритмы**:  1. быстрая сортировка.  2. функция для сравнения коллекций, которая возвращает итератор на первой различающийся позиции.  3. функция, которая присваивает заданное значение элементам, которые удовлетворяют условию. |
| **24** | **Контейнер**: vector.  **Алгоритмы**:  1. сортировка Шелла.  2. функция нахождения первого элемента, который удовлетворяет заданным критериям.  3. функция, которая удаляет элементы, удовлетворяющие условию. |
| **25** | **Контейнер**: list.  **Алгоритмы**:  1. сортировка пузырьком.  2. функция, которая определяет первую позицию вхождения последовательности.  3. функция, которая удаляет дубликаты. |

# **Лабораторная работа №8**

**Тема работы**: Игра

**Цель работы:** Реализовать игру на языке С++. Закрепить полученные знания по дисциплине ООП.

**Варианты заданий:**

1. Лабораторная работа выполняется в любой студии.

2. Соблюдение основных принципов ООП и правил хорошего тона (не использовать глобальные переменные и т.д.).

***Общая часть задания:***

1. Реализоваться игру на языке С++, согласно выбранному варианту.
2. Реализовать красивый и удобный интерфейс.
3. Реализовать от 3 и более уровней сложности.
4. Не превращать разработку программного приложения в использование набора существующих библиотек!!!