МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ

Методические указания к выполнению лабораторных работ по курсу Технология программирования

ГУАП

Санкт-Петербург

2019

Составители: К.Н. Рождественская, К.А. Курицын

Рецензент –

Представлены методические указания по выполнению лабораторных работ, инструкции по применению программных продуктов Microsoft Visual Studio. Приведен список лабораторных работ с заданиями, пояснениями и комментариями по их выполнению.

Издание предназначено для студентов специалитета и бакалавриата по программе «Технология программирования» направления «Информатика и вычислительная техника», а также смежным специальностям, изучающим вопросы объектно-ориентированного программирования.

ВВЕДЕНИЕ

Язык C++ является универсальным языком программирования, который опирается на опыт языка C. Язык C++ позволяет решить любую задачу программирования, но в силу разнородности задач и сред применения он применяется когда-то чаще, а когда-то – реже.

Программа на языке C++ - это набор взаимосвязанных инструкций, которые пишутся в специальном текстовом редакторе. При запуске программы компьютер будет последовательно выполнять описанные инструкции и алгоритм, который программист описал в своей программе. Чтобы компьютер мог выполнить программу, написанную на C++, существует компилятор, который переводит текстовые инструкции в машинные инструкции. Иными словами, компилятор читает файл с текстом программы, анализирует ее, проверяет на ошибки, и если он их не обнаруживает, создает исполняемый файл, т.е. файл который можно запускать многократно. Скомпилировав программу один раз, ее можно запускать не один раз с различными данными.

Самый распространенный компилятор – это Visual C++. Этот компилятор является средой разработки, который включает в себя текстовый редактор, компилятор, отладчик и еще ряд дополнительных программ и библиотек для разработки.

В данном методическом пособии мы расскажем, как создавать проект и работать в среде Microsoft Visual Studio при программировании на языке C++. Описание выполнения лабораторных работ позволит освоить студентам базовые знания о процессе программирования на языке C++.

В данном методическом пособии для компактности и наглядности изложения материала иногда реализация методов будет описана в самих классах, а не представлена отдельно.

1. СРЕДА РАЗРАБОТКИ MICROSOFT VISUAL STUDIO C++

Microsoft Visual Studio (Visual C++) является названием для библиотек и средств разработки на языке ассемблера, C++, C и пр., входящих в состав Visual Studio в Windows. С помощью Visual C++ можно разработать что угодно — от простых консольных приложений до сложных графических приложений для операционных систем Windows.

1.1. Создание консольного приложения в Visual C++

В этом разделе представлено пошаговое создание проекта в Visual C++.

Перед началом работы над проектом, создается отдельная рабочая директория, в которой будут храниться все данные вашей программы.

Процесс создания проекта состоит из шагов, описанных ниже:

Запустить Microsoft Visual Studio. После требуется создать новый проект, для этого выбираем в меню File → New → New Project. В результате на экране возникнет окно, предлагающее выбрать тип создаваемого проекта (Рисунок 1).

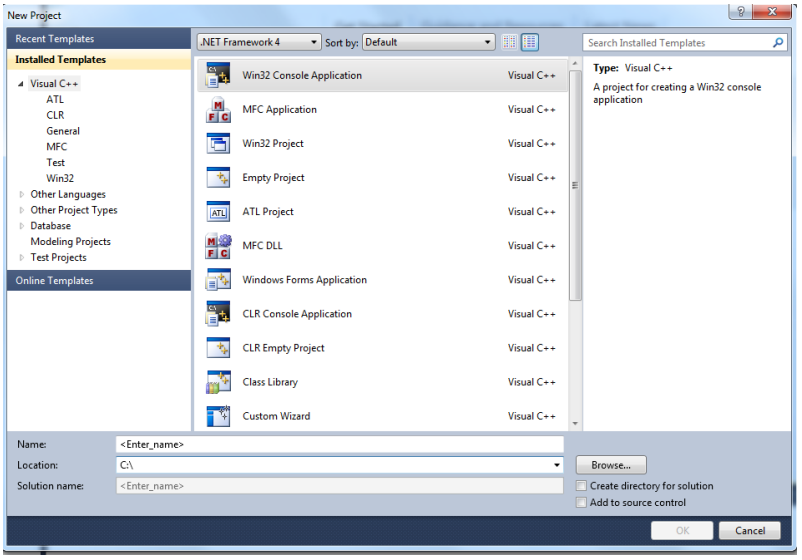
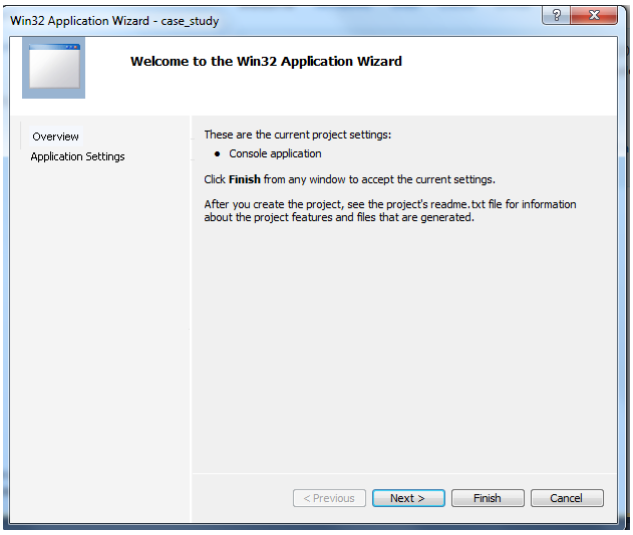


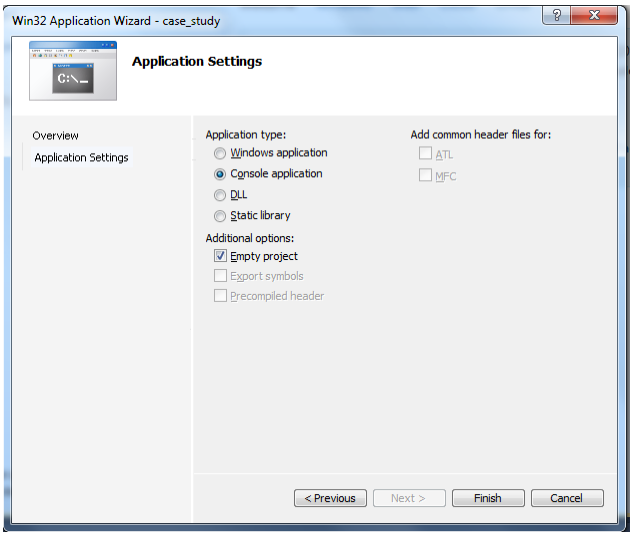
Рисунок 1 – Диалоговое окно «New Project» с выбором типа нового проекта

При выполнении лабораторных работ требуется создание консольного приложения, для этого следует выбрать «Win32 Console Application». В поле «Name» указывается наименование проекта, в поле «Location» - созданная рабочая директория для проекта.

После заполнения требуемых полей нажимаем кнопку «OK».

1. Следующее диалоговое окно (Рисунок 2, А) является информативным, которое сообщает пользователю о выбранном типе проекта, следует выбрать кнопку «Next».





А) Б)

Рисунок 2 – А) первое диалоговое окно, Б) второе диалоговое окно

Во втором диалоговом окне (Рисунок 2, Б), которое появится вслед за предыдущим, пользователю предлагается выбрать настройки проекта. Следует убрать выделение пункта «Precompiled header» и выбрать «Empty project». Далее следует нажать кнопку «Finish».

2.В результате перед нами возникнет текстовый редактор, приглашающий начать писать проект (Рисунок 3).

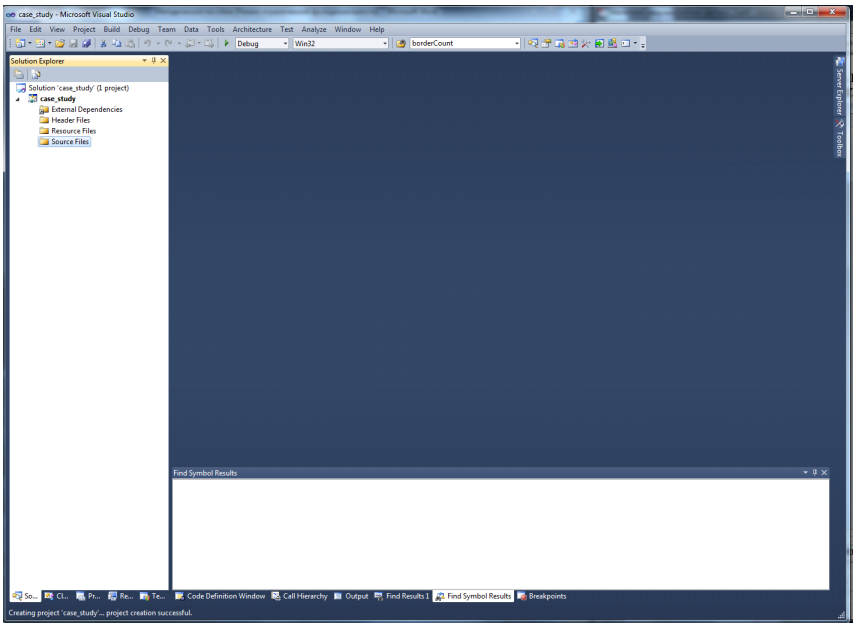


Рисунок 3 – Диалоговое окно текстового редактора созданного проекта

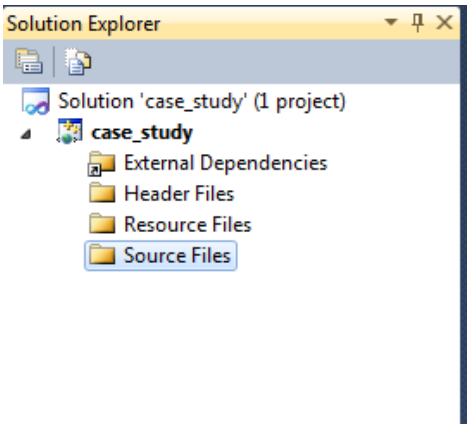


Рисунок 4 – Обозреватель проекта

Слева в разделе «Solution Explorer» находится перечисление папок и файлов (Рисунок 4), которые подключены к проекту. Сейчас он пуст. Требуется добавить файл, в котором будем описывать проект.

1.2. Добавление файлов в проект

При создании ООП программы следует разделять объявление класса и реализацию.

Объявление класса – это описание класса, включая переменные и методы. Но, в объявлении указываются только прототипы, реализация методов внутри класса не описывается. Объявление класса описывается в h-файлах и располагается в папке «Header Files» в разделе «Solution Explorer» (Рисунок 4). h-файлы называются заголовочными файлами.

Для добавления h-файла в проект необходимо сделать следующие шаги:

а) Щелкнуть правой кнопкой мыши на папке «Header Files», в открывшемся меню выбрать «Add», затем «New Item…». Последовательность действий представлена на Рисунке 5.

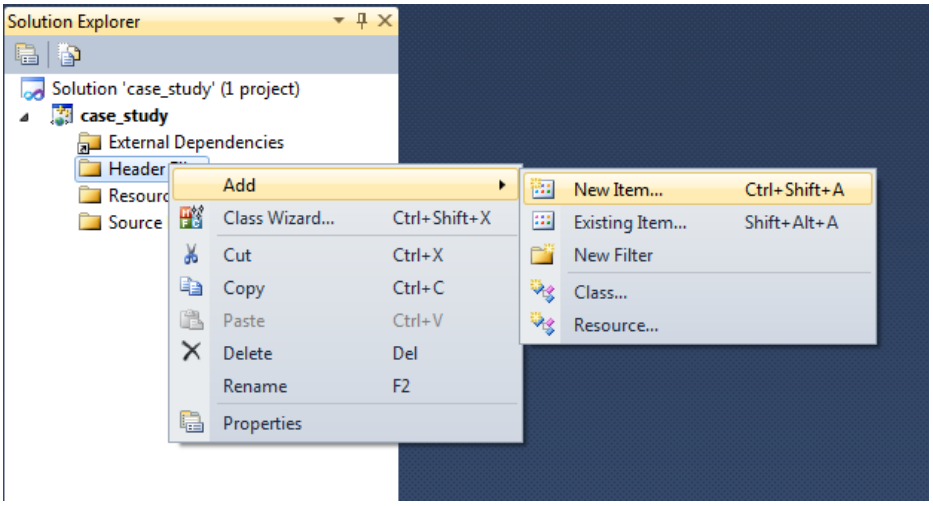


Рисунок 5 – Добавление h-файла в проект

b) В открывшемся диалоговом окне «Add New Item» необходимо выбрать «Header File (.h)» В поле «Name» необходимо написать название для файла. Название для h-файла дается, как правило, по названию класса, который будет в нем описан. Диалоговое окно представлено на Рисунке 6.

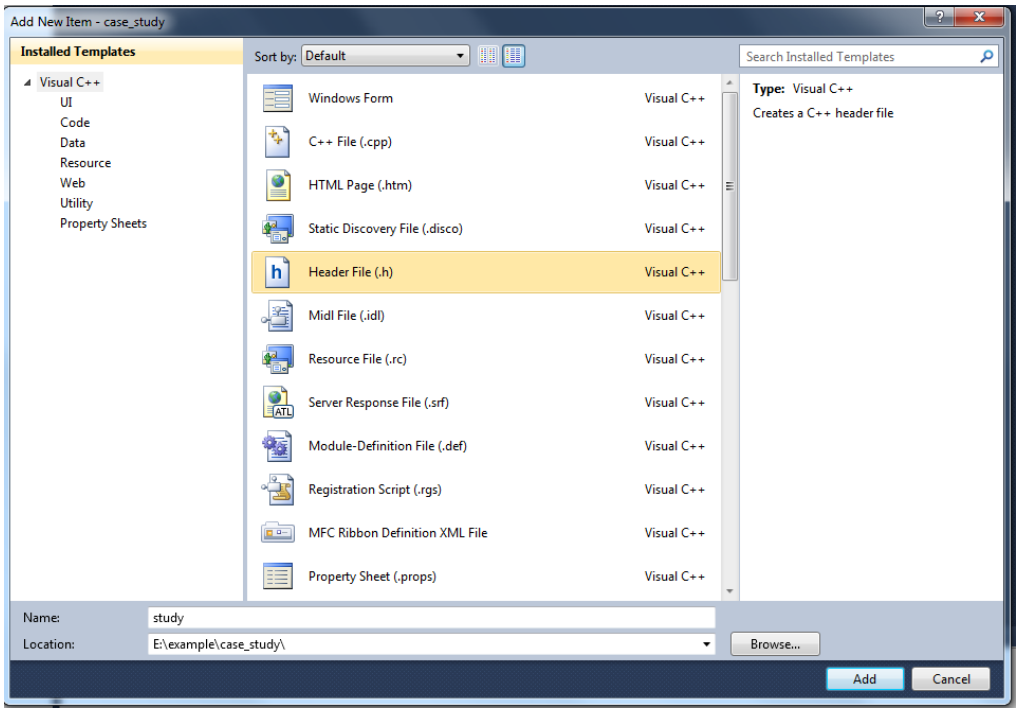


Рисунок 6 – Создание h-файла для проекта

В поле «Location» среда разработки сама укажет путь, где будет располагаться h-файл.

Для завершения работы с данным диалоговым окном выбираем кнопку «Add».

c) В результате проделанных действий будет создан и добавлен h-файл в созданный проект (Рисунок 7). Файл готов к работе.

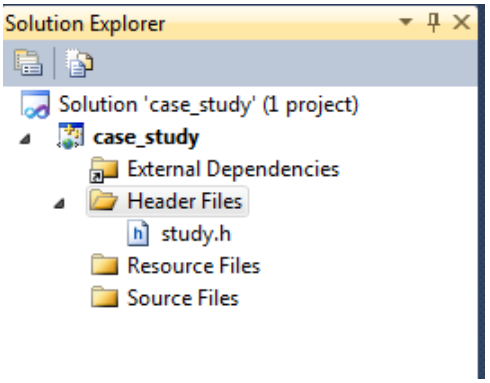


Рисунок 7 – h-файл в «Solution Explorer»

d) В созданном файле в поле текстового редактора описывается класс. Пример приведен на Рисунке 8.

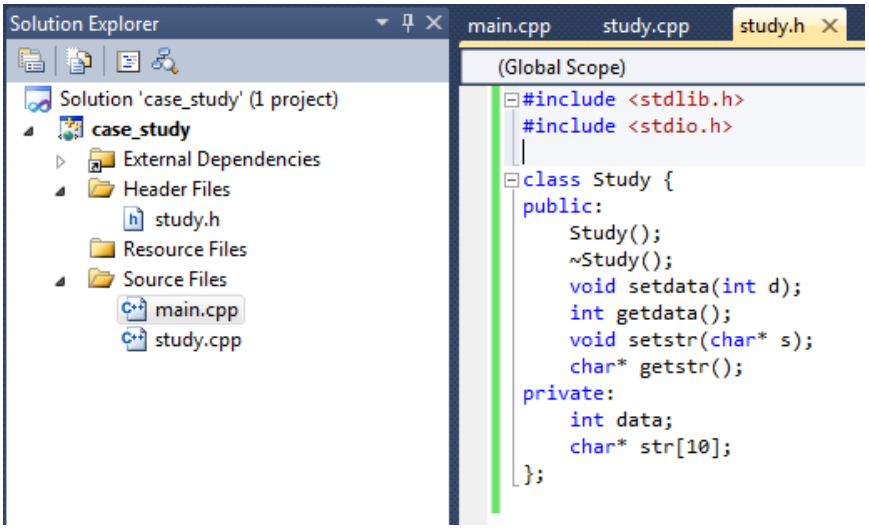


Рисунок 8 – Пример описания класса в h-файле

Реализация описанных методов класса описывается в cpp-файле, имя которого также определяется наименованием класса. Cpp-файл называется файлом исходного кода.

Чтобы добавить в проект cpp-файл необходимо выполнить последовательность шагов, описанную ниже.

a) Щелкнуть правой кнопкой мыши на папке «Header Files», в открывшемся меню выбрать «Add», затем «New Item…». Последовательность действий приведена на Рисунке 9.

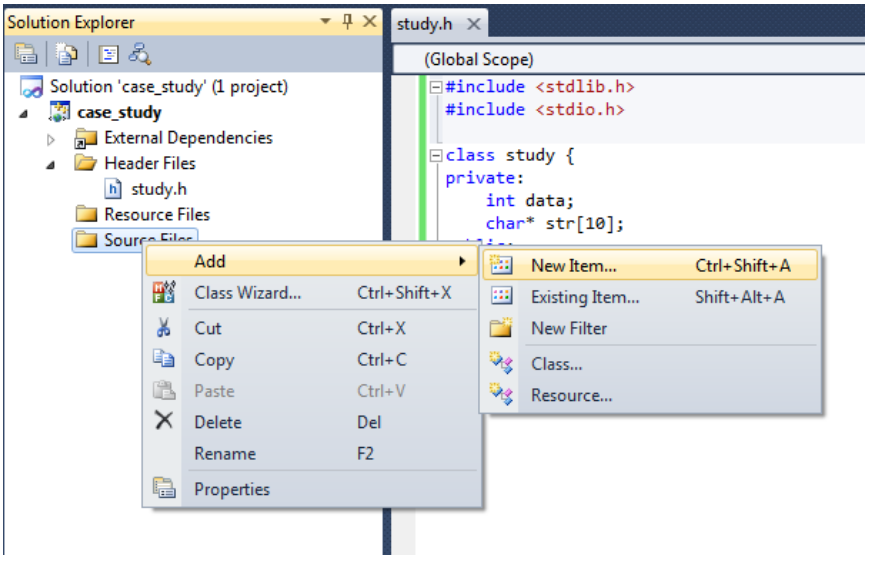


Рисунок 9 – Добавление cpp-файла в проект

b) В открывшемся диалоговом окне «Add New Item» (Рисунок 10) необходимо выбрать «C++ File (.cpp)» В поле «Name» необходимо написать название для файла. Название для cpp-файла дается, как правило, по названию класса, который будет в нем описан.

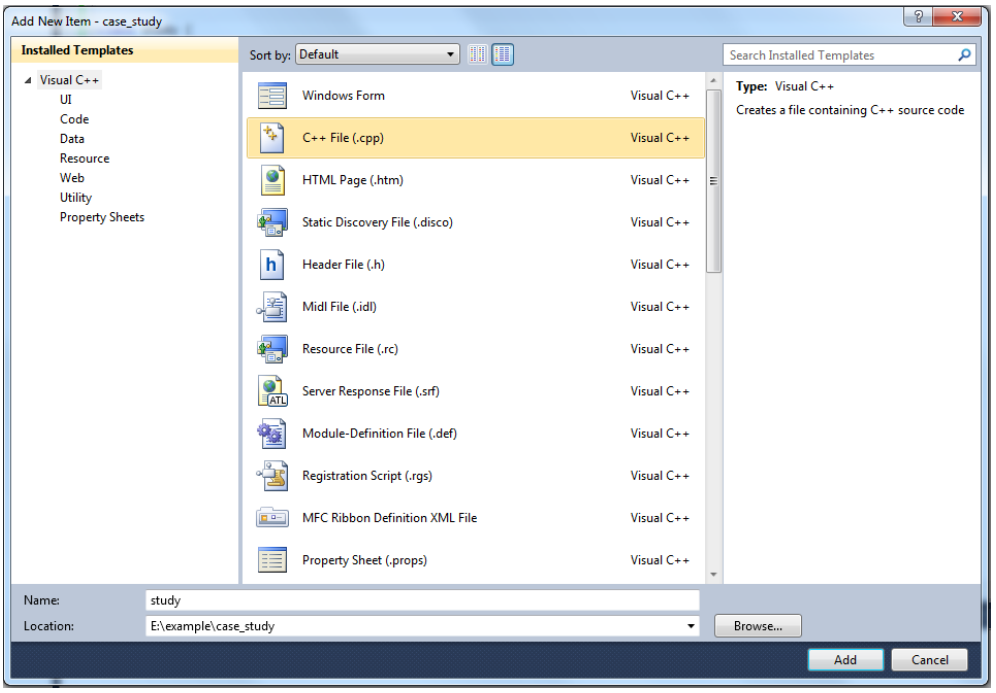


Рисунок 10 – Создание cpp-файла для проекта

В поле «Location» среда разработки сама укажет путь, где будет располагаться cpp-файл.

Для завершения работы с данным диалоговым окном выбираем кнопку «Add».

c) В результате проделанных действий будет создан и добавлен cpp-файл в созданный проект (Рисунок 11). Файл готов к работе.

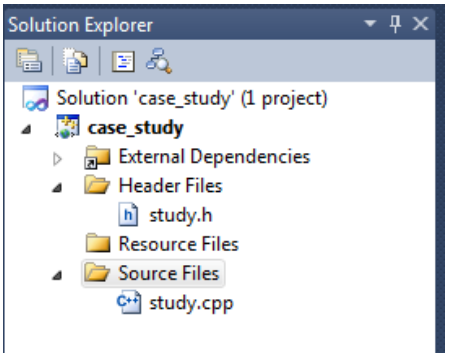


Рисунок 11 – cpp-файл в «Solution Explorer»

d) В созданном файле в поле текстового редактора пишется реализация для соответствующего класса. Пример приведен на Рисунке 12.

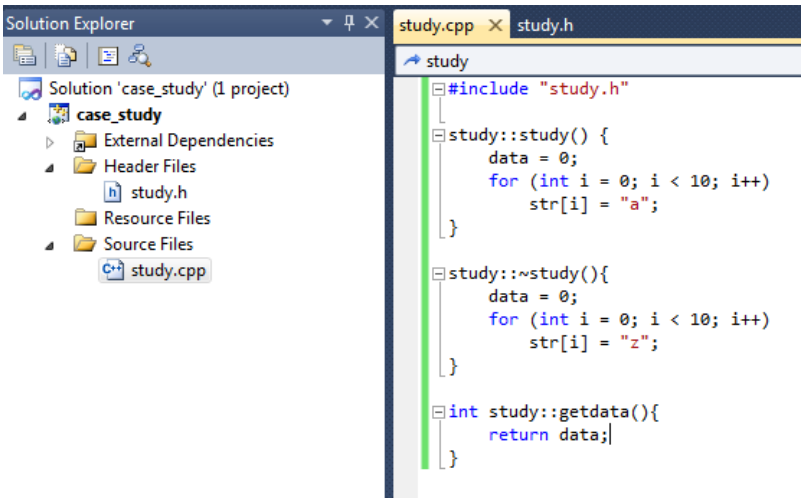


Рисунок 12 - Пример реализации класса в cpp-файле

Таким образом, мы добавили описание и реализацию класса Study в проект, создание объектов и работа с ними осуществляется из функции main(). Чтобы создать эту функцию, необходимо добавить еще один cpp-файл в проект и назвать его main. Это дает возможность разработчику быстро найти точку запуска программы и внести изменения, если это необходимо.

Если придерживаться правил создания файлов проекта (имя файла однозначно определяет к чему он относится: к некоторому классу или main), то пользователю вашей программы или разработчику не потребуется много времени, чтобы определить, что и в каком файле у вас находится. Это в разы упрощает работу с проектом.

1.3. Стиль программирования

Существуют основные правила хорошего стиля программирования, их число достаточно велико, ниже приведены основные пункты:

а) Отступы

Отступы необходимо использовать и вертикальные и горизонтальные. Отступы отображают структуру кода.

Например:

```
1.    study::study() {
2.       data = 0;
3.       for (int i = 0; i < 10; i++)
4.           str[i] = "a";
5.    }
```

Отступы на строках 2 и 3 обозначают последовательное выполнение этих строк. Отступ строк 2 и 3 больше, чем на строках 1 и 5, они находятся как бы «внутри». То же определение относится к строке 4 относительно оператора for.

Каждая инструкция находится на отдельной строчке.

b) Наименование переменных

Все наименования следует записывать по-английски, а не транслитом.

Имена переменных должны быть присвоены согласно их функциональному назначению, не должно быть двойного смысла. Имена переменных пишутся в смешанном регистре, начиная с маленькой буквы, например: `point`, `pointToPoint`.

c) Название методов и функций

Совпадает с правилом для переменных, но отличие между ними состоит в том, что они должны быть глаголами, например: `getValue()`, `setValue(int val)`.

d) Имена типов шаблонов

Имена типов шаблонов следует называть одной заглавной буквой, например: `template<class T>`, `template<class C, class D>`. Позволяет выделить имена шаблонов среди других используемых имён.

e) Разделы класса public, protected, private

Разделы класса public, protected, private должны быть отсортированы. То есть в классе последовательно представлен каждый тип раздела. Все разделы должны быть явно указаны.

Например:

```
class Example {
private:
  . . .
protected:
  . . .
public:
  . . .
};
```

f) Члены класса области private

Наиболее важное свойство такой переменной - это область видимости, поэтому ее следует выделять, чтобы избавиться от путаницы между членами класса и переменными. Пример такой переменной: `int length_`.

g) Методы set/get

Везде, где происходит прямое обращение к переменной необходимо использовать слова set (установить) или get (получить). Например: `employee.getName(); employee.setName(name)`. Это общая практика разработчиков.

h) Исходный код представляется двумя частями: заголовочными файлами (h-файлы) и файлами исходного кода (cpp-файлы). Имена файлов совпадают с именем класса.

i) Защита от вложенного исключения

Заголовочные файлы должны содержать защиту от вложенного включения. Конструкция позволяет избегать ошибок компиляции.

Например:

#ifndef CLASSNAME_H

#define CLASSNAME_H

:

#endif CLASSNAME_H

j) Оператор goto

Не следует использовать оператор goto, так как этот оператор нарушает принципы структурного программирования.

1.4. Отладка программы и точки останова

Бывает, что код скомпилировался, но работает не так как задумал разработчик, или работа программы прекращается в процессе ее эксплуатации из-за ошибки. Если обнаружить ошибку просмотром кода не удается, то стоит пройти программу пошагово, проследить состояние переменных и ход выполнения программы.

Например, у нас есть следующий код:

```
#include <stdlib.h>
#include <stdio.h>
#include "study.h"
int main (){
 int toStudy = 10;
 char* strFromStudy;
 Study study;
 study.setdata(toStudy);
 strFromStudy = study.getstr();
}
```

1.4.1. Пошаговый проход программы

Для запуска пошагового прохода вашей программы нажмите F10 («Debug», «Step Over»). Желтая стрелка будет указывать на строку, которая будет выполнена следующей. При запуске пошагового прохода программы стрелка будет указывать на строку int main(), так как именно она является начальной точкой запуска вашей программы. Пример запуска пошагового прохода программы приведен на Рисунке 13

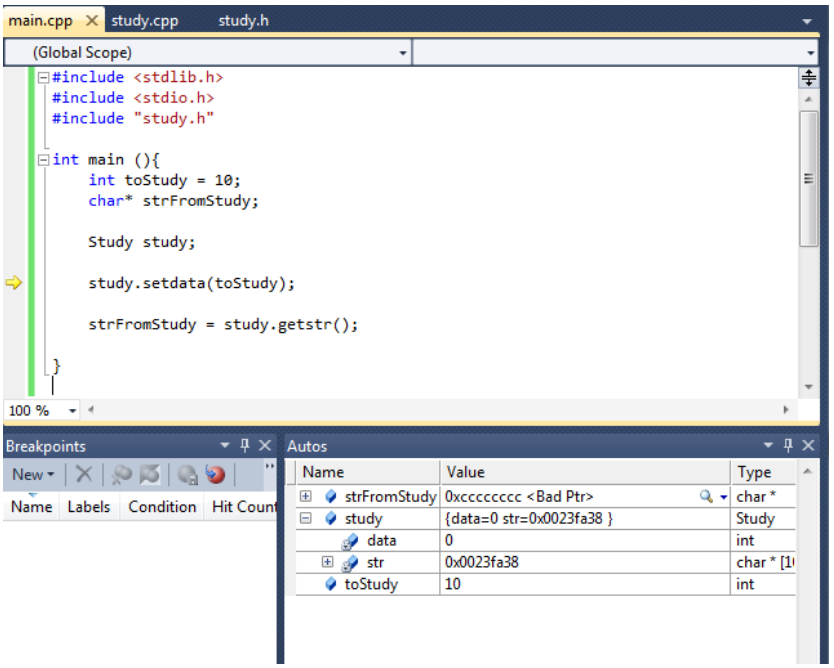


Рисунок 13 – Пошаговый проход программы

Внизу в разделе Watch (или Autos) будет отображаться: наименование переменной, текущее значение и ее тип. По ходу исполнения программы переменные будут изменять свои значения согласно алгоритму, заложенному программистом, что позволит отследить работу программы при пошаговом проходе.

Существует три возможности пошагового прохода программы: «Шаг с обходом» («Step Over», F10), «Шаг с заходом» («Step Into», F11) и «Шаг с выходом» («Step Out», Shift+F11).

- «Шаг с обходом» («Step Over», F10) выполняет оператор, но не заходит в него. Например, строка `study.setdata(toStudy);` будет выполнена, но внутрь метода мы не попадем.
- «Шаг с заходом» («Step Into», F11) позволяет попасть внутрь метода, например при выполнении шага с заходом для строки `study.setdata(toStudy);` отладчик перейдет к коду, изображенному на Рисунке 14.

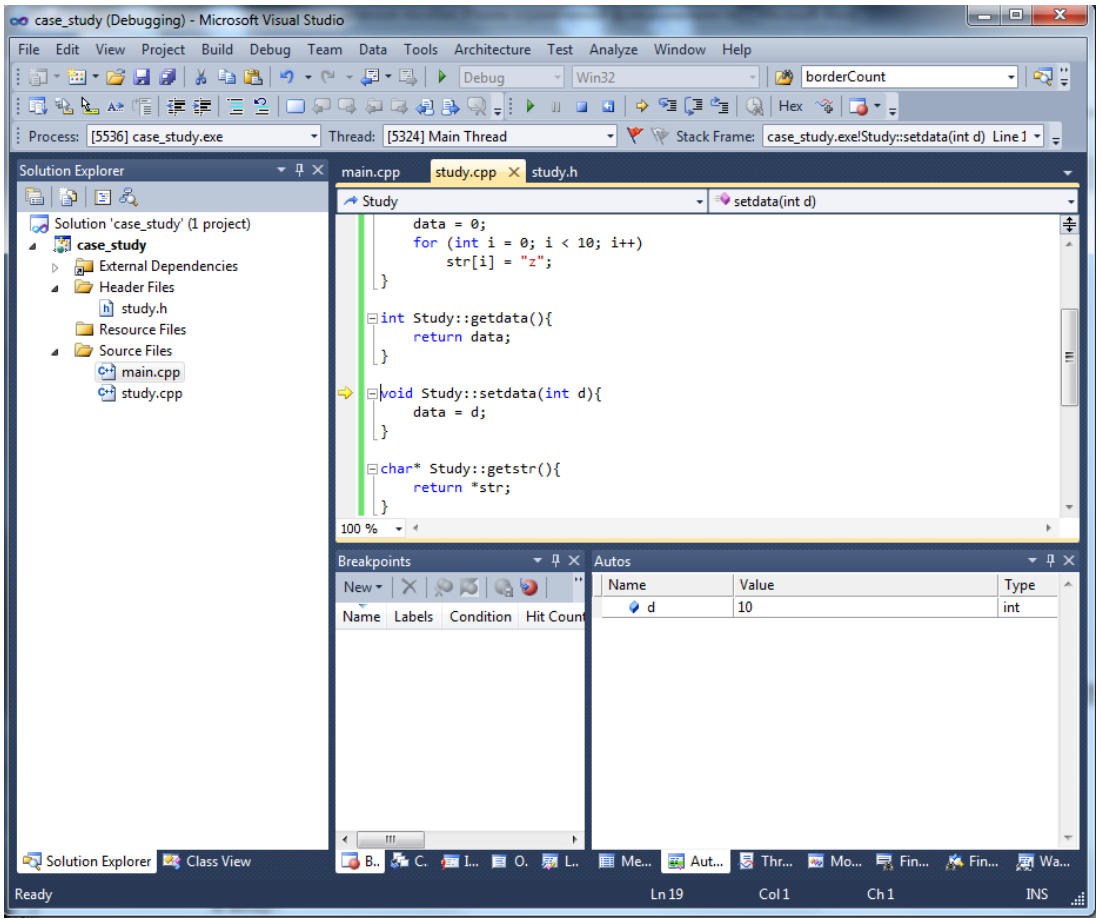


Рисунок 14 – Шаг с заходом в метод setdata

● «Шаг с выходом» («Step Out», Shift+F11) позволяет вернуться на уровень выше, если был совершен шаг с заходом. Так для описанного примера шага с заходом при выполнении шага с выходом желтая стрелка будет указывать уже на следующую строчку, а предыдущий оператор будет выполнен (Рисунок 15).

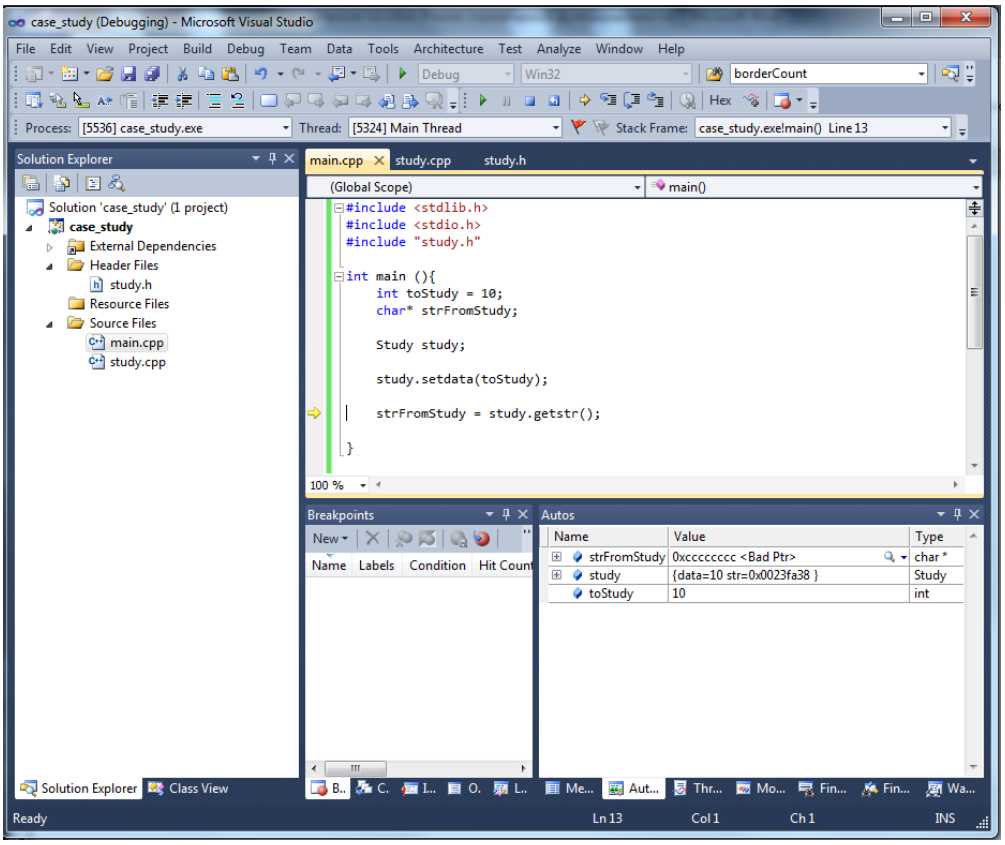


Рисунок 15 – Результат применения шага с выходом для метода setdata

1.4.2. Точки останова

Если необходимо проверить определенный участок кода, то ставят точку останова (breakpoint) на требуемой строке. Для этого необходимо поставить курсор на строку, на которой надо остановиться и нажимают F9. Точку останова можно поместить только на операторе (Рисунок 16).

После того как выбрали строку и поместили на нее точку останова, запускаем программу (F5). Программа будет исполняться до тех пор, пока не встретит точку останова. Это удобно, если существует объемный ввод данных или проект достаточно большой, а строка, вызывающая сомнения находится в середине исполнения.

Точка останова может иметь условие, при котором программа сделает на ней остановку. Иными словами, программа не будет останавливаться на выбранной точке, если не будет выполнено условие. Чтобы поставить условие на точку останова необходимо кликнуть правой кнопкой мыши на точке и в меню выбрать «Condition…» (Рисунок 17). При выборе этого пункта появится диалоговое окно «Breakpoint Condition» (Рисунок 18), где в текстовом поле

предоставлена возможность добавления условия. Теперь программа будет работать до тех пор, пока toStudy не будет равно 0.

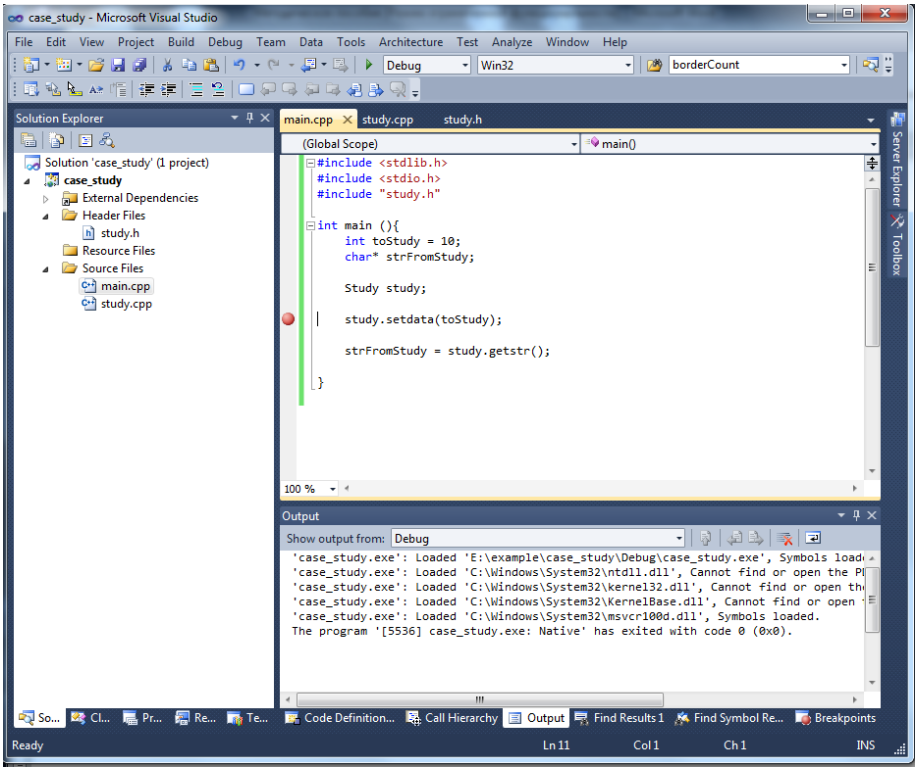


Рисунок 16 – Пример точки останова на функции setdata

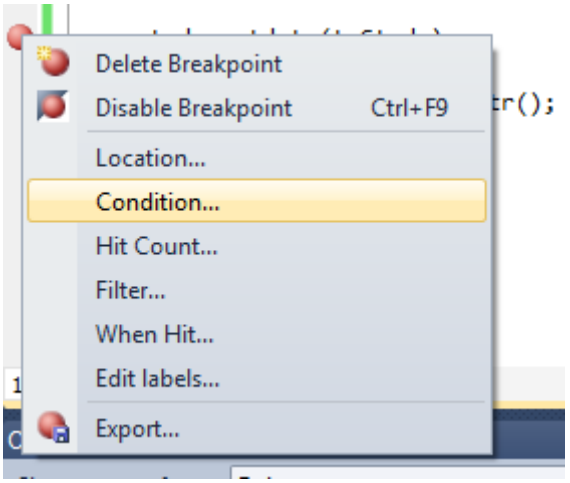


Рисунок 17 – Меню точки останова

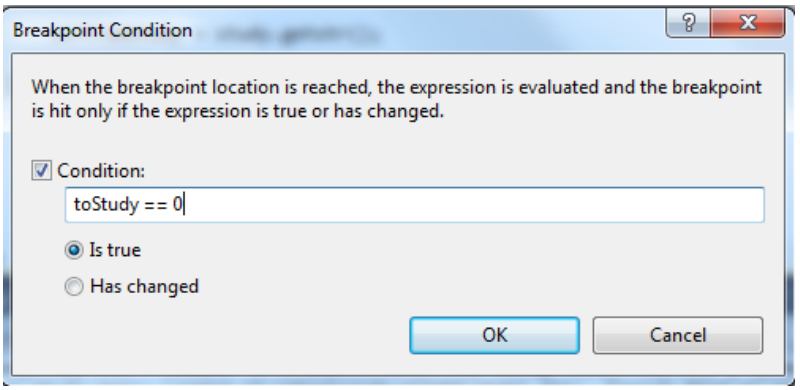


Рисунок 18 – Диалоговое окно установки условия на точку останова

1.5. Компиляция программы

Добавление файлов в проект и написание исходного кода - это часть процесса создания программы. Написанный код необходимо скомпилировать, чтобы получить исполняемый файл. Проект может быть консольного типа или графический. В лабораторных работах необходимо создавать консольное приложение. То есть ваши проекты могут быть запущены из командной строки. Они не имеют собственного графического интерфейса (Graphical Interface Unit, GUI) и компилируются в независимые исполняемые файлы.

После того как вы добавили все требуемые файлы в проект и написали исходный код, необходимы выполнить компиляцию, для этого нажать клавишу F7, или в меню выбрать “Build”, затем “Build solution”. В результате выполнения данной операции компилятор, если ошибок не возникло, в окне Output выведет следующую информацию “Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped”. Иначе будет выведен список ошибок и “Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped”.

Чтобы запустить скомпилированную программу, нажмите Ctrl+F5, или перейдите в меню “Debug” и выберите “Start Without Debugging”.

Или же можно выполнить команду запуска с компиляцией с помощью клавиши F5 или в меню выбрать “Debug”, а затем “Start”.

После выполнения команды запуска программы появится консольное окно, которое выведет информацию или предложит ввод, согласно алгоритму, который в программе описан.

2. СИСТЕМА КОНТРОЛЯ ВЕРСИЙ GIT

2.1. Общая информация о Git

Системы контроля предназначены для учета изменений (версий) в исходном коде, выполняемых группой пользователей. Системы контроля версий, как правило, позволяют получить: историю изменений как отдельного файла, так и проекта целиком, с возможностью извлечения исходного кода на любую из версий; учет изменений выполняемых независимо в так называемых ветках, по слиянию изменений, выполняемых пользователями и т.д.

Git является одной из самых распространенных систем контроля версий. Среди его преимуществ можно выделить высокую скорость работы, хорошую поддержку ведения параллельной разработки в ветках, отсутствие централизации.

В отличие от других систем Git хранит репозиторий (реестр всех изменений) локально, и большинство операций производятся также локально. Это обеспечивает с одной стороны надежность (в случае выхода из строя сервера с центральным хранилищем в качестве исходных данных может выступать репозиторий любого из пользователей), а с другой - высокую скорость работы. При фиксации изменения файла в репозитории Git сохраняет измененный файл, и соответствующая новая версия исходного кода будет содержать ссылку на измененный файл. Для неизмененных файлов ссылки не изменятся [5]. Локальный репозиторий синхронизируется с центральным репозиторием при помощи отдельных команд Git.

Для знакомства с Git потребуется установить на локальный компьютер клиентскую часть Git, а в центральный репозиторий можно хранить на одном из бесплатных и общедоступных сервисов, таких как GitHub, GitLab, BitBucket. В главе 2.2 рассказывается о том, как создать центральный репозиторий в GitHub. В последующих главах будут рассмотрены основные команды Git применяемые при разработке (практически не зависит от провайдера центрального репозитория).

2.2. Создание репозитория в GitHub

Для создания репозитория потребуется завести учетную запись в GitHub (https://github.com/): указать логин, email и задать пароль для работы с GitHub. После подтверждения email на домашней странице пользователя https://github.com/<логин> будет отображаться информация об активностях пользователя на GitHub и созданных им репозиториях - хранилищах исходного кода. На вкладке Repositories выводится список его репозиториев.

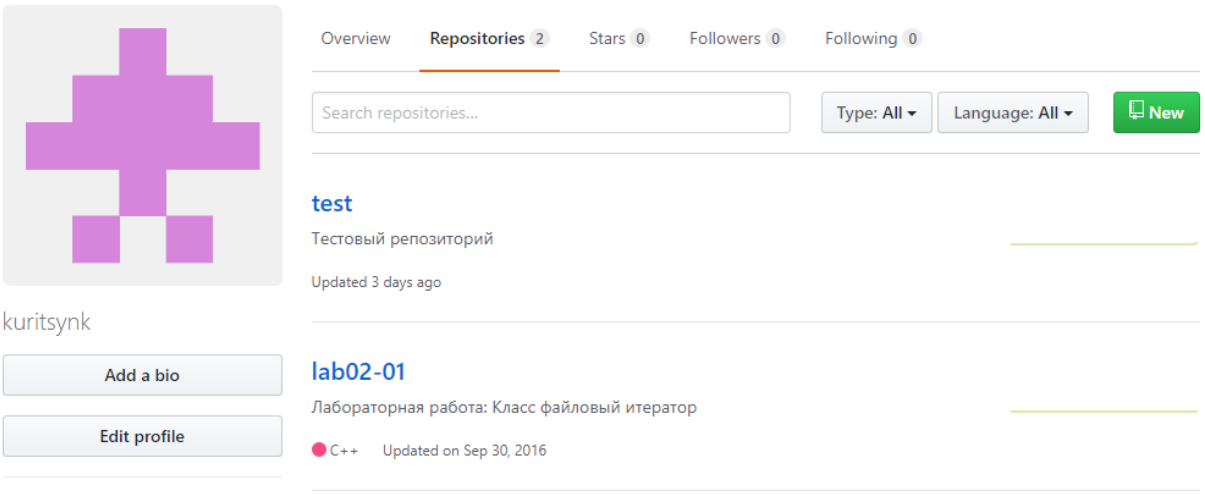


Рисунок 19 - Перечень репозиториев в веб-интерфейсе GitHub

Для создания нового репозитория необходимо нажать кнопку “New”, указать имя нового репозитория (например, test02) и дать текстовое описание (например, “Текстовый репозиторий”). Также полезно включить галку “Initialize this repository with a README”, чтобы репозиторий был не пустой.

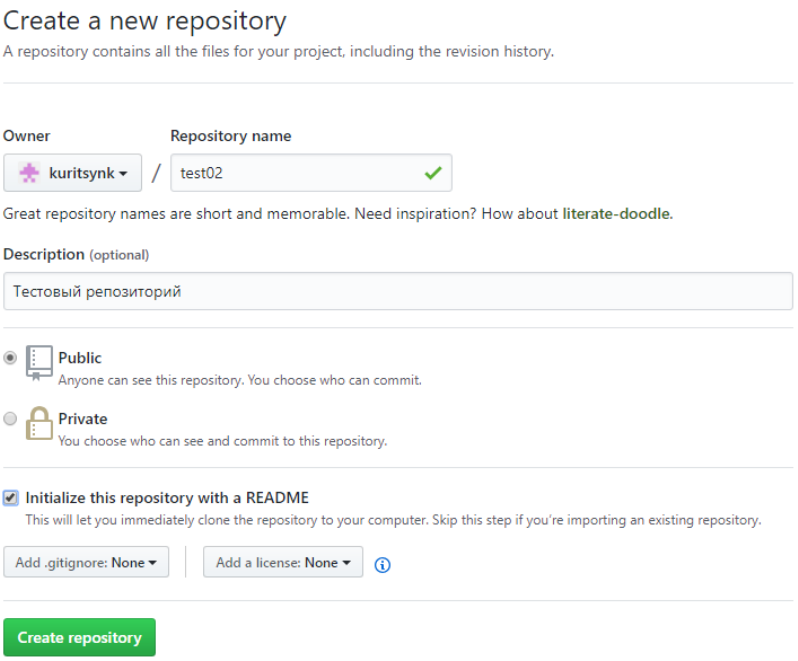


Рисунок 20 - Страница создания репозитория в веб-интерфейсе GitHub

После нажатия кнопки “Create repository” будет создан репозиторий test, имеющий адрес https://github.com/<логин>/test02, в котором присутствует файл README.md, содержимое которого отображается при открытии страницы с репозиторием.

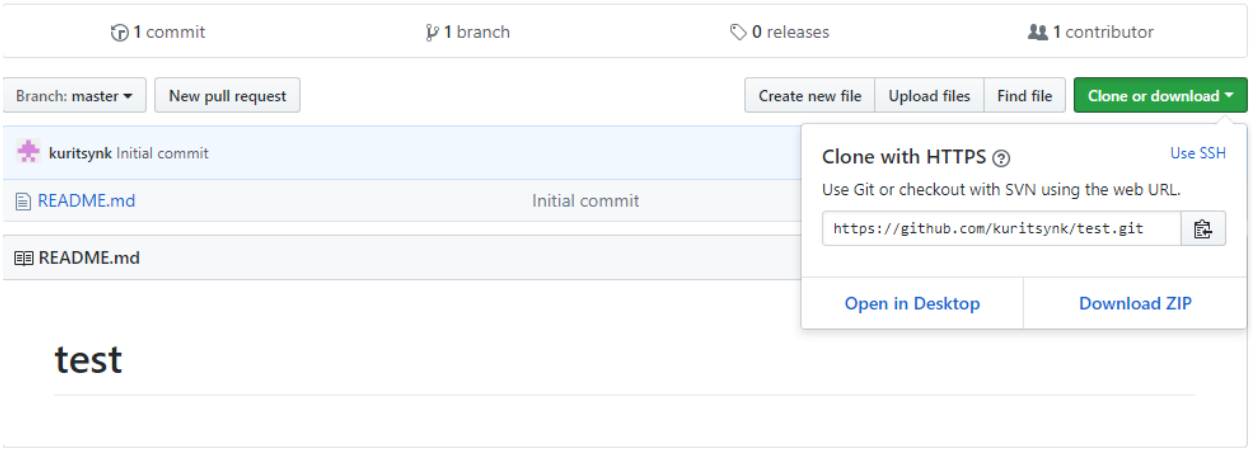


Рисунок 21 - Основная страница репозитория в веб-интерфейсе GitHub

На экране также отображается список последних изменений в исходном коде - список коммитов. В частности, поскольку при создании репозитория мы включили “Initialize this repository with a README” был автоматически создан коммит с комментарием “Initial commit” и файлом README.md, содержащим название репозитория.

Новый созданный репозиторий имеет адрес https://github.com/<логин>/test02.git. Адрес можно получить, нажав кнопку “Clone or download” на странице репозитория (рис. 21).

При дальнейшей работе с репозиторием веб-интерфейс не понадобится. Будем использовать клиентское приложение Git.

2.3. Создание локального репозитория

Чтобы начать работу с созданным репозиторием необходимо скопировать созданный (удаленный) репозиторий локально. Для этого нужно запустить Git Bash, открыть локальный каталог (пусть это будет C:\project\git), в котором будет располагаться репозиторий и выполнить команду clone.

```
$ cd c:/project/git

$ git clone https://github.com/kuritsynk/test.git
Cloning into 'test'...
remote: Enumerating objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 3
Unpacking objects: 100% (3/3), done.
```

В результате выполнения команды появится подкаталог test, который будет содержать файл README и локальный репозиторий в каталоге .git.

```
$ cd test

$ ls -A
.git/  README.md
```

Внутри каталога test строка приглашения кроме указания пользователя и текущего каталога в скобках добавилось слово master - название текущей ветки в локальном репозитории.

На данный момент локальный репозиторий синхронен с удаленным. Чтобы в этом убедиться нужно выполнить команду status.

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

В результате выполнения команды git сообщил, что текущая ветка - master синхронна с веткой origin/master - веткой master на удаленном сервере origin. Соответствие между локальной веткой master и удаленной веткой было автоматически установлено при выполнении команды clone. Также при выполнении clone был создан псевдоним origin для удаленного репозитория.

Посмотреть историю изменений в репозитории можно при помощи команды log

```
$ git log
commit 9b56ceb9080e35097059bf9ccdaac74a5df8e9a3 (HEAD -> master, origin/master,
origin/HEAD)
Author: kuritsynk <kuritsynk@mail.ru>
Date:   Thu Sep 27 12:36:31 2018 +0300

    Initial commit
```

В истории содержится информация об изменениях (коммитах) в репозитории. Шестнадцатеричная строка, начинающаяся с 9b56ce, представляет собой уникальный идентификатор коммита. Для многих целей можно указывать не всю строку, а ее часть, например, первые шесть символов. В информацию о коммите также входит указание автора и его e-mail, дату и время коммита и комментарий.

Имя и e-mail пользователя настраиваются в git. Для указания этих параметров нужно выполнить команду config.

```
$ git config --local user.name "Konstantin Kuritsyn"

$ git config --local user.email kuritsyn@mail.ru
```

Параметр --local означает, что настройки будут применяться только к текущему репозиторию.

Список параметров локального репозитория можно получить при помощи той же команды config.

```
$ git config --list --local
remote.origin.url=https://github.com/kuritsynk/test.git
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
branch.master.remote=origin
branch.master.merge=refs/heads/master
user.name=Konstantin Kuritsyn
user.email=kuritsyn@mail.ru
```

Команда log может выводить информацию в различных представлениях, в зависимости от параметров. Например, вывод в одну строку с кратким идентификатором коммита:

```
$ git log --format=oneline --abbrev-commit
9b56ceb (HEAD -> master, origin/master, origin/HEAD) Initial commit
```

Информация в скобках говорит нам о том, что ветка master сейчас ссылается на этот коммит (т.е.в этой ветке это последний коммит), а текущий указатель HEAD также ссылается на ветку master (т.е. текущей веткой является master). Наличие для коммиты строки origin/master, означает что в ветке master в удаленном репозитории указанный коммит является последним. Надо отметить, что команда log выдает информацию на основании локальных данных и сведения об указателях на origin являются актуальными на момент синхронизации с удаленным хранилищем (в нашем случае это выполнение команды clone).

При добавлении нового коммита в репозиторий в нем будет установлена ссылка на предыдущий коммит - коммит, на который указывает master. Автоматически указатель master будет сдвинут на новый коммит.

2.4. Добавление изменений в локальный репозиторий

2.4.1. Управление индексом

Создадим в Visual Studio новое консольное приложение в каталоге test: название проекта Test, имя решения - Test, решение размещаем в той же папке, что и проект. Добавим в проект файл main.cpp со следующим содержимым:

```
#include <iostream>

int main() {
      std::cout << "Hello user" << std::endl;
}
```

Теперь скомпилируем, сохраним проект и добавим его исходный код в репозиторий.

Добавление изменений в репозиторий происходит в два этапа. Вначале изменения добавляются в так называемый индекс, а затем производится фиксация индекса в репозитории в рамках одного коммита.

Чтобы узнать текущее состояние индекса нужно выполнить команду status:

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        Test/

nothing added to commit but untracked files present (use "git add" to track)
```

Git показал, что в рабочей директории обнаружен новый каталог Test, и если необходимо его добавить, то нужно выполнить команду add, указав имя файла.

```
$ git add Test/

$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   Test/.vs/Test/v16/.suo
        new file:   Test/.vs/Test/v16/Browse.VC.db
        new file:   Test/.vs/Test/v16/ipch/AutoPCH/c4955346c4b29c3d/MAIN.ipch
        new file:   Test/Debug/Test.exe
        new file:   Test/Debug/Test.ilk
        new file:   Test/Debug/Test.log
        new file:   Test/Debug/Test.pdb
        new file:   Test/Debug/Test.tlog/CL.command.1.tlog
        new file:   Test/Debug/Test.tlog/CL.read.1.tlog
        new file:   Test/Debug/Test.tlog/CL.write.1.tlog
        new file:   Test/Debug/Test.tlog/Test.lastbuildstate
        new file:   Test/Debug/Test.tlog/link.command.1.tlog
        new file:   Test/Debug/Test.tlog/link.read.1.tlog
        new file:   Test/Debug/Test.tlog/link.write.1.tlog
        new file:   Test/Debug/main.obj
        new file:   Test/Debug/vc142.idb
        new file:   Test/Debug/vc142.pdb
        new file:   Test/Test.sln
        new file:   Test/Test.vcxproj
        new file:   Test/Test.vcxproj.filters
        new file:   Test/Test.vcxproj.user
        new file:   Test/main.cpp
```

2.4.2. Настройка .gitignore

Теперь команда status показывает, что в индексе содержится целый набор файлов проекта, являющихся новыми для git. Но не все файлы нужно хранить в репозитории: не нужно хранить

результаты компиляции проекта, временные файлы, пользовательские настройки и т.д. В репозитории необходимо оставить только файлы исходного кода, позволяющие каждому, кто получит копию репозитория скомпилировать проект с требуемыми настройками (эталонными настройками, хранящимися в репозитории). Таким образом нам нужно удалить из индекса все, что не относится и сделать так, чтобы репозиторий ненужные файл не попадали.

```
$ git rm -r --cached Test/.vs Test/Debug *.filters *.user
rm 'Test/.vs/Test/v16/.suo'
rm 'Test/.vs/Test/v16/Browse.VC.db'
rm 'Test/.vs/Test/v16/ipch/AutoPCH/c4955346c4b29c3d/MAIN.ipch'
rm 'Test/Debug/Test.exe'
rm 'Test/Debug/Test.ilk'
rm 'Test/Debug/Test.log'
rm 'Test/Debug/Test.pdb'
rm 'Test/Debug/Test.tlog/CL.command.1.tlog'
rm 'Test/Debug/Test.tlog/CL.read.1.tlog'
rm 'Test/Debug/Test.tlog/CL.write.1.tlog'
rm 'Test/Debug/Test.tlog/Test.lastbuildstate'
rm 'Test/Debug/Test.tlog/link.command.1.tlog'
rm 'Test/Debug/Test.tlog/link.read.1.tlog'
rm 'Test/Debug/Test.tlog/link.write.1.tlog'
rm 'Test/Debug/main.obj'
rm 'Test/Debug/vc142.idb'
rm 'Test/Debug/vc142.pdb'
rm 'Test/Test.vcxproj.filters'
rm 'Test/Test.vcxproj.user'
```

И создаем в корневом каталоге файл .ignore, каждая строка которого содержит шаблон имени файла или каталога, которые git должен игнорировать при определении изменений в рабочем каталоге. Содержимое файла .gitignore (может в дальнейшем дополняться и меняться зависимости от версии Visual Studio, например, чтобы файлы релизных сборок не попадали в репозиторий нужно добавить строку Release/)

```
$ cat .gitignore
Debug/
.vs/
*.filters
*.user
```

Теперь команда status выдаст

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   Test/Test.sln
```

```
        new file:   Test/Test.vcxproj
        new file:   Test/main.cpp

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore
```

Теперь в индексе содержатся только необходимые для сборки проекта файлы. Вне индекса остался .gitignore. Его также следует разместить в репозитории, чтобы у всех пользователей репозитория он применялся. Добавляем его в индекс и смотрим, какие изменения попадут в коммит.

```
$ git add .gitignore

$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   .gitignore
        new file:   Test/Test.sln
        new file:   Test/Test.vcxproj
        new file:   Test/main.cpp
```

2.4.3. Фиксация изменений

Теперь мы готовы внести изменения в репозиторий. Четыре новых файла зафиксируем в рамках одного изменения. Для этого выполним команду commit указав обязательный комментарий (если его не указать в командной строке, то будет git откроет редактор, в котором потребуется набрать текст комментария). В качестве комментария обычно указывается номер задачи (в соответствующей системе трекинга) и суть вносимых изменений, сформулированных в виде указания (глагол в инфинитиве).

```
$ git commit -m "Создать проект Test"
[master fb6f137] Создать проект Test
 4 files changed, 171 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 Test/Test.sln
 create mode 100644 Test/Test.vcxproj
 create mode 100644 Test/main.cpp
```

Мы создали коммит fb6f137 в ветке master нашего репозитория. Посмотрим, что теперь произошло с индексом

```
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

Индекс пуст, изменений в рабочих директориях с точки зрения git тоже нет. Git также сообщил, что наша ветка master опережает удаленную ветку master на сервере origin на один коммит. Это логично, поскольку команда commit фиксирует изменения только в локальном репозитории. Посмотрим, как теперь выглядит история изменений.

```
$ git log
commit fb6f137b9c3361a3caf35e7d111d2620e0da6500 (HEAD -> master)
Author: Konstantin Kuritsyn <kuritsyn@mail.ru>
Date:   Fri Jul 12 17:55:30 2019 +0300

    Создать проект Test

commit 9b56ceb9080e35097059bf9ccdaac74a5df8e9a3 (origin/master, origin/HEAD)
Author: kuritsynk <kuritsynk@mail.ru>
Date:   Thu Sep 27 12:36:31 2018 +0300

    Initial commit
```

В логе присутствуют два изменения. При этом master ссылается на второй коммит, а origin/master по-прежнему на первый.

2.4.4.Работа с удаленным репозиторием

Чтобы отправить изменения на удаленный репозиторий необходимо выполнить команду push. При этом все коммиты будут отправлены на тот удаленный репозиторий, из которого ранее был получен локальный репозиторий при помощи команды clone - на сервер origin. Если с момента clone кто-то успел сделать push в удаленный репозиторий (в ветку master), то наш запрос команды push будет отклонен, и потребуется вначале принять изменения из удаленного репозитория, разрешить конфликты (если такие возникнут), и повторить вызов push.

```
$ git push
Counting objects: 7, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (7/7), 1.95 KiB | 998.00 KiB/s, done.
Total 7 (delta 0), reused 0 (delta 0)
To https://github.com/kuritsynk/test.git
   9b56ceb..fb6f137  master -> master
```

После успешной передачи изменений на удаленный сервер команда log выведет следующую картину:

```
$ git log
commit fb6f137b9c3361a3caf35e7d111d2620e0da6500 (HEAD -> master, origin/master,
origin/HEAD)
Author: Konstantin Kuritsyn <kuritsyn@mail.ru>
Date:   Fri Jul 12 17:55:30 2019 +0300

    Создать проект Test

commit 9b56ceb9080e35097059bf9ccdaac74a5df8e9a3
Author: kuritsynk <kuritsynk@mail.ru>
Date:   Thu Sep 27 12:36:31 2018 +0300

    Initial commit
```

Изменения по сравнению с результатами предыдущего вызова log заключаются в том, что итогом выполнения команды push с т.з. локального репозитория стало перемещение указателей origin/master и origin/HEAD на последний коммит (на которые ссылаются локальные master и HEAD). Это значит, что все нижележащие коммиты также содержатся в удаленном репозитории. Чтобы убедиться в этом можно открыть веб-интерфейс репозитория (https://github.com/kuritsynk/test.git) и увидеть что в нем содержится два коммита.

Создадим еще один локальный каталог и в него поместим еще один клон удаленного репозитория. Этот второй локальный репозиторий будет демонстрировать одновременную работу разных пользователей.

Создание второго локального репозитория (/c/project/git/second/test)

```
$ mkdir /c/project/git/second/

$ cd /c/project/git/second/

$ git clone https://github.com/kuritsynk/test.git
Cloning into 'test'...
remote: Enumerating objects: 10, done.
remote: Total 10 (delta 0), reused 0 (delta 0), pack-reused 10
Unpacking objects: 100% (10/10), done.

$ cd /c/project/git/second/test

$ git log
commit fb6f137b9c3361a3caf35e7d111d2620e0da6500 (HEAD -> master, origin/master,
origin/HEAD)
```

```
Author: Konstantin Kuritsyn <kuritsyn@mail.ru>
Date:   Fri Jul 12 17:55:30 2019 +0300

    Создать проект Test

commit 9b56ceb9080e35097059bf9ccdaac74a5df8e9a3
Author: kuritsynk <kuritsynk@mail.ru>
Date:   Thu Sep 27 12:36:31 2018 +0300

    Initial commit
```

Внесем изменения в файл README.md в рабочем каталоге второго репозитория:

Второй репозиторий (/c/project/git/second/test). Содержимое файла README.md

```
# Тестовый репозиторий

Знакомство с git.
```

Зафиксируем изменения в локальном репозитории и отправим их на удаленный сервер:

Второй репозиторий (/c/project/git/second/test)

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")

$ git commit -a -m "Исправить README"
[master f88fad9] Исправить README
 Committer: Константин А. Курицын <kuritsyn@yamoney.ru>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly. Run the
following command and follow the instructions in your editor to edit
your configuration file:

    git config --global --edit

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

 1 file changed, 3 insertions(+), 1 deletion(-)

$ git push
```

```
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 412 bytes | 412.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/kuritsynk/test.git
   fb6f137..f88fad9  master -> master

$ git log --oneline --graph
* f88fad9 (HEAD -> master, origin/master, origin/HEAD) Исправить README
* fb6f137 Создать проект Test
* 9b56ceb Initial commit
```

Во втором удаленном репозитории теперь три коммита, и коммиты доставлены на удаленный репозиторий.

Если же переключиться на первый репозиторий и посмотреть его состояние, то ожидаемо в нем третий коммит не будет виден.

Первый репозиторий

```
$ git log --oneline -- graph
* fb6f137 (HEAD -> master, origin/master, origin/HEAD) Создать проект Test
* 9b56ceb Initial commit
```

Чтобы получить изменения с удаленного репозитория мы должны выполнить команду fetch.

Первый репозиторий

```
$ git fetch
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/kuritsynk/test
   fb6f137..f88fad9  master     -> origin/master

$ git log --oneline --graph
* fb6f137 (HEAD -> master) Создать проект Test
* 9b56ceb Initial commit

$ git log --oneline --all --graph
* f88fad9 (origin/master, origin/HEAD) Исправить README
* fb6f137 (HEAD -> master) Создать проект Test
* 9b56ceb Initial commit
```

После того, как была выполнена команда fetch и получены коммиты с удаленного репозитория, команда log вывела только два коммита, поскольку без ключа --all она выводит лог только локальной ветки, а в локальной ветке master еще нет третьего коммита, он есть в удаленной

ветке origin/master. Выполнение команды log с ключом --all дает возможность увидеть все три коммита. Как видно ветка master отстает от origin/master на один коммит. Чтобы принять изменения в свою локальную ветку необходимо выполнить слияние при помощи команды merge.

Первый репозиторий

```
$ git merge
Updating fb6f137..f88fad9
Fast-forward
 README.md | 4 +++-
 1 file changed, 3 insertions(+), 1 deletion(-)

$ git log --oneline --graph
* f88fad9 (HEAD -> master, origin/master, origin/HEAD) Исправить README
* fb6f137 Создать проект Test
* 9b56ceb Initial commit
```

Поскольку в локальной ветке master не выполнялось изменений, то результате выполнения merge (по умолчанию выполняет слияние с соответствующей удаленной меткой) указатель master просто переместился на указатель origin/master.

Так как получение изменений с удаленного репозитория и последующее слияние -- достаточно типичные действия, в git предусмотрена отдельная команда pull, которая фактически выполняет последовательный вызов команд fetch и merge.

2.4.5. Ветки

Чтобы понять, что такое ветки в git следует разобраться со структурой коммита. Коммит, как мы выяснили, представляет собой некоторые данные, которые характеризуются идентификатором (хэшем), датой и временем, автором коммита, комментарием. Коммит также хранит ссылку на дерево измененных в рамках коммита файлов.

Кроме указанных данных каждый коммит хранит ссылку (или ссылки, если это коммит слияния) на предыдущие (родительские) коммиты. Если изменения делаются в одной ветке master последовательно, то цепочка коммитов будет представлять собой односвязный список. Команда log с ключом --parents добавляет в вывод указание родительских коммитов.

Первый репозиторий

```
$ git log --oneline --graph --parents
* f88fad9 fb6f137 (HEAD -> master, origin/master, origin/HEAD) Исправить README
* fb6f137 9b56ceb Создать проект Test
```

```
* 9b56ceb Initial commit
```

Ветка в git – это именованный (по умолчанию, master) указатель на коммит, который автоматически перемещается, когда делается коммит. HEAD - это указатель на текущую ветку. Соответственно, когда мы фиксируем изменения, появляется новый коммит, в качестве родительского коммита у него будет указан коммит, на который ссылается текущая ветка master (на которую указывает HEAD), затем указатель master начинает ссылаться на новый коммит.

Создание ветки в Git - это создание нового указателя, а переключение между ветками - перемещение указателя HEAD и применение данных репозитория к локальному каталогу.

Список веток и информацию о текущей ветке можно получить с использованием команды branch:

```
$ git branch
* master
```

Вывод команды говорит о том, что в локальном репозитории есть одна ветка master и она является текущей.

Использование веток позволяет вести параллельную работу над задачами. Перед началом выполнения задачи создается ветка, в ней выполняются изменения для решения задачи и затем ветка сливается с основной веткой.

Предположим в нашем тестовом проекте будут две задачи:

1. Запрашивать имя пользователя и выводить его на экран в строке приветствия.
2. После вывода приветствия сразу не завершать приложение, дожидаясь ввода строки от пользователя.

Решать задачи будем параллельно в соответствующих двух репозиториях. В первом репозитории создадим ветку “feature/1-ask-user-name” при помощи команды branch:

Первый репозиторий

```
$ git branch feature/1-ask-user-name

$ git branch
  feature/1-ask-user-name
* master
```

Команда branch создала новую ветку, но не переключилась на нее. Чтобы выполнить переключение на другую ветку (в процессе переключения локальный рабочий каталог будет заменен соответствующими ветке файлами), нужно выполнить команду checkout:

Первый репозиторий

```
$ git checkout feature/1-ask-user-name
Switched to branch 'feature/1-ask-user-name'

$ git branch
* feature/1-ask-user-name
  master
```

Теперь внесем изменения в файл main.cpp проекта (в каталоге первого репозитория /c/project/git/test):

Первый репозиторий. Файл main.cpp

```
#include <iostream>
#include <string>

std::string ask_user_name() {
	std::cout << "Enter user name:";
	std::string user_name;
	std::cin >> user_name;
	return user_name;
}

int main() {
	std::cout << "Hello, " << ask_user_name() << "!" << std::endl;
}
```

Зафиксируем изменения в локальном репозитории:

Первый репозиторий

```
$ git status
On branch feature/1-ask-user-name
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   Test/main.cpp

no changes added to commit (use "git add" and/or "git commit -a")

$ git add Test/main.cpp

$ git status
On branch feature/1-ask-user-name
```

```
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   Test/main.cpp


$ git commit -m "#1 Запрашивать имя пользователя и выводить его на экран в строке
приветствия"
[feature/1-ask-user-name 3407d6e] #1 Запрашивать имя пользователя и выводить его на
экран в строке приветствия
 1 file changed, 9 insertions(+), 1 deletion(-)
```

Теперь переключимся на второй репозиторий и в нем создадим ветку для работы над второй задачей:

Второй репозиторий

```
$ git branch feature/2-do-not-close-app-immediately

$ git branch
  feature/2-do-not-close-app-immediately
* master

$ git checkout feature/2-do-not-close-app-immediately
Switched to branch 'feature/2-do-not-close-app-immediately'
```

Изменим файл main.cpp проекта (в каталоге второго репозитория /c/project/git/second/test):

Второй репозиторий. Файл main.cpp

```
#include <iostream>

int main() {
      std::cout << "Hello user" << std::endl;

      std::cin.get();
}
```

Заметим, что в файле main.cpp во втором репозитории еще не учтены изменения, которые внесены в рамках решения первой задачи.

Зафиксируем изменения в локальном репозитории и попробуем отправить изменения на удаленный сервер.

Второй репозиторий

```
$ git status
On branch feature/2-do-not-close-app-immediately
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
```

```
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   Test/main.cpp

no changes added to commit (use "git add" and/or "git commit -a")

$ git commit -a -m "#2 После вывода приветствия сразу не завершать приложение,
дожидаясь ввода строки от пользователя"
[feature/2-do-not-close-app-immediately 8b466f6] #2 После вывода приветствия сразу не
завершать приложение, дожидаясь ввода строки от пользователя
 1 file changed, 2 insertions(+)

$ git push
fatal: The current branch feature/2-do-not-close-app-immediately has no upstream
branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin feature/2-do-not-close-app-immediately
```

Git при попытке выполнить push сообщил, что для нашей ветки в локальном репозитории не установлено соответствие с веткой удаленного репозитория и предложил вариант команды, которая устанавливает такое соответствие. Выполним его и посмотрим лог.

Второй репозиторий

```
$ git push --set-upstream origin feature/2-do-not-close-app-immediately
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 497 bytes | 497.00 KiB/s, done.
Total 4 (delta 3), reused 0 (delta 0)
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
remote:
remote: Create a pull request for 'feature/2-do-not-close-app-immediately' on GitHub
by visiting:
remote:
https://github.com/kuritsynk/test/pull/new/feature/2-do-not-close-app-immediately
remote:
To https://github.com/kuritsynk/test.git
 * [new branch]      feature/2-do-not-close-app-immediately ->
feature/2-do-not-close-app-immediately
Branch 'feature/2-do-not-close-app-immediately' set up to track remote branch
'feature/2-do-not-close-app-immediately' from 'origin'.

$ git log --oneline --graph --all
* 8b466f6 (HEAD -> feature/2-do-not-close-app-immediately,
origin/feature/2-do-not-close-app-immediately) #2 После вывода приветствия сразу не
завершать приложение, дожидаясь ввода строки от пользователя
* f88fad9 (origin/master, origin/HEAD, master) Исправить README
* fb6f137 Создать проект Test
* 9b56ceb Initial commit
```

Из лога видно, что в удаленном репозитории теперь есть две ветки master и feature/2-do-not-close-app-immediately, которые ссылаются на разные коммиты - ветки разошлись.

Теперь отправим изменения из первого репозитория на удаленный сервер.

Первый репозиторий

```
$ git push --set-upstream origin feature/1-ask-user-name
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 552 bytes | 552.00 KiB/s, done.
Total 4 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
remote:
remote: Create a pull request for 'feature/1-ask-user-name' on GitHub by visiting:
remote:      https://github.com/kuritsynk/test/pull/new/feature/1-ask-user-name
remote:
To https://github.com/kuritsynk/test.git
 * [new branch]      feature/1-ask-user-name -> feature/1-ask-user-name
Branch 'feature/1-ask-user-name' set up to track remote branch
'feature/1-ask-user-name' from 'origin'.

$ git log --oneline --graph --all
* 3407d6e (HEAD -> feature/1-ask-user-name, origin/feature/1-ask-user-name) #1
Запрашивать имя пользователя и выводить его на экран в строке приветствия
* f88fad9 (origin/master, origin/HEAD, master) Исправить README
* fb6f137 Создать проект Test
* 9b56ceb Initial commit
```

По данным лога видно, что в на удаленном сервере также ветка feature/1-ask-user-name, но нет ветки feature/2-do-not-close-app-immediately. Для того чтобы увидеть актуальное состояние удаленного сервера, нужно вытянуть данные с удаленного репозитория командой pull:

Первый репозиторий

```
$ git pull
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 4 (delta 3), reused 4 (delta 3), pack-reused 0
Unpacking objects: 100% (4/4), done.
From https://github.com/kuritsynk/test
 * [new branch]      feature/2-do-not-close-app-immediately ->
origin/feature/2-do-not-close-app-immediately
Already up to date.

$ git log --oneline --graph --all
* 8b466f6 (origin/feature/2-do-not-close-app-immediately) #2 После вывода приветствия
сразу не завершать приложение, дожидаясь ввода строки от пользователя
```

```
| * 3407d6e (HEAD -> feature/1-ask-user-name, origin/feature/1-ask-user-name) #1
Запрашивать имя пользователя и выводить его на экран в строке приветствия
|/
* f88fad9 (origin/master, origin/HEAD, master) Исправить README
* fb6f137 Создать проект Test
* 9b56ceb Initial commit
```

Команда pull сообщила, что с удаленного сервера получена информация о новой ветке feature/2-do-not-close-app-immediately. И теперь лог показывает актуальную информацию: в локальном репозитории есть 2 ветки - master и feature/1-ask-user-name, в удаленном репозитории их три - master, feature/1-ask-user-name, feature/2-do-not-close-app-immediately. При этом обе ветки ссылаются на разные коммиты, имеющие общего родителя - коммит f88fad9. На этот коммит ссылается master.

Изменения по задаче могут состоять из нескольких коммитов. Если все необходимые изменения по задаче внесены и доставлены на удаленный сервер, то ветка может быть слита с исходной (master). В зависимости от принятых в команде стандартов разработки и используемого удаленного репозитория, возможны разные варианты слияния. Часто по завершению изменений в ветке и заливки их на удаленный сервер, создается запрос на слияние (Merge request в gitlab, Pull request в github, bitbucket), в котором будут видны все изменения. Запрос может быть проверен другими разработчиками проекта, и либо отвергнут, либо принят. В первом случае необходимо устранить замечания и добавить новые коммиты в ветку (они автоматически отобразятся в запросе на слияние). В случае принятия запроса будет выполнено слияние ветки в целевую. При этом исходная ветка может быть удалена, поскольку выполнение задачи завершено (здесь надо помнить, что ветка - это лишь указатель, и ее удаление не влияет на историю коммитов в git).

Выполним вручную в первом репозитории слияние изменений из ветки feature/1-ask-user-name в master. Для этого нужно переключиться в master и выполнить команду merge.

Первый репозиторий

```
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.

$ git merge feature/1-ask-user-name
Updating f88fad9..3407d6e
Fast-forward
```

```
 Test/main.cpp | 10 +++++++++-
 1 file changed, 9 insertions(+), 1 deletion(-)
```

Поскольку в ветке master не было коммитов с с момента отделения ветки feature/1-ask-user-name слияние оказалось исключительно простым: указатель master переместился на feature/1-ask-user-name.

Чтобы получить перечень веток, слитых с текущей, нужно выполнить команду branch с ключое --merged, чтобы посмотреть ветки, которые еще не слиты -- с ключом --no-merged:

Первый репозиторий

```
$ git branch --no-merged

$ git branch --merged
  feature/1-ask-user-name
* master
```

Теперь ветку feature/1-ask-user-name можно удалить (т.к. все изменения теперь доступны в master) командой branch с ключом -d и поместить изменения в удаленный репозиторий:

Первый репозиторий

```
$ git branch -d feature/1-ask-user-name
Deleted branch feature/1-ask-user-name (was 3407d6e).

$ git push
Total 0 (delta 0), reused 0 (delta 0)
To https://github.com/kuritsynk/test.git
   f88fad9..3407d6e  master -> master

$ git log --oneline --graph --all
* 8b466f6 (origin/feature/2-do-not-close-app-immediately) #2 После вывода приветствия
сразу не завершать приложение, дожидаясь ввода строки от пользователя
| * 3407d6e (HEAD -> master, origin/master, origin/feature/1-ask-user-name,
origin/HEAD) #1 Запрашивать имя пользователя и выводить его на экран в строке
приветствия
|/
* f88fad9 Исправить README
* fb6f137 Создать проект Test
* 9b56ceb Initial commit
```

Теперь завершим разработку второй задачи. Во втором репозитории переключимся на ветку мастер, выполним слияние изменений из ветки feature/2-do-not-clode-app-immediately и попробуем поместить изменения на удаленный сервер командой push:

Второй репозиторий

```
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.

$ git merge feature/2-do-not-close-app-immediately
Updating f88fad9..8b466f6
Fast-forward
 Test/main.cpp | 2 ++
 1 file changed, 2 insertions(+)

$ git push
To https://github.com/kuritsynk/test.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/kuritsynk/test.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Как мы видим слияние в master прошло также через перемотку указателя master на 2-do-not-clode-app-immediately. А вот попытка поместить эти изменения на удаленный сервер завершилась неуспехом по причине того, что на удаленном сервере master уже успел сдвинуться по отношению к локальному. Git предложил выполнить команду pull в ветке master, чтобы выполнить слияние изменений из origin/master в локальный master.

Второй репозиторий

```
$ git pull
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2), pack-reused 0
Unpacking objects: 100% (4/4), done.
From https://github.com/kuritsynk/test
   f88fad9..3407d6e  master                  -> origin/master
 * [new branch]      feature/1-ask-user-name -> origin/feature/1-ask-user-name
Auto-merging Test/main.cpp
CONFLICT (content): Merge conflict in Test/main.cpp
Automatic merge failed; fix conflicts and then commit the result
```

Автоматическое слияние в нашем случае не получилось. Поскольку в обеих задачах менялся один и тот же файл main.cpp и изменения были в соседних строчках, git не смог разобраться в том, что же должно получиться в итоге и пометил файл Test/main.cpp как конфликтный. Следует открыть файл main.cpp:

Второй репозиторий. Содержимое файла main.cpp с конфликтом слияния

```
#include <iostream>
#include <string>

std::string ask_user_name() {
	std::cout << "Enter user name:";
	std::string user_name;
	std::cin >> user_name;
	return user_name;
}

int main() {
<<<<<<< HEAD
	std::cout << "Hello user" << std::endl;

	std::cin.get();
=======
	std::cout << "Hello, " << ask_user_name() << "!" << std::endl;
>>>>>>> 3407d6e65668439aedaa354aa14013cc8b0d96cc
}
```

В файле содержится промежуточный результат слияния, содержащий как и автоматически принятые изменения с удаленной ветки (например, объявление и реализация функции ask_user_name), так и блоки с неразрешенным конфликтом. В выделенных областях содержатся локальные изменения и изменения в удаленной ветке. Необходимо отредактировать файл, оставив только необходимое:

Второй репозиторий. Содержимое файла main.cpp с разрешенным конфликтом

```
#include <iostream>
#include <string>

std::string ask_user_name() {
	std::cout << "Enter user name:";
	std::string user_name;
	std::cin >> user_name;
	return user_name;
}

int main() {
	std::cout << "Hello, " << ask_user_name() << "!" << std::endl;

	std::cin.get();
}
```

Теперь нужно зафиксировать изменения - это будет коммит слияния. И повторно отправить изменения в удаленный репозиторий.

Второй репозиторий

```
$ git status
On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 1 different commits each, respectively.
  (use "git pull" to merge the remote branch into yours)

You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   Test/main.cpp

no changes added to commit (use "git add" and/or "git commit -a")

$ git commit -a -m "Merge feature/2-do-not-close-app-immediately to master"
[master f0c36e9] Merge feature/2-do-not-close-app-immediately to master

$ git push
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 396 bytes | 396.00 KiB/s, done.
Total 4 (delta 3), reused 0 (delta 0)
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
To https://github.com/kuritsynk/test.git
   3407d6e..f0c36e9  master -> master
```

Теперь команда push успешно выполнена и можно посмотреть, как выглядит история изменений в репозитории:

Второй репозиторий

```
$ git log --oneline --graph --all
*   f0c36e9 (HEAD -> master, origin/master, origin/HEAD) Merge
feature/2-do-not-close-app-immediately to master
|\
| * 3407d6e (origin/feature/1-ask-user-name) #1 Запрашивать имя пользователя и
выводить его на экран в строке приветствия
* | 8b466f6 (origin/feature/2-do-not-close-app-immediately,
feature/2-do-not-close-app-immediately) #2 После вывода приветствия сразу не
завершать приложение, дожидаясь ввода строки от пользователя
|/
* f88fad9 Исправить README
* fb6f137 Создать проект Test
* 9b56ceb Initial commit
```

Как видно у коммита слияния есть два родительских коммита. Теперь в master содержатся изменения от обеих веток.

3. Создание класса и иерархия классов

3.1. Создание класса

Для определения пользовательского типа данных, требуется ключевое слово `class`. Это слово сообщает компилятору, что дальше будет содержаться описание класса.

Синтаксис написания класса следующий:

```
class <ИМЯ_КЛАССА> {
   ...
};
```

Класс содержит объявления членов класса – элементов данных и элементов функций. Элементы данных называют полями класса, а элементы функций – методами класса.

В классе реализованы базовые принципы ООП:

- абстракция данных - в классе описываются только те параметры и действия, которые наиболее характерны для него и достаточны для полной работы с ним;
- инкапсуляция - в классах объединяются данные и методы (функции) для работы с ними, так как лишь через методы возможен доступ к закрытым данным класса;
- полиморфизм – это возможность использования одних и тех же методов для работы с различными объектами базового и классов-наследников.

Пример класса приведен ниже.

```
class Study {
public:
   Study();
   Study(int d);
   Study (int d1, int d2);
   ~Study();
   void setData(int d);
   int getData();
   void setStr(char* s);
   char* getStr();
private:
   int data1, data2;
   char* str[10];
};
```

Данное объявление класса пишется в h-файле. А реализация методов, конструкторов и деструкторов в cpp-файле.

```
void Study::setData(int d){
  data = d;
```

```
}
int Study::getData(){
  return data;
}
```

Имя класса, например Study, является новым типом данных. Переменная данного типа будет объектом (экземпляром) класса, например, `Study s`. Переменные и методы, объявленные в разделе класса private, имеют область видимости в пределах класса. Видимыми вне класса являются переменные и методы, объявленные в областях protected и public.

Модификатор доступа public не ограничивает доступ к объявленным в нем переменным и методам. Об этом необходимо помнить при написании класса, так как пользователь может исказить открытые данные, что может негативно повлиять на работу программы, вплоть до выхода из строя. Методы, расположенные в открытом доступе обязаны проверять данные которые принимают от пользователя на корректность.

Данные класса располагаются в области private, а методы класса, как правило, в области public. Не все методы имеет смысл делать общедоступными, то есть частью интерфейса класса. Некоторые методы (сервисные методы) остаются закрытыми и служат в качестве вспомогательных для других функций класса, для этого определен модификаторы private и protected. Сервисный метод не является частью интерфейса. Это закрытый метод, который поддерживает работу открытых методов класса. В protected как правило располагают именно сервисные методы, которые не доступны пользователю, но открыты для наследников класса.

Без указания области доступа (или public, или protected, или private) компилятор по умолчанию установит доступ private. В отличие от класса, переменные и функции структуры по умолчанию являются открытыми.

В функции main создание объекта аналогично объявлению переменной встроенного типа данных, например, Study s1. Вызов методов через объект класса осуществляется одним из двух способов:

- имя_объекта**.**имя_функции(аргументы)

  ```
  Study s;
  s.setaData(5);
  ```

- указатель_на_объект -> имя_функции(аргументы)

  ```
  Study *s(10);
  s->setData(50);
  ```

При создании объекта класса вызывается конструктор, а при окончании его использования вызывается деструктор для освобождения памяти. Так как при написании класса только программист понимает его назначение и допустимые величины, которые могут хранится в полях данных класса, необходимо определять конструкторы, чтобы определить назначение класса для

сторонних пользователей программы, а также для сообщения компилятору о корректности или некорректности использования объекта класса.

3.2. Конструкторы класса

Конструктор – это метод класса, который называется также как класс. Конструктор вызывается автоматически при создании объекта класса. Он инициализирует объект и переменные класса. У конструктора нет возвращаемого значения, но он может иметь аргументы и быть перегруженным.

Бывает такое, что необходимо создать объект класса, но начальные значения еще не известны, но задать их необходимо, чтобы сообщить о том, что класс не содержит реальных данных. Это инициализация объекта класса. Для этого определяется конструктор по умолчанию, для него не требуется передавать параметры, он служит именно для сообщения о том, что объект создан, но реальные данные в него не добавлены.

Ниже приведен пример написания конструктора без параметров для класса `Study`.

```
Study::Study() {
    data1 = 0;
   data2 = 0;
    for (int i = 0; i < 10; i++)
         str[i] = "a";
}
```

Конструктор с одним и двумя параметрами будет выглядеть следующим образом:

```
Study::Study(int d) {
    data1 = d;
    data2 = 0;
    for (int i = 0; i < 10; i++)
         str[i] = "a";
}
Study::Study(int d1, int d2){
    data1 = d1;
    data2 = d2;
     for (int i = 0; i < 10; i++)
         str[i] = "a";
}
```

Для инициализации полей также можно использовать список инициализации. Для класса `Study` он может иметь вид:

```
Study::Study() : data(0) {
    for (int i = 0; i < 10; i++)
         str[i] = "a";
```

```
}
```

Список инициализации удобен в использовании при наследовании классов и передачи с помощью него параметров для инициализации. О таком использовании рассказано в разделе 3.6.

Важным замечанием при создании конструкторов класса является следующее: если программист определил один конструктор с параметром, то он обязан пользоваться именно им, создание объекта без аргументов для конструктора становится невозможно.

Передача аргументов в конструктор при создании массива объектов происходит следующим образом:

Study s1[3] = {1, 2, 3}; // вызов конструктора с одним параметром

Study s2[3] = {{1,2},{2,3},{3,4}}; // вызов конструктора с двумя параметрами

Данные в этом случае присваиваются позиционно, на основе порядке, в котором объявлены в классе, поэтому важно соблюдение типов данных и передаваемых значений. Если же у вас не определено конструктора с параметрами, а данные находятся в закрытой области видимости относительно пользователя, то такая инициализация невозможна, необходимо использовать соответствующие параметрам функции “set”.

Если массив объектов объявлен без передаваемых для инициализации значений, то вызывается конструктор по умолчанию:

Study s3[5]; // вызов конструктора по умолчанию для каждого объекта

Отдельно стоит упомянуть о конструкторе копирования. Конструктор копирования и оператор присваивания (=) выполняют почти одинаковую работу. Оба копируют значения из одного объекта в значения другого объекта. Однако конструктор копирования используется при инициализации новых объектов, тогда как оператор присваивания заменяет содержимое уже существующих объектов. Конструктор копирования вызывается компилятором автоматически при инициализации нового объекта уже существующим и при передаче объекта в функцию по значению. О необходимости написания конструктора копирования программистом подробно написано в Учебном пособии “Технология программирования. Введение в ООП” [].

Конструктор копирования для класса Study будет выглядеть следующим образом:

```
Study::Study(const Study& src){
 data = src.data;
 for (int i = 0; i < 10; i++)
        str[i] = src.str[i];
}
```

Здесь в конструктор копирования передается константный аргумент. Данный факт сообщает о том, что в конструкторе копирования не будет изменений, связанных с объектом, передающимся как аргумент.

3.3. Деструктор класса

Задача деструктора – освобождение ресурсов, выделенных пользователями в процессе работы, то есть функция деструктора обратна функции конструктора. К примеру, деструктор может обнулять поля класса и освобождать динамически выделенную память. Деструктор имеет то же имя, что и класс, но перед именем должен стоять знак ‘~’.

Ниже приведен пример написания деструктора для класса `Study`.

```
Study::~Study(){
    data = 0;
    for (int i = 0; i < 10; i++)
         str[i] = "z";
}
```

Деструктор, как и конструктор, всегда присутствует у класса, даже, если программист не описал его явно. Деструктор всегда у класса только один, так как он занимается освобождением памяти и обнулением параметров. Перегрузка деструкторов недопустима по той же причине.

При создании классов, содержащих виртуальные и чистые виртуальные функции, требуется создание виртуального деструктора. Подробно такой вид деструкторов описан в разделе 5.4.

3.4. Статические переменные и методы класса

Поля класса и методы могут являться статическими (статические переменные сохраняют свои значения и не уничтожаются даже после выхода из блока, в котором они объявлены). Пример статической переменной:

```
int counterI() {
  static int i = 0;
  return ++i;
}
int main() {
  cout << counterI() << '\n';
  cout << counterI() << '\n';
  cout << counterI() << '\n';
  return 0;
}
```

Вывод на экран консоли будет следующий: 1 2 3.

Такой вывод на экран связан с тем, что переменная i не обнулилась при выходе из функции `counterI`(), а сохранила свое значение.

Приведем пример программы с классом:

```
class Study {
public:
    int data = -1;
}
int main() {
    Study  s1;
    Study  s2;
    s1.data = 2;
    cout << s1.data << '\n';
    cout << s2.data << '\n';
    return 0;
}
```

При создании объекта класса, каждый из них получает свою собственную копию всех переменных класса. Поскольку создано два объекта класса A, то будет две копии переменной a: s1.data и s2.data, причем s1.data отличается от s2.data. Следовательно, результат программы: -1 2. При добавлении ключевого слова static для переменной data, она становится общей для всех объектов.

```
class Study  {
public:
    static int data;
}
int Study ::data = -1; // определяем статическую переменную класса
int main() {
    Study  s1;
    Study  s2;
    s1.a = 4;
    cout << s1.a << '\n';
    cout << s2.a << '\n';
    return 0;
}
```

В этом случае результат будет: -1 -1. Переменная s1.data - это переменная s2.data.

Статические переменные существуют даже если объекты не созданы. Они создаются при запуске программы и уничтожаются, когда программа завершает свое выполнение. То есть они являются частью класса, а не конкретного объекта. Поэтому чтобы определять статические переменные-члены класса необходимо определить статический член вне класса, в глобальной области видимости, как это сделано в примере. Если этого не сделать, то переменная будет равна 0.

Если класс определен в файле .h, то определение статического члена обычно помещается в файл с кодом класса (например, Study.cpp). Если класс определен в файле .cpp, то определение статического члена обычно пишется непосредственно под классом. Не пишите определение

статического члена класса в заголовочном файле (подобно глобальным переменным). Если этот заголовочный файл подключают больше одного раза, то вы получите несколько определений этого члена, что приведет к ошибке компиляции.

Если статическая переменная указана в области private, то доступа к ней не будет, хотя она и будет существовать с самого запуска программы независимо от существования объекта класса. Чтобы получить доступ к такой переменной необходимо создать статический метод класса. Подобно статическим переменным класса статический метод не привязан к объекту класса. Поскольку статические методы не привязаны к определенному объекту, то их можно вызывать напрямую через имя класса и оператор разрешения области видимости, их также можно вызвать и через объекты класса (но это не рекомендуется).

```
class Study {
private:
  static int data;
public:
  static int getData() { return data; }
}
int Study::data = -1; // определяем статическую переменную класса
int main() {
    cout << Study::getData() << '\n';
    return 0;
}
```

У статического метода нет скрытого указателя this, это следует из того, что статический метод не привязан к конкретному объекту класса. Статические методы могут напрямую обращаться к другим статическим членам (переменным или функциям), но не могут к не статическим членам. Это связано с тем, что нестатические члены принадлежат объекту класса, а статические методы — нет.

3.5. Наследование классов

Наследование классов – очень мощная возможность в объектно-ориентированном программировании. Оно позволяет создавать производные классы (классы-наследники), взяв за основу все методы и элементы базового класса (класса-родителя). Классы наследуют свойства и поведение базовых классов и приобретают дополнительно новые качества.

Объекты производного класса свободно могут использовать всё, что создано и отлажено в базовом классе. При этом мы можем в производный класс дописать необходимый код для усовершенствования программы: добавить новые элементы, методы и т.д. Базовый класс останется нетронутым.

Простой пример базового класса:

```
class Parent{
private:
    int a;
protected:
    int b;
public:
    int c;
    Parent ();
    ~Parent();
};
```

Описано три модификатора доступа в классе: `public`, `protected`, `private`. Для наследования они имеют следующий доступ:

- private (закрытый) – имеют доступ только методы и данные класса, в котором описаны, и дружественные к классу элементы.
- protected (защищенный) – имеют доступ только методы данного класса, классов-наследников и дружественные к классу элементы.
- public (открытый) – имеют доступ все.

Синтаксис написания наследников базового класса:

```
class <ИМЯ_КЛАССА_НАСЛЕДНИКА> : <public / protected / private> <ИМЯ_БАЗОВОГО_КЛАССА>
```

В зависимости от указанного модификатора доступа при наследовании будет изменяться каждый из существующих модификаторов доступа в базовом классе. Зависимости при наследовании при разных модификаторах доступа отображены в Таблице 1.

Таблица 1 - Зависимость модификатора доступа от области видимости в базовом классе

Модификатор доступа / Область видимости в базовом классе	private	protected	public
private	private	private	private
protected	private	protected	protected
public	private	protected	public

Определим двух наследников для базового класса `Parent`:

```
class Child1 : protected Parent {...};
class Child2 : private Parent {...};
```

Для класса Child1 поля родительского класса Parent, бывшие private, остались private, наследовались уровни доступа protected и public.

Child2 не сможет "напрямую" использовать поле “a”, а его потомки – поля “b” и “c”.

Более подробный пример наследования с разными областями доступа представлен в Учебном пособии “Технология программирования. Введение в ООП” [].

Вызов конструкторов в случае наследования будет следующим: вызов базового конструктора, затем конструктор наследника. То есть создается сначала “основа” для построения конкретного объекта наследника, свойственная всем, а на ее основе уже “достраивается” конкретный объект. Вызов деструкторов всегда обратный вызову конструкторов, за исключением статических объектов.

Список инициализации удобен тем, что он может передавать в конструктор базового класса через конструктор наследника параметры для инициализации полей, которые есть у базового класса. Пример приведен ниже.

```
class Parent{
private:
   int a;
protected:
   int b;
public:
   int c;
   Parent (int data) {a = data; b = 0; c = 0;};
   ~Parent() {a = 0; b = 0; c = 0;};
};
class Child : public Parent {
...
public:
  Child (int data): Parent(data) {...};
 ~Child() {...};
};
void main ( ){
  Child obj(5);
}
```

3.6. Композиция классов

Помимо наследования в ООП определена композиция классов. Классы могут включать в качестве элементов объекты других классов. Например, опишем некоторый класс `Discipline`.

```
class Discipline{
public:
```

```
    Discipline();
    ~Discipline();
    void createTablesheet();
private:
    int hours[10];
    char* teacher[10];
};
```

Тогда в класс `Study` можно добавить объект класса `Discipline`. Чтобы не возникло ошибок распознавания объекта `discipline`, необходимо подключить h-файл класса `Discipline`.

```
#include "discipline.h"
class Study {
public:
    Study();
    ~Study();
    void setdata(int d);
    int getdata();
    void setstr(char* s);
    char* getstr();
private:
    int data;
    char* str[10];
    Discipline discipline;
};
```

Такое включение классов друг в друга называется композицией классов.

При композиции классов порядок вызова конструкторов будет следующим: вызов конструкторов для вложенных объектов в порядке их объявления в классе, последним вызовется конструктор для создаваемого итогового объекта.

Композиция отличается от наследования тем, что класс, в который включается объект другого класса не приобретает его методы. В случае наследования справедлива фраза “все, что характерно базовому классу, характерно и наследнику”. То есть наследник в то же время является и базовым классом. в случае же композиции можно сказать, что один класс обладает параметрами второго класса. То есть не “является”, а ”обладает”. Время жизни одного класса зависит от времени жизни второго класса, так как является его составной частью.

4. ПЕРЕГРУЗКА ОПЕРАТОРОВ

При создании пользовательского типа данных – класса, выполнение сложения двух классов, которые создал программист и которые не являются встроенными, невозможно. Чтобы иметь

возможность сравнивать, присваивать и выполнять ряд других операций программисту необходимо перегружать операторы.

То есть, чтобы выполнить следующий код, компилятору необходимо прописать инструкции, которые ему необходимо выполнить.

```
Study a, b, c;
a = b + c;
if (b < c)
   a = c – b;
```

Существуют правила, которые относятся ко всем перегружаемым операторам:

● для перегрузки операторов нельзя использовать аргументы по умолчанию;

● нельзя изменить число операндов, которое подразумевает оператор (унарная операция остается унарной, бинарная – бинарной);

● нельзя создавать новые операторы с использованием ключевого слова operator, только существующие могут быть перегружены;

● нельзя перегрузить оператор для работы со встроенными типами, только для объектов новых типов. Перегрузка работает только с объектами, тип которых определен пользователем, или в смешанных ситуациях, когда объект пользовательского типа участвует в операции вместе с объектом встроенного типа.

Перегрузка оператора возможна двумя способами: как метод класса или как дружественная функция. При выборе способа перегрузки – через метод или через функцию – можно руководствоваться следующими соображениями:

● Если операция затрагивает несколько операндов, и экземпляр класса выступает в операции наравне с другими операндами, рекомендуется использовать дружественные функции.

● Если же можно выделить операнд, по отношению к которому выполняется операция, то логичнее перегружать операцию при помощи метода класса этого операнда.

4.1. Перегрузка с помощью метода класса или с помощью дружественной функции

В случае перегрузки оператора при помощи метода, левый операнд перегружаемой операции передается через указатель this. Поэтому при перегрузке бинарной операции соответствующий оператор метод класса имеет лишь один входной аргумент – правый операнд. При перегрузке унарной операции как метод входные аргументы не передаются.

Приведем пример перегрузки оператора *=. Метод будет принимать в качестве передаваемого параметра целое число (имя параметра r – от слова right(правый)). Результатом операции будет наш текущий объект класса, значения которого умножены на переданное число.

```
#include <stdlib.h>
#include <stdio.h>
#include <iostream>
using namespace std;
class Vector{
private:
    int* data;
    int size;
public:
    Vector();
    ~Vector();
    int getSize();
    void setSize(int s);
    void show();
    Vector& operator *= (int r);
};
Vector& Vector::operator *= (int r){
    for (int i=0; i<size; i++){
       data[i] *= r;
    }
    return *this;
}
int main (){
    Vector v1;
    v1.setSize(4);
    v1 *= 3;
    v1.show();
}
```

При перегрузке оператора с помощью дружественной функции количество входных параметров соответствует числу операндов перегружаемой операции. Дружественная функция определяется вне области действия класса. так как она не принадлежит ему как метод, но имеет доступ к закрытым полям класса.

Приведем пример перегрузки оператора умножения двух классов с помощью дружественной функции. Результатом перемножения двух векторов будет целочисленное значение.

```
class Vector{
friend int operator* (const Vector &l, const Vector &r);
private:
    int* data;
    int size;
public:
    Vector();
```

```
    ~Vector();
    int getSize();
    void setSize(int s);
    void show();
    Vector& operator *= (int r);
};
int operator* (const Vector &l, const Vector &r){
       int res = 0;
       for (int i=0; i < l.size; ++i)
              res += l.data[i] * r.data[i];
       return res;
}
int main (){
       Vector v2, v3;
       int sum = 0;
       v2.setSize(5);
       v3.setSize(5);
       sum = v2*v3;
       cout << sum;
}
```

Имеется два важных ограничения применительных к дружественным функциям. Первое заключается в том, что производные классы не наследуют дружественных функций. Второе - дружественные функции не могут объявляться с ключевым словом static.

Существуют операторы, которые имеют требования к способу перегрузки:

- Оператор присваивания (=) можно перегрузить только как метод класса, потому что он неразрывно связан с объектом, находящимся слева от "=".
- Специальные операторы ([], (), ->, *, &) обязательно перегружаются как метод класса.
- Запрещены к перегрузке операторы: точка (.), разыменования указателя на член класса (.*), разрешения области видимости (::), тернарный оператор (:?), определения размера переменной (sizeof), оператор указания компилятору (#), динамическая идентификация типа данных (typeid).

4.2. Унарные операторы

Унарный оператор взаимодействует с одним операндом, к которому применен, результат выполнения оператора сохраняется в текущем операнде.

Существует постфиксное и префиксное инкрементирование и декрементирование. Встроенный оператор инкрементирования увеличивает значение на единицу, а декрементирования - уменьшает значение на единицу. Встроенный префиксный оператор увеличивает (уменьшает)

значение на единицу и возвращает значение, а постфиксный сначала возвращает значение, потом его увеличивает (уменьшает).

```
int i = 2;
int j = ++i; //i = 3; j = 3
int k = i++; // k = 3; i = 4
```

Операторы могут быть перегружены как методы и как дружественные функции. Для того, чтобы компилятор распознавал префиксную и постфиксную формы инкремента и декремента, введен фиктивный параметр `int` для постфиксной формы.

При перегрузке операторов с помощью метода будет следующий синтаксис написания заголовков:

```
// префикс – возвращает после инкремента
Vector& operator ++ / -- ();
// постфикс – возвращает значение ДО инкремента
Vector operator ++ / -- (int);
```

При перегрузке операторов с помощью дружественных функций синтаксис написания заголовков будет следующий:

```
// префикс – возвращает после инкремента
const Vector& operator ++(--)(Vector& v)
// постфикс – возвращает значение ДО инкремента
const Vector operator ++(--) (Vector &v, int);
```

Приведем пример перегрузки постфиксного оператора индексирования для класса Vector.

```
class Vector{
friend int operator* (const Vector &l, const Vector &r);
private:
    int* data;
    int size;
public:
    Vector();
    ~Vector();
    int getSize();
    void setSize(int s);
    void show();
    Vector& operator *= (int r);
    Vector& operator ++ (int);
};
Vector& Vector::operator ++ (int){
    for (int i=0; i<size; i++){
       data[i] += 2;
    }
```

```
    return *this;
}
int main (){
    Vector v4;
    v4.setSize(6);
    v4++;
    v4.show();
}
```

4.3. Бинарные операторы

Бинарный оператор может быть перегружен для работы, как с другим экземпляром пользовательского класса, так и для работы с числами.

```
Vector operator* (int r);
Vector operator* (Vector r);
```

При создании такой перегрузки, появляется возможность выполнить следующие операции:

```
Vector v5, v6, v7;
v5.setSize(4);
v6.setSize(4);
v6 = v5 * 5;
v7 = v5 * v6;
```

Операции вида +=, -=, *= и /= отличаются от рассмотренных выше операций тем, что они используют левый операнд в качестве входного и выходного параметров. Результатом этих операций является левый операнд после применения к нему соответствующей операции. Такого рода операции логично определять при помощи метода класса. Перегрузка операторов +, -, * и / не дают автоматическую перегрузку операторов +=, -=, *= и /=. Каждый из представленных арифметических операторов требует отдельного описания перегрузки.

Перегрузка бинарных операторов относится и к операторам сравнения: ==, !=, >, <, >=, <=. Результатом операции сравнения является значение типа bool.

```
friend boolean operator==(const Vector& l, const Vector &r);
boolean operator==(const Vector& l, const Vector &r) {
       if (l.size != r.size)
              return false;
       for (int i = 0; i < l.size; ++i)
              if (l.data[i] != r.data[i])
                     return false;
       return true;
}
```

4.4. Перегрузка операторов преобразования типов данных

C++ выполняет неявное преобразование типов данных, которое возможно только для встроенных типов данных:

```
int a = 5;
float b = a; // значение int конвертируется в значение float
```

Для выполнения аналогичного преобразования для пользовательского типа данных требуется перегрузка соответствующей операции преобразования типа.

Приведем пример следующий код, содержащий описание класса Roubles:

```
class Roubles {
private:
    int m_roubles;
public:
    Roubles(int roubles = 0)    {
        m_roubles= roubles;
    }
    int getRoubles() { return m_roubles; }
    void setRoubles(int roubles) { m_roubles= roubles; }
};
void printInt(int value){
    cout << value;
}
int main() {
    Roubles roubles(9);
    printInt(roubles.getRoubles()); // выведется 9
    return 0;
}
```

Для работы в main с рублями как с типом int можно использовать постоянно функции set и get, но это не всегда удобно. Проще перегрузить операцию преобразования значений типа Roubles в тип int. Делается это следующим образом:

```
class Roubles {
private:
    int m_roubles;
public:
    Frank (int roubles=0) {
        m_roubles = roubles;
    }
    operator int() { return m_roubles; }
    int getRoubles() { return m_roubles; }
    void setRoubles(int roubles) { m_roubles= roubles; }
};
void printInt(int value) {
```

```
    cout << value;
}
int main() {
    Roubles roubles(9);
    printInt (roubles); // выведется 9
    return 0;
}
```

Таким же образом можно перегрузить конвертацию в любой встроенный тип данных и конвертацию данных между разными пользовательскими типами данных.

5. ВИРТУАЛЬНЫЕ ФУНКЦИИ

5.1. Раннее и позднее связывание

При написании программ с наследованием часто возникает ситуация, когда методы, описанные в базовом классе и классе-наследнике должны исполнять инструкции отличные друг от друга. В этом случае необходимо определять эти методы как виртуальные. Если этого не сделать, то будет присутствовать ранее связывание (или статическое связывание), или иными словами, связывание на этапе компиляции. То есть вся информация, которая отвечает за определение вызова функций, известна на этапе компиляции программы. Явный пример - это перегруженные операторы. Достоинством является более высокая скорость работы программы, а недостаток - отсутствие гибкости программы.

Пример простой программы с ранним связыванием для вывода на экран.

```
class A {
private:
  int x;
public:
void show(){ cout << “X = ” << x; }
};
class B : public A {
private:
  int y;
public:
void show(){ cout << “Y = ” << y; }
};
void main(){
  A aa;
  B b;
  a.show(); // будет вызван метод базового класса
  b.show(); // будет вызван метод базового класса
}
```

Ниже приведен пример создания того же метода вывода на экран, но с помощью виртуальной функции.

```
class A{
private:
 int x;
public:
 virtual void show(){ cout << “X = ” << x; }
};
class B : public A {
private:
 int y;
public:
 void show () {cout << “Y = ” << y; }
};
void main(){
 A *a, aa;
 B b;
 aa.show(); // будет вызван метод базового класса
 b.show(); // будет вызван метод класса-наследника
 a = &aa;
 a->show(); // будет вызван метод базового класса
 a = &b;
 a->show(); // будет вызван метод класса-наследника
 b.show(); // будет вызван метод класса-наследника
}
```

При приведенном описании базового класса и наследника будет вызван соответствующий метод. Это связано с поздним связыванием, то есть компилятор будет определять во время исполнения программы, какой объект создан, и к какому методу следует обратиться при вызове метода из main. При позднем связывании (динамическое связывание) вызов виртуального метода перенаправляется из соответствующего класса.

Определение в производном классе виртуальной функции, не совпадающей по типу возвращаемого значения или списку параметров с функцией в базовом классе, приводит к синтаксической ошибке.

Важно отметить, что в массивах указателей на базовый класс можно хранить объекты только полиморфных типов (базовый и все производные).

Когда в базовом классе появляется хотя бы одна виртуальная функция, то для всех наследников создается Таблица виртуальных функций.

5.2. Таблица виртуальных функций

Таблица виртуальных функций - это массив указателей на функции; в массиве столько элементов, сколько виртуальных функций в классе. Это обычный статический массив, который создается компилятором во время компиляции. Для каждого класса создается своя таблица, а количество элементов в них одинаковое. Виртуальная таблица содержит по одной записи на каждую виртуальную функцию, которая может быть вызвана объектами класса. Каждая запись в этой таблице — это указатель на функцию, указывающий на наиболее дочерний метод, доступный объекту этого класса. Компилятор также добавляет скрытый указатель на родительский класс, который мы будем называть *__vptr. Этот указатель автоматически создается при создании объекта класса и указывает на виртуальную таблицу этого класса. В отличие от скрытого указателя *this, который фактически является параметром функции, используемый компилятором для «указания на самого себя», *__vptr является реальным указателем. Следовательно, размер каждого объекта увеличивается на размер этого указателя. *__vptr также наследуется дочерними классами.

Для всех наследных классов таблицы будут содержать разные значения, будут записаны адреса методов текущего класса. Это таблица для поиска функций для выполнения вызовов функций в режиме позднего (динамического) связывания. Виртуальную таблицу еще называют «vtable».

Приведем пример для трех классов, использующих виртуальные функции show() и showMe().

```
class A{
private:
  int x;
public:
  virtual void show(){ cout << “X = ” << x; }
  virtual void showMe() { cout << “A!” << endl; }
};
class B : public A {
private:
  int y;
public:
  void show () {cout << “Y = ” << y; }
};
class C : public A {
private:
  int z;
public:
  void showMe() { cout << “C!” << endl; }
};
```

В представленных классах A, B, C описаны 2 виртуальные функции поэтому каждая таблица виртуальных функций будет содержать две записи (Рисунок 22).

Объект класса A имеет доступ только к членам класса A, он не имеет доступа к членам классов B и C. Поэтому запись таблицы виртуальных функций show будет указывать на A::show(), а запись showMe будет указывать на A::showMe().

Объект класса B имеет доступ как к собственным методам, так и к методам базового класса. Класс B имеет переопределение метода show, но не имеет переопределения showMe. Поэтому запись show будет указывать на B::show(), а showMe на A::showMe().

Аналогично классу B выглядит класс C. Метод show в классе C не определен, поэтому соответствующая запись будет указывать на A::show().

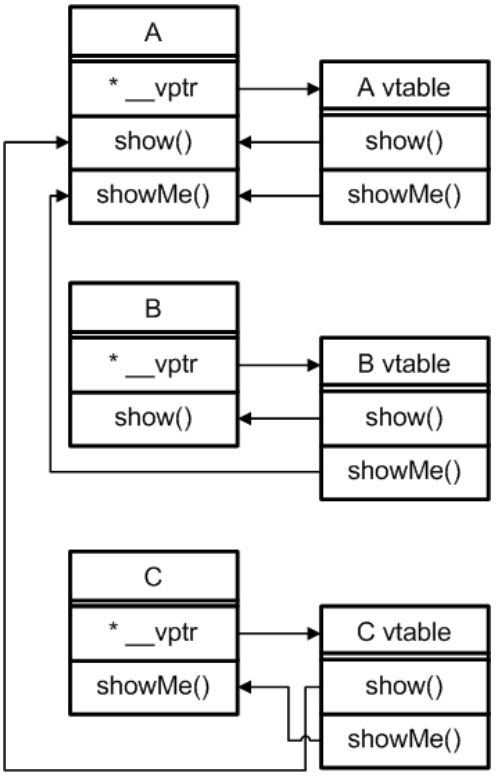


Рисунок 22 - Представление таблиц виртуальных функций для классов A, B и C

Согласно представленным таблицам и будут осуществляться вызовы функций. То есть при вызове c.show() будет вызвана функция A::show().

5.3. Чистая виртуальная функция и абстрактный базовый класс

Бывает так, что в программе нет необходимости использования объектов базового класса, то есть базовый класс необходим только для построения иерархии классов и определения общих переменных и методов. Такие классы можно делать абстрактными. Абстрактный класс содержит

чистые виртуальные функции, которые обязаны определяться в производных классах. Благодаря полиморфизму этот интерфейс класса будет доступен всем функциям в иерархии. Объекты абстрактного класса создать нельзя.

Важное свойство абстрактных классов при наследовании заключается в том, что если производный класс от базового абстрактного класса не определяет чистую виртуальную функцию, то она остается чистой и в производном классе, то есть производный класс сам становится абстрактным.

Чистая виртуальная функция – это функция, объявление которой завершается инициализатором “= 0”. Пример объявления такой функции приведен ниже.

```
virtual void F() = 0;
```

Теперь приведем пример написания абстрактного базового класса и одного наследника.

```
class Figure {
public:
    virtual void print() = 0;
    virtual void showFigureType() = 0;
};
class Rectangle : public Figure {
private:
...
public:
...
    void print(){...}
    void showFigureType(){
      std::cout << "Rectangle" << std::endl;
       }
};
void main(){
   Figure f; // ошибка!
   Figure *f; // такое использование абстрактного класса возможно
   Rectangle r;
   f = &r;
   f-> showFigureType();
}
```

Пример демонстрирует возможность использования абстрактного базового класса в main, а именно - использование указателя на базовый класс для работы с наследниками. Это может быть удобно, когда существует много наследников и создаются массивы.

5.4. Виртуальный деструктор

Деструктор абстрактного базового класса должен объявляться виртуальным, потому что только это обеспечивает корректное разрушение объекта производного класса через указатель.

Приведем пример использования виртуального деструктора.

```
class C {
public:
   C() { cout << “C created!” << endl; }
   ~C() { cout << “C deleted!” << endl; }
};
class A {
public:
   A() { cout << “A created!” << endl; }
   virtual ~A(){ cout << “A deleted!” << endl; }
};
class B : public A{
   B(){ cout << “B created!” << endl; }
   ~B() { cout << “B deleted!” << endl; }
   C cc;
};
void main(){
   A *a = new B;
   delete a;
}
```

В приведенном примере в функции main указателю базового класса присваивается динамически созданный объект наследника. Потом этот объект разрушается. Наличие виртуального деструктора в базовом классе обеспечивает вызовы всех деструкторов в верном порядке, а именно:

```
B deleted!
C deleted!
A deleted!
```

Если же деструктор не является виртуальным, то будет разрушена только та часть объекта, которая относится к базовому классу. И вывод программы будет уже другим.

```
A deleted!
```

5.5. Модификаторы override и final

В C++ 11 добавлены два новых модификатора: override и final.

Модификатор override появился в силу того, что разработчики упускают из внимания синтаксис перегружаемых функций. Переопределяемая функция должна иметь тот же возвращаемый тип данных и список параметров, что и виртуальная функция базового класса.

Например:

```
class A{
private:
  int x;
public:
```

```
 virtual void show1(){ cout << “X = ” << x; }
 virtual void show2(){ cout << “2*X = ” << 2*x; }
};
class B : public A {
private:
 int y;
public:
 void show1(int y) {cout << “Y = ” << y; }
 void show2() const { cout << “2*Y = ” << 2*y; }
};
```

В примере хорошо видно, что перегружаемые функции show1 и show2, описанные в базовом классе и классе - наследнике, отличаются друг от друга. Поэтому при работе main будет выведено на экран только “X = ” и “2*X = ”. Модификатор override позволяет решать данную проблему. Модификатор override может использоваться с любым методом, который должен быть переопределен. Если метод не совпадает ни с одной виртуальной функцией родительского класса, то компилятор выдаст ошибку.

```
class A{
private:
 int x;
public:
 virtual void show1(){ cout << “X = ” << x; }
 virtual void show2(){ cout << “2*X = ” << 2*x; }
};
class B : public A {
private:
 int y;
public:
 void show1(int y) override {cout << “Y = ” << y; } // ошибка компиляции
 void show2() const override { cout << “2*Y = ” << 2*y; } // ошибка компиляции
};
```

Ошибки возникнут, так как ни один из методов не является переопределением методов базового класса. Чтобы не возникло ошибок необходимо следующее написание исходного кода:

```
class A{
private:
 int x;
public:
 virtual void show1(){ cout << “X = ” << x; }
 virtual void show2(){ cout << “2*X = ” << 2*x; }
};
class B : public A {
private:
 int y;
```

```
public:
 void show1() override {cout << “Y = ” << y; }
 void show2() override { cout << “2*Y = ” << 2*y; }
};
```

Использование модификатора override никак не влияет на эффективность или производительность программы, но помогает избежать непреднамеренных ошибок. Следовательно, настоятельно рекомендуется использовать модификатор override для каждого из своих переопределений.

Модификатор final служит для запрещения переопределения виртуальных функций или запрещения наследования класса. Указывается модификатор final на той же позиции, что и модификатор override.

```
class A{
private:
 int x;
public:
 virtual void show1(){ cout << “X = ” << x; }
 virtual void show2(){ cout << “2*X = ” << 2*x; }
};
class B : public A {
private:
 int y;
public:
 void show1() override {cout << “Y = ” << y; }
 void show2() override final { cout << “2*Y = ” << 2*y; }
};
class C : public B {
private:
 int z;
public:
 void show1() override {cout << “Z = ” << y; }
 void show2() override { cout << “2*Z = ” << 2*z; } // ошибка компиляции
};
```

В классе B указано, что дальнейшие действия по перегрузке функции show2 запрещены, поэтому при предложенном описании класса C будет выведена ошибка при компиляции. С переопределением функции show1 никаких проблем не возникнет, переопределение не запрещено и может быть описано.

В случае, если мы хотим запретить наследование определенного класса, то модификатор final указывается после имени класса:

```
class A{
private:
 int x;
```

```
};
class B final : public A {
private:
 int y;
};
class C : public B { // ошибка компиляции
private:
 int z;
};
```

При компиляции будет выведена ошибка, так как наследовать класс B нельзя.

6. ШАБЛОНЫ

Шаблон - это, по сути, инструкция, по которой создается локальная версия для определенного набора параметров и типов данных. Могут быть шаблонные функции и шаблонные классы.

6.1. Шаблоны функций

Шаблоны функций отлично справляются с однотипными задачами для разных типов данных. Например, чтобы вывести на экран числа, которые хранятся в массиве для разных типов данных потребуется написание как минимум четырех функций (для типов int, float, double, char), алгоритм которых идентичен, но вывод на экран будет разных типов значений. Шаблон функции вывода на экран решает данную задачу написания четырех идентичных функций. Создается шаблон функции, в котором описываются все типы данных. Это позволяет освободить исходный код от лишних строк.

Все шаблоны функций начинаются с ключевого слова template, после него идут угловые скобки, в которых определяется список параметров. Перед каждым параметром предшествует зарезервированное слово class или typename.

`template <class T>` или `template <typename T>`

Вместо буквы T можно использовать любую другую. По сути T - это зарезервированное место для любого типа данных.

`template <class R>` или `template <typename T1, typename T2>`

В качестве параметра может быть использован как встроенный тип данных, так и пользовательский (класс или структура).

При вызове шаблонной функции компилятор анализирует параметр, передаваемый в функцию и создает экземпляр функции для соответствующего типа данных. Компилятор сам создает временные копии функций.

Приведем пример шаблона функции для перемены местами двух значений.

```
template <typename T>
void mySw (T& left, T& right){
   T temp(left);
   left = right;
   right = temp;
}
int main(){
   int a = 3;
   int b = 5;
   mySw(a,b);
   float aa = 11.11;
   float bb = 22.22;
   mySw(a,b);
}
```

В примере продемонстрировано использование шаблонной функции в main. Компилятор видит, что пользователь передает сначала в функцию int, а затем float, поэтому будет создано два экземпляра функции для соответствующих типов данных.

Приведем пример использования нескольких типов с шаблонами функций для вывода на экран:

```
template <typename T1, typename T2>
void show(T1 data1, T2 data2){
   cout << “First = ” << data1 << endl;
   cout << “Second = ” << data2 << endl;
}
```

6.2. Шаблоны классов

Применение шаблона класса разумно при тех же условиях, что и применение шаблона функции, за исключением того, что речь идет не о выполнении рядовой операции, а более сложное взаимодействие с данными или совокупностью данных, которые необходимо организовывать как класс. То есть шаблон класса имеет универсальное поведение, допускающее повторное использование с другим типом данных.

Как объявление, так и реализация шаблона класса содержит угловые скобки, в первом случае после ключевого слова template, в которых перечисляются параметры шаблона. Это список не может быть пустым. Каждому параметру шаблона должно предшествовать ключевое слово typename или class. Приведем пример шаблона класса.

```
template <typename T>
class My {
public:
   T sum (T a, T b){
```

```
            return (a+b);
        }
        T del (T a, T b){
            return (a/b);
        }
};
int main(){
        My<double> m1;
        My<int> m2;
        cout << m1.sum(1.2, 5.5) << endl;
        cout << m2.del(30, 5) << endl;
}
```

Во время выполнения программы параметр T будут подставлены double и int, определенные пользователем в main. Для каждого типа данных будет автоматически создана копия класса с указанным типом данных. Такой процесс называется конкретизацией шаблона.

Ошибочными будут следующие объявления шаблонов класса:

● Переменная класса или метод не может быть одноименным с его параметром. Пример такой программы приведен ниже:

```
template <class T>
class My{
  …
  private:
  double T;
  T data;
};
```

● Имя параметра шаблона может быть указано в списке только один раз.

```
template <class T, class T>
class My{...};
```

Шаблоны классов позволяют пользоваться аргументами по умолчанию и использовать более одного типа параметров, например:

```
template <typename T = int, int size = 10>
class My{
public:
   T* arr;
   My(){
      arr = new T [size];
      memset(arr, 0, size * sizeof(T));
   }
}
```

В примере выше класс параметризован типом T и целым (константным, поскольку раскрытие шаблона производится на этапе компиляции) size. Поскольку size является константой, указанный шаблонный класс может быть переписан без использования динамической памяти:

```
template <typename T = int, int size = 10>
class My{
public:
   T arr[size];
}

int main() {
  My x;              // размер переменной x = sizeof(My) = sizeof(int) * sizeof(10)
  My<int, 10> y = x; // типы x и y одинаковы
  My<int, 15> z;     // z = x - ошибка компиляции, несовместимые типы
}
```

ПРИЛОЖЕНИЕ. Структуры данных

Очередь и циклическая очередь

Очередь – это упорядоченный набор элементов, которые могут быть извлечены только с её начала, а добавлять новые элементы можно только в конец. Очередь организована согласно дисциплине FIFO – First In First Out: первый пришедший первым обслужен.

С очередью можно производить следующие операции:

- инициализировать очередь (initialization);
- добавить элемент в конец очереди (push);
- извлечь первый элемент из очереди (pop);
- проверить очередь на пустоту (isEmpty).

Доступа к произвольному элементу в очереди нет, пользователь может работать с очередью с помощью специальных методов.

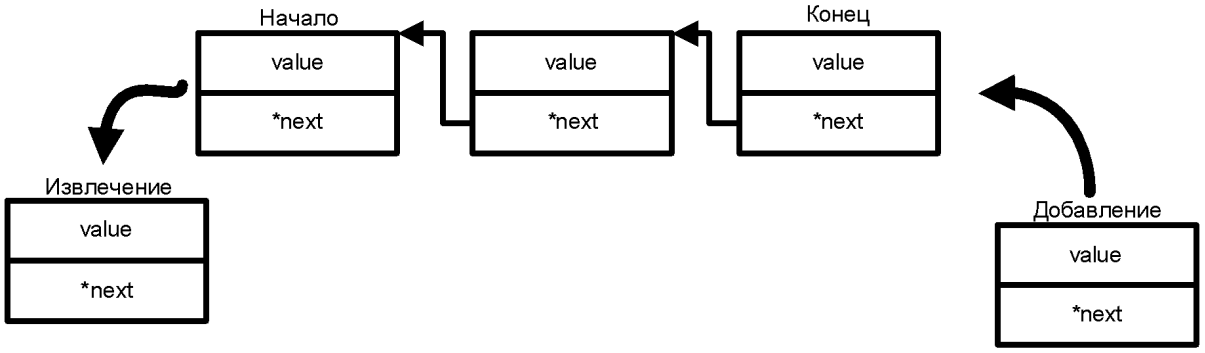


Рисунок 1 - Очередь

Циклическая очередь отличается тем, что:

- очередь фиксированного размера N;
- существует индекс головы (head);
- существует индекс конца (tail), который указывает на первую свободную ячейку.

Идея циклической очереди состоит в том, что очередь представляется в виде кольца, то есть за последним элементом очереди следует его первый элемент. Иными словами, получается непрерывная очередь, в которой от конца переходим к началу.

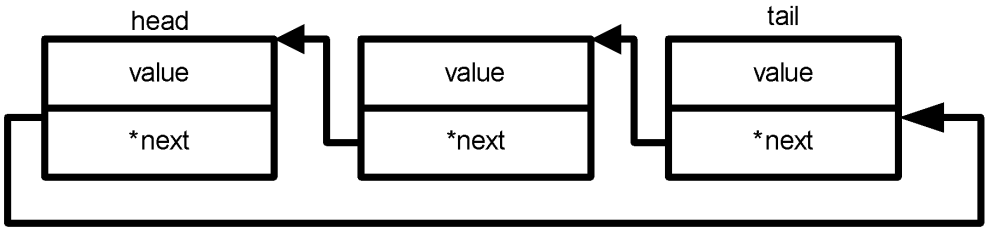


Рисунок 2 – Циклическая очередь

При добавлении нового элемента необходимо сначала определить, если ли свободное место в очереди. При извлечении элемента необходимо проверять очередь на пустоту и извлечение последнего элемента.

Также существует двусторонняя очередь (или дэк), которая расширяет поведение обычной очереди. В дек можно извлекать и добавлять элементы, как с начала, так и с конца очереди.

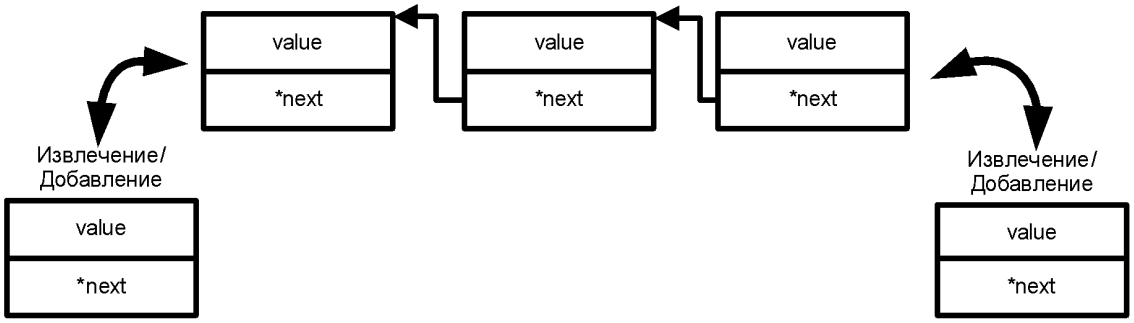


Рисунок 3 – Дэк

2. Стек

Стек – это упорядоченный набор элементов, в котором размещение и извлечение происходят с одного конца. Стек организован согласно дисциплине обслуживания LIFO – Last In First Out: последний пришедший первым обслужен.

Со стеком можно производить следующие операции:

- инициализировать стек (initialization);
- добавить элемент в стек (push);
- извлечь элемент из стека (pop);
- проверить стек на пустоту (isEmpty);
- узнать сколько элементов в стеке (length).

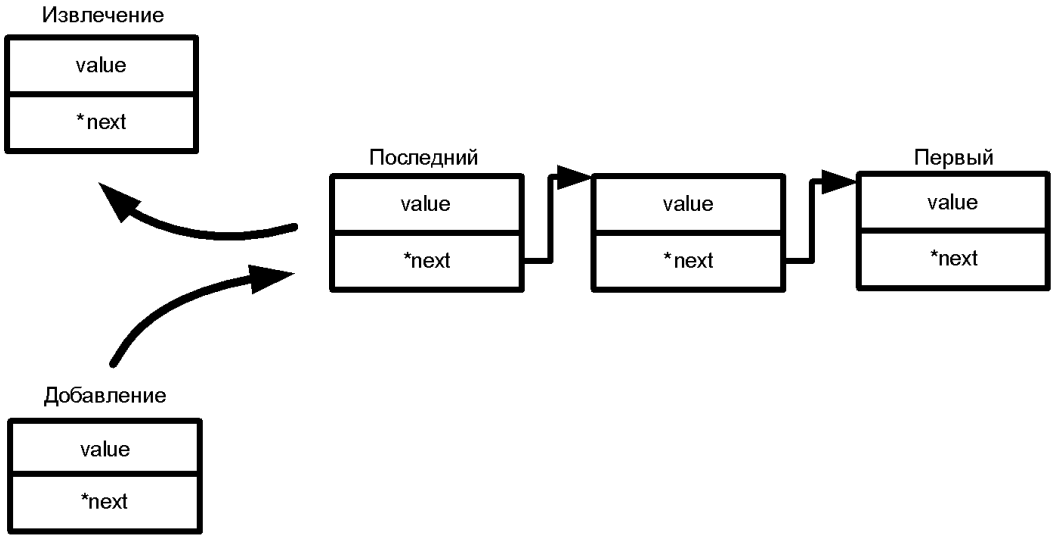


Рисунок 4 – Стек

Доступа к произвольному элементу в стеке нет, пользователь может работать со стеком с помощью специальных методов.

3. Список

Список – это самый простой тип данных динамической структуры. Элементы списка можно добавлять и извлекать произвольным образом. Доступ к списку осуществляется с помощью указателя, который содержит адрес первого элемента списка. Такой указатель называется корнем списка.

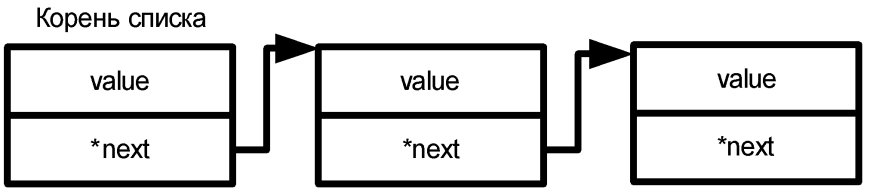


Рисунок 5 – Список

По количеству полей указателей различают однонаправленный (односвязный) и двунаправленный (двусвязный) списки.

Списки еще можно разделить по типу связи элементов на линейные и циклические. В линейном списке последний элемент указывает на NULL. В циклическом списке последний элемент указывает на первый элемент в списке.

Таким образом, получаем несколько видов списков:

- Односвязный линейный список

  Каждый элемент содержит один указатель на следующий элемент, последний указатель содержит NULL.

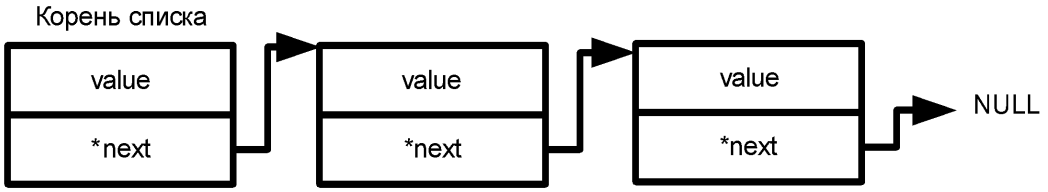


Рисунок 6 – Односвязный линейный список

- Односвязный циклический список

  Каждый элемент содержит одно поле указателя на следующий элемент, последний указатель содержит адрес первого элемента (адрес корня списка).

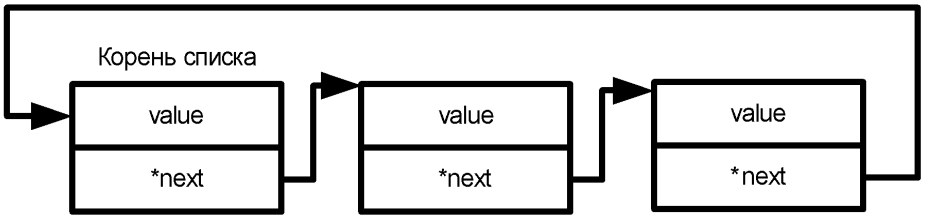


Рисунок 7 – Односвязный циклический список

- Двусвязный линейный список

Каждый элемент содержит указатели на следующий и предыдущий элементы. Поле указателя на следующий элемент последнего элемента содержит NULL. Поле указателя на предыдущий элемент корня списка (первого элемента) содержит NULL.

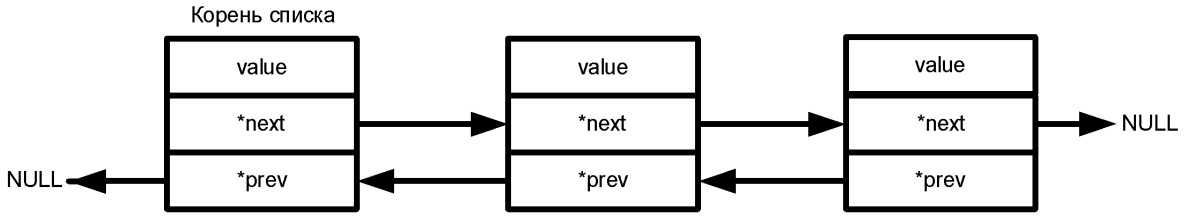


Рисунок 8 – Двусвязный линейный список

- Двусвязный циклический список

Каждый элемент содержит указатель на предыдущий элемент и указатель на следующий элемент. Поле указателя на следующий элемент последнего элемента содержит адрес первого элемента (корня списка). Поле указателя на предыдущий элемент содержит адрес последнего элемента.

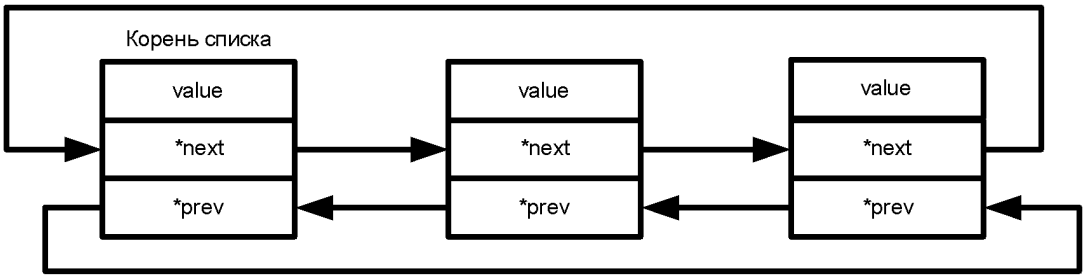


Рисунок 9 – Двусвязный циклический список

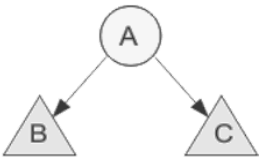
Со списком можно производить следующие операции:

- Инициализация списка: необходима для создания корневого элемента списка, у которого поля указателей содержат нулевой указатель (initialization).

- Добавление элемента в список: для выполнения этой операции необходимо знать индекс элемента списка, после которого происходит добавление, и элемент для добавления (или данные) (push).
- Извлечение элемента: для выполнения этой операции требуется знать индекс извлекаемого элемента, функция возвращает указатель на следующий элемент за удаляемым (pop).
- Извлечение корня списка: необходимо вернуть указатель на новый корень списка (getRoot).
- Вывод на экран элементов списка (showList).
- Обмен данными двух списков.

4. Бинарное дерево

Бинарное дерево - это конечное множество элементов (узлов), которое либо пусто, либо содержит элемент (корень), связанный с двумя различными бинарными деревьями, называемыми левым и правым поддеревьями. Связи между узлами дерева называются его ветвями. Таким образом, узлы хранят данные, указатель на правую ветку и указатель на левую ветку. Основное правило формирования бинарного дерева в C++: если значение узла больше добавляемого – добавляется ветка справа, иначе создается ветка слева. Дерево строится относительно «корня» - пользовательского значения.



Обход дерева в нисходящем порядке: A, B, C. То сначала отображается корень дерева, затем рекурсивный «спуск» по левому поддереву, потом по правому.

Обход дерева в восходящем порядке: B, C, A. То есть сначала рекурсивный «подъем» по левому поддереву, потом по правому и отображение корня дерева.

ПРИЛОЖЕНИЕ. Алгоритмы сортировки

1. Двоичный поиск

Двоичный поиск можно использовать только в том случае, если есть массив, все элементы которого упорядочены (отсортированы).

Шаги работы алгоритма:

Шаг 1. Зона поиска (на первом шаге ей является весь массив) делиться на две равные части, путем определения ее среднего элемента;

Шаг 2. Средний элемент сравнивается с искомым (key), результатом этого сравнения будет один из трех случаев:

- key<mid. Крайней правой границей области поиска становится элемент, стоящий перед средним (right ← mid-1).
- key>mid. Крайней левой границей области поиска становится следующий за средним элемент (left ← mid+1).
- key=mid. Значения среднего и искомого элементов совпадают, следовательно элемент найден, работа алгоритма завершается.

Шаг 3. Если для проверки не осталось ни одного элемента, то алгоритм завершается, иначе выполняется переход к пункту 1.

Приведем пример работы алгоритма. Предположим у нас есть отсортированный массив: 1 2 3 4 5 6 7 8 9 0. Пользователь указывает число, которое ему следует найти: 7. Сначала размер массива делится на 2: 10/2 = 5. Проверяем элемент массива, который расположен на этом индексе: [5] = 6. 6!=7. Искомый элемент больше элемента, расположенного посередине, поэтому поиск не стоит проводить в левой половине массива, так как там расположены числа меньше 6. Поиск сместится в правую часть исходного массива, выделяется подмассив:6 7 8 9 0. Размер делится на 2: 5/2=2. [7]=8. 8!=7. Поиск смещается в левую часть подмассива: 6 7 8. Размер делится на 2: 3/2 = 1. [6] = 7. 7==7. Таким образом, найден исходный элемент, его индекс [6].

2. Сортировка выбором

Алгоритм сортировки выбором находит в исходном массиве максимальный или минимальный элементы, в зависимости от того как необходимо отсортировать массив, по возрастанию или по убыванию. Если массив должен быть отсортирован по возрастанию, то из исходного массива необходимо выбирать минимальные элементы. Если же массив необходимо отсортировать по убыванию, то выбирать следует максимальные элементы.

Приведем пример сортировки массива по возрастанию. В исходном массиве находим минимальный элемент, меняем его местами с первым элементом массива. В оставшейся части массива опять ищем минимальный элемент. Найденный минимальный элемент меняем местами со вторым элементом массива и т. д. Таким образом, суть алгоритма сортировки выбором сводится к многократному поиску минимального (максимального) элементов в неотсортированной части массива.

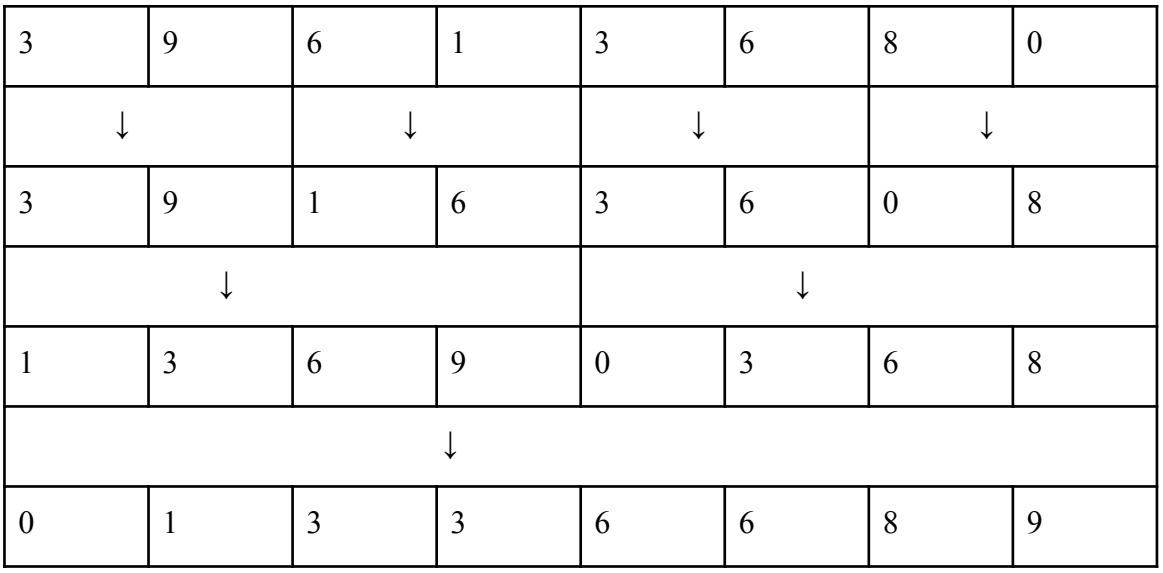
Например, дан массив 5 2 2 7 8 4 5 0. Находим минимальный элемент в массиве, это 0. Меняем местами минимальный и первый элемент. Результат: 0 2 2 7 8 4 5 5. Находим минимальный элемент в неотсортированной части массива: 2. Менять местами не надо. Следующий элемент снова 2. Затем – 4, меняем местами с 7; результирующий массив: 0 2 2 4 8 7 5 5. И т.д. в итоге получаем отсортированный массив: 0 2 2 4 5 5 7 8.

3. Сортировка слиянием

Список разделяется на равные или практически равные части, каждая из которых сортируется отдельно. После чего уже упорядоченные части сливаются воедино.

Массив рекурсивно разбивается пополам, и каждая из половин делиться до тех пор, пока размер очередного подмассива не станет равным единице. Далее выполняется операция алгоритма, называемая слиянием. Два единичных массива сливаются в общий результирующий массив, при этом из каждого выбирается меньший элемент (сортировка по возрастанию) и записывается в свободную левую ячейку результирующего массива. После чего из двух результирующих массивов собирается третий общий отсортированный массив, и так далее. В случае если один из массивов закончиться, элементы другого дописываются в собираемый массив; В конце операции слияния, элементы перезаписываются из результирующего массива в исходный.

Приведем пример для сортировки массива чисел: 3 9 6 1 3 6 8 0.



4. Сортировка вставками

В начале сортировки первый элемент массива считается отсортированным, все остальные — не отсортированные. Начиная со второго элемента массива и заканчивая последним, алгоритм вставляет неотсортированный элемент массива в нужную позицию в отсортированной части массива. Таким образом, за один шаг сортировки отсортированная часть массива увеличивается на один элемент, а неотсортированная часть массива уменьшается на один элемент. На каждом шаге сортировки сравнивается текущий элемент со всеми элементами в отсортированной части.

Приведем пример для массива 4 1 4 5 9 0.

Шаг	Отсортированная часть массива	Текущий элемент
1	4	1
2	1 4	4
3	1 4 4	5
4	1 4 4 5	9
5	1 4 4 5 9	0
6	0 1 4 4 5 9	

5. Метод Шелла

Метод построен на основе метода вставки с минимизацией промежуточных шагов. Общая схема метода состоит в следующем.

Шаг 1. Происходит упорядочивание элементов n/2 пар $(x_i,x_{n/2+i})$ для $1 < i < n/2$.

Шаг 2. Упорядочиваются элементы в n/4 группах из четырех элементов $(x_i,x_{n/4+i},x_{n/2+i},x_{3n/4+i})$ для $1 < i < n/4$.

Шаг 3. Упорядочиваются элементы уже в n/4 группах из восьми элементов и т.д.

На последнем шаге упорядочиваются элементы сразу во всем массиве x1,x2,...,xn. На каждом шаге для упорядочивания элементов в группах используется метод сортировки вставками.

Пример сортировки Шелла. Исходный массив чисел:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
5	3	8	0	7	4	9	1	6	2

Шаг 1. 10/2 = 5. Числа расположены на расстоянии 5 друг от друга. Список пар следующий: (5,4), (3,9), (8,1), (0,6),(7,2). Отсортируем внутри пары по возрастанию и расставим в исходном массиве.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
4	3	1	0	2	5	9	8	6	7

Шаг 2. 5/2=2. Числа расположены на расстоянии 2 друг от друга. Отсортируем внутри пары по возрастанию и расставим в исходном массиве.

Выполняем сортировку последовательно. Пара (4,1):

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
1	3	4	0	2	5	9	8	6	7

Пара (3,0):

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
1	0	4	3	2	5	9	8	6	7

Пара (4,2):

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
1	0	2	3	4	5	9	8	6	7

Пара (3,5):

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
1	0	2	3	4	5	9	8	6	7

Пара (4,9):

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
1	0	2	3	4	5	9	8	6	7

Пара (5,8):

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
1	0	2	3	4	5	9	8	6	7

Пара (9,6):

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
1	0	2	3	4	5	6	8	9	7

Пара (8,7):

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
1	0	2	3	4	5	6	7	9	8

Шаг 3. 2/2 = 1. Числа расположены на расстоянии 2 друг от друга. Отсортируем внутри пары по возрастанию и расставим в исходном массиве.

Выполняем сортировку последовательно как на предыдущем шаге. Результат сортировки приведен ниже.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
0	1	2	3	4	5	6	7	8	9

ПРИЛОЖЕНИЕ. Лабораторные работы

1. Общие вопросы

Для ускорения процесса сдачи лабораторных работ (ЛР), студент может присылать ссылки на github преподавателю в личный кабинет pro.guap.ru. Преподаватель ознакомится с программой и выскажет пожелания к изменению кода программы, если это требуется. Если программа отвечает минимуму, необходимому для защиты (это не обязательно будет максимальное количество баллов), то будет разрешено оформление отчета.

Отчет оформляется после одобрения программы преподавателем. Отчет отправляется на pro.guap.ru на проверку и только после одобрения может быть напечатан.

Отчет является одной из частей каждой ЛР. Грамотное написание отчета - это первая тренировка студентов перед написанием дипломной работы. Грамотное изложение информации в письменном виде - полезный навык, который легко начать тренировать на отчетах по текущему предмету.

Каждую ЛР необходимо защищать преподавателю для получения оценки. Программа при демонстрации не должна завершаться ошибкой при вводе как корректных, так и некорректных данных, создаваемые сущности должны функционировать согласно общеизвестным правилам (например, всего 24 часа 59 минут и 59 секунд, не может быть времени 25:100:79).

Термин “пользовательское число” или “пользовательские данные” - это данные, которые пользователь программы вводит с клавиатуры при работе с программой. В программе не должно

существовать каких-либо константных или заданных программистом заранее переменных, все данные вводятся пользователем с клавиатуры.

Полный набор функций определяет студент самостоятельно, в задании может быть перечислена только часть. Полнота взаимодействия пользователя с программой также оценивается. При возникающих вопросах при написании ЛР обращайтесь к преподавателю через pro.guap.ru.

В каждой ЛР необходимо помнить и не нарушать инкапсуляцию, в том числе при наследовании. Данные обязаны располагаться в области private, и не быть доступными пользователю. Для работы с данными должны быть определены set/get функции для каждого параметра.

Оценки в конце семестра выставляются тем студентам, которые предоставили все отчеты по всем ЛР и сдали все ЛР до начала экзаменационной сессии.

2. Содержание отчета

- Титульный лист с указанием темы лабораторной работы

Актуальное оформление титульного листа располагается по адресу: http://guap.ru/guap/standart/titl_main.shtml

- Постановка задачи

Необходимо написать полное задание, которое требуется выполнить в лабораторной работе.

- Формализация задачи

Формализация(от лат. forma — вид, образ) — отображение результатов мышления в точных понятиях и утверждениях. При формализации изучаемым объектам, их свойствам и отношениям ставятся в соответствие некоторые устойчивые, хорошо обозримые и отождествляемые конструкции, дающие возможность выявить и зафиксировать существенные стороны объектов.

- Исходный код

Привести полный исходный код, который полностью выполняет требования и задание лабораторной работы.

- Результаты работы программы

Привести примеры работы программы с описанием действий пользователя и описанием того, что программа выводит на экран.

- Выводы

Сделать выводы о проделанной работе и изученном материале.

3. Требования к защите

При защите ЛР преподавателю студент защищает текущую программу с отчетом

При оценке преподавателем ЛР будут учитываться следующие показатели:

1. Листинг программы и полнота реализации задания.
2. Ответы по листингу программы. Обоснованность написания алгоритма и исходного кода при решении задачи.
3. Ответы на теоретические вопросы.

Для защиты лабораторной работы необходимо продемонстрировать работу программы, ответить на вопросы и предоставить корректно оформленный отчет. Лабораторная работа считается успешно сданной при подписанном отчете преподавателем и студентом, и выложенном отчете в личном кабинете студента на сайте ГУАП и подтвержденным преподавателем.

Максимальное число баллов за лабораторную работу - 15. Но студент может получить меньшее число баллов, в зависимости от его ответов и представленной программы.

4. Задания

4.1. Лабораторная работа №1

Создать родительский класс "Очередь" с функциями инициализации очереди, добавления элемента в очередь и извлечения элемента из очереди.

Создать метод создания копии очереди. Результатом должен стать новый экземпляр класса “Очередь”, состоящий из элементов (копий элементов) исходной очереди. Порядок следования элементов должен быть сохранен.

Создать функцию слияния двух очередей. Результатом должна быть третья очередь, состоящая элементов первой очереди и второй очереди. Порядок следования элементов должен быть сохранен.

На основе родительского класса "Очередь" создать три дочерних класса "Очередь1" (по каждому типу наследования: public, protected и private) с функциями нахождения и отображения на экране требуемого в соответствии с вариантом задания значения (см. таблицу ниже).

В main необходимо работать только с классами-наследниками (должна быть возможность выбора для пользователя с каким наследником необходимо работать), базовый класс не должен быть объявлен в main и использоваться.

Внимательно ознакомьтесь с разделом 1 текущего Приложения. Обязательно к выполнению:

- Необходимо разделение на h и cpp файлы для каждого класса. Функция main должна располагаться в отдельном cpp файле.

- Элемент очереди содержит данные и ссылку на предыдущий элемент. Элемент реализовать с помощью класса или структуры.
- Реализовать динамическое выделение памяти для очереди/элемента очереди и динамическое очищение памяти при извлечении элемента очереди.
- Заранее число элементов очереди неизвестно, очередь заполняется постепенно пользователем программы.
- Не принимается очередь, реализованная с помощью STL контейнеров или в виде статического массива.
- В родительском классе обязательно должно быть три модификатора доступа с объявленными в них полями данных и/или методами. Данные очереди обязаны находиться в области доступа private родительского класса.
- Реализовать пользовательское меню, например: 1 – Добавление элемента очереди; 2 – Извлечение элемента очереди; 3 – Вывод очереди на экран; 4 – Выполнение расчета по варианту; 5 – Создание копии очереди; 6 – Слияние двух очередей; 7 – Выход из программы.
- Пользователь с клавиатуры вводит значения.
- По списку студент определяет свой порядковый номер и этому номеру соответствует задание в таблице.

Объяснить разницу работы программы при разных модификаторах доступа при наследовании. Продемонстрировать работу программы.

Таблица 1 - Задания для реализации класса “Очередь1”

Вариант	**Задание для расчета**
1	Подсчет **среднего арифметического**. Формула подсчета среднего арифметического следующая: где N – число элементов в очереди, x_i – i-тый элемент очереди
2	Подсчет **среднего гармонического**. Формула подсчета среднего гармонического следующая: где N – число элементов в очереди, x_i – i-тый элемент очереди

3	Подсчет размаха ряда элементов – разности между максимальным и минимальным .
4	Нахождение минимального элемента
5	Нахождение максимального элемента
6	Вычисление количества нечетных элементов
7	Вычисление количества четных элементов
8	Нахождение последнего элемента, меньшего среднего арифметического
9	Нахождение последнего элемента, меньшего среднего гармонического
10	Подсчет числа элементов, больших среднего арифметического значения
11	Подсчет числа элементов, больших среднего гармонического значения
12	Подсчет числа элементов, больших размаха (разности между максимальным и минимальным элементами)
13	Подсчет числа элементов, значение которых превышает значение предыдущего элемента
14	Подсчет среднего арифметического для элементов с нечетным порядковым номером.
15	Подсчет среднего гармонического для элементов с четным порядковым номером.
16	Подсчет размаха (разности между максимальным и минимальным элементами) для элементов с нечетным порядковым номером.
17	Нахождение первого элемента, большего среднего арифметического
18	Нахождение первого элемента, большего среднего гармонического
19	Нахождение первого элемента, большего размаха (разности между максимальным и минимальным элементами)
20	Подсчет числа четных элементов, больших среднего арифметического
21	Подсчет суммы элементов, меньших среднего арифметического
22	Подсчет суммы элементов, меньших среднего гармонического
23	Подсчет суммы четных элементов
24	Подсчет суммы нечетных элементов
25	Подсчет среднего арифметического четных элементов
26	Подсчет среднего гармонического четных элементов
27	Подсчет размаха (разности между максимальным и минимальным элементами) для четных элементов

28	Подсчет числа четных элементов, больших среднего гармонического
29	Подсчет суммы нечетных элементов, меньших среднего арифметического
30	Подсчет суммы нечетных элементов, меньших среднего гармонического

4.2. Лабораторная работа №2

Внимательно ознакомьтесь с разделом 1 текущего Приложения. Обязательно к выполнению:

- Необходимо разделение на h и cpp файлы для каждого класса. Функция main должна располагаться в отдельном cpp файле.
- Элемент очереди/стека/списка необходимо реализовать с помощью класса или структуры.
- Необходимо работать с динамическим выделением памяти. Не использовать STL библиотеку.
- Необходимо написать для каждого класса: необходимое число конструкторов (без параметров, с параметрами, копирования) и деструктор.
- Необходимо реализовать передачу и возвращение объектов класса по значению, по указателю и по ссылке.
- Реализовать пользовательское меню согласно заданию. Пользователю должны быть доступны все действия по работе с объектом класса.
- Необходимо использовать аргументы по умолчанию и спецификатор explicit.
- Внимательно читайте задание. Перегрузка операторов возможна как для работы со встроенными типами данных (числами), так и с экземплярами вашего класса.
- У пользователя должна быть возможность ввода, модификации и удаления значений (обнуление) среди прочих доступных действий. Весь перечень действий необходимо определить студенту самостоятельно исходя из задания и создаваемых классов. Например, при работе с очередью необходимо иметь возможности очищения, добавления, удаления и пр.
- В main вид унарной операции должен иметь вид: a = b++; a = ++b; a = --b; a = b--. На экран необходимо выводить a и b. Принцип действия постфиксных и префиксных операторов должен быть сохранен.
- В main вид бинарной операции должен иметь вид: a = b + c. На экран необходимо вывести a, b и c для проверки результата вычисления.
- Номер варианта соответствует номеру студента в списке. Каждому студенту необходимо выполнить оба задания.

Вариант 1.

• Задание 1 Унарная операция

Создать класс «Циклическая очередь». Необходимо перегрузить следующие операторы: оператор ! перегрузить для проверки циклической очереди на пустоту; оператор префиксного инкрементирования как метод для добавления элемента в очередь, при этом пользователь вводит значение с клавиатуры; оператор префиксного декрементирования как дружественную функцию для извлечения элемента из очереди; оператор постфиксного инкрементирования как дружественную функцию для добавления двух элементов в очередь, при этом пользователь вводит значения с клавиатуры; оператор постфиксного декрементирования как метод для удаления из очереди двух элементов с выводом на экран.

• Задание 2. Бинарная операция

Создать класс целых чисел. Необходимо перегрузить следующие операторы: оператор сложения как метод для работы с числами (не с экземплярами вашего класса), вводимыми пользователем с клавиатуры; оператор вычитания как дружественную функцию для работы с числами (не с экземплярами вашего класса), вводимыми пользователем с клавиатуры. Число в этих операторах может располагаться как слева, так и справа. Также необходимо перегрузить операторы сравнения (<, >, ==, !=, <=, >=) с другими экземплярами вашего класса. На усмотрение студента остается решение какой оператор какой перегрузки требует (метод или дружественная функция).

Вариант 2

• Задание 1 Унарная операция

Создать класс целых чисел. Необходимо перегрузить следующие операторы: оператор префиксного декрементирования как метод для уменьшения значения, хранящегося в классе, на пользовательское число; оператор префиксного инкрементирования как метод для увеличения значения на единицу; оператор постфиксного декрементирования как дружественную функцию для уменьшения значения на два; оператор постфиксного инкрементирования как дружественную функцию для увеличения значения на два; оператор преобразования типа в int.

• Задание 2. Бинарная операция

Создать класс «Двусвязный линейный список», который хранит целые числа. Размер списка вводится с клавиатуры пользователем, значения списка заполняются случайным образом числами от 5 до 35. Необходимо перегрузить операторы сравнения (<, >, ==, !=, <=, >=) для работы с числами (не с экземплярами вашего класса), вводимыми пользователем с клавиатуры. Сравнение должно выполнять работу с каждым значением, хранящимся в списке. Например, оператор сравнения больше для списка {5, 7, 15, 25} с числом 9 на экране будет выглядеть следующим образом: 5 > 9 - false, 7 > 9 - false, 15 > 9 - true, 25 > 9 - true. На усмотрение студента остается решение какой оператор какой перегрузки требует (метод или дружественная функция). Также

требуется перегрузить оператор “круглые скобки” для возвращения подсписка, индексы для этого оператора вводятся пользователем с клавиатуры.

Вариант 3

• Задание 1 Унарная операция

Создать класс вещественных чисел. Необходимо перегрузить следующие операторы: оператор префиксного декрементирования как метод для уменьшения значения на 0,1; оператор префиксного инкрементирования как метод для увеличения значения на 0,5; оператор постфиксного декрементирования как дружественную функцию для уменьшения значения на 0,2; оператор постфиксного инкрементирования как дружественную функцию для увеличения значения на 2,5; операторы ввода и вывода в поток для вывода на экран и ввода с клавиатуры данных в объект соответственно.

• Задание 2. Бинарная операция

Создать класс"Стек", размер стека вводится с клавиатуры пользователем, стек заполняется случайными числами от 10 до 30. Требуется перегрузить арифметические операторы для работы с пользовательским числом (не с экземплярами вашего класса), вводимыми пользователем с клавиатуры: +, -, /, *, +=, -=, /=, *=. Для операторов +, -, * и / число может стоять как слева от оператора, так и справа. На усмотрение студента остается решение какой оператор какой перегрузки требует (метод или дружественная функция).

Вариант 4

• Задание 1. Унарная операция

Создать класс “Очередь”. Необходимо перегрузить следующие операторы: оператор постфиксного инкрементирования как метод для добавления элемента в очередь, при этом значение вводится пользователем с клавиатуры; оператор постфиксного декрементирования как метод для извлечения элемента из очереди; оператор префиксного инкрементирования как дружественную функцию для увеличения всех чисел в очереди на единицу; оператор префиксного декрементирования как дружественную функцию для уменьшения всех чисел в очереди на единицу; оператор ! как метод для проверки очереди на пустоту.

• Задание 2. Бинарная операция

Создать класс координат, содержащий x, y и z. Перегрузить оператор сложения как метод и оператор вычитания как дружественную функцию для сложения и вычитания соответственно с другим экземпляром класса и числом, вводимыми пользователем с клавиатуры, при этом число может располагаться как слева, так и справа от оператора; операторы ввода и вывода в поток; все операторы сравнения с другим экземпляром класса (сравнение должно происходить между

соответствующими координатами). На усмотрение студента остается решение какой оператор какой перегрузки требует (метод или дружественная функция).

Вариант 5

• Задание 1 Унарная операция

Создать класс “Дата”, который содержит число, месяц и год. Необходимо перегрузить следующие операторы: оператор префиксного инкрементирования как метод для увеличения на выбор пользователя либо дня, либо месяца, либо года на 1; оператор префиксного декрементирования как метод для аналогичного уменьшения; оператор постфиксного инкрементирования как дружественную функцию для увеличения на выбор пользователя либо дня, либо месяца, либо года на пользовательское число; оператор постфиксного декрементирования как метод для аналогичного уменьшения; оператор преобразования типа int.

• Задание 2. Бинарная операция

Создать класс “Очередь”. Размер и значения должны вводиться пользователем с клавиатуры. Необходимо перегрузить следующие операторы: оператор сложения как метод для добавления нового элемента в очередь, вводимым пользователем с клавиатуры; оператор умножения как дружественную функцию для умножения данных очереди на пользовательское число; оператор вычитания для извлечения элемента из очереди; оператор равенства для работы с другими экземплярами класса; операторы сравнения на равенство и неравенство поэлементное двух экземпляров класса (сравнение должно выполнять работу с каждым значением, хранящимся в очереди, например, оператор сравнения на равенство для очередей {5, 7, 15, 25} и {1, 7, 10, 25} на экране выведет: 5 == 1 - false, 7 ==7 - true, 15 == 10 - false, 25 == 25 - true); операторы ввода и вывода в поток. На усмотрение студента остается решение какой оператор какой перегрузки требует (метод или дружественная функция).

Вариант 6

• Задание 1 Унарная операция

Создать класс “Циклический односвязный список”. Размер и значения объекта должны вводиться пользователем с клавиатуры. Необходимо перегрузить следующие операторы: оператор префиксного инкрементирования как метод для добавления одного элемента в список, вводимого пользователем с клавиатуры; оператор постфиксного инкрементирования как метод для добавления двух элементов в список, вводимых пользователем с клавиатуры; оператор префиксного декрементирования как дружественную функцию для сообщения пользователю о том, что хранится на текущей позиции, затем удаления элемента и перехода на следующий элемент; оператор постфиксного декрементирования как дружественную функцию для сообщения

пользователю о том, что хранится на текущей позиции и на следующей, после - удаление текущей позиции и следующей за ней.

• Задание 2 Бинарная операция

Создать класс “Габариты”, который включает в себя высоту, ширину, глубину. Параметры объекта вводятся пользователем с клавиатуры. Необходимо перегрузить арифметические операторы +, -, /, *, +=, -=, /=, *= для работы с числами, вводимыми пользователем с клавиатуры. Для операторов +, -, * и / число может стоять как слева от оператора, так и справа. На усмотрение студента остается решение какой оператор какой перегрузки требует (метод или дружественная функция).

Вариант 7

Создать класс "Циклическая очередь". Размер циклической очереди вводится пользователем с клавиатуры, а значения задаются случайным образом от -10 до 10. Необходимо перегрузить префиксные унарные операции как метод для извлечения и добавления в очередь значений, вводимых пользователем с клавиатуры; постфиксные унарные операции как дружественную функцию для уменьшения или увеличения соответственно всех значений в очереди на пользовательское число; бинарные операторы /, /=, +, +=, -=, =, !=, ==, >=, <=, >, < для работы с другими экземплярами класса и числами. Для бинарных операторов, которые это позволяют, число может располагаться в том числе и слева от оператора; операторы ввода в поток и вывода в поток. На усмотрение студента остается решение какой оператор какой перегрузки требует (метод или дружественная функция).

Вариант 8

• Задание 1. Унарная операция

Создать класс «Время», который состоит из секунд, минут и часов. Необходимо перегрузить операторы следующим образом: оператор префиксного инкрементирования как дружественную функцию для увеличения на выбор пользователя либо секунд, либо минут, либо часов на единицу; оператор префиксного декрементирования как дружественную функцию для аналогичного уменьшения; оператор постфиксного инкрементирования как метод для увеличения на выбор пользователя либо секунд, либо минут, либо часов на пользовательское значение; оператор постфиксного декрементирования как метод для аналогичного уменьшения.

• Задание 2. Бинарная операция

Создать класс “Очередь”. Размер каждой очереди вводится пользователем с клавиатуры, а значения задаются случайным образом от 10 до 20. Необходимо перегрузить следующие операторы: оператор += как дружественную функцию для добавления элемента в очередь, при этом значение элемента вводится пользователем с клавиатуры; оператор сложения как метод для

сложения двух экземпляров класса; оператор -= как метод для извлечения элемента из очереди; оператор вычитания как дружественную функцию для вычитания экземпляров класса друг из друга; оператор умножения как метод для поэлементного умножения экземпляров класса друг на друга; оператор деления как метод для деления очереди на пользовательское число.

Вариант 9

• Задание 1. Унарная операция

Создать класс координат, содержащий поля x, y, z. Необходимо перегрузить следующие операторы: оператор постфиксного инкрементирования как метод для прибавления 1,5 ко всем координатам; оператор постфиксного декрементирования как метод для вычитания 5,9 от каждой координаты; оператор префиксного инкрементирования как дружественную функцию для сложения всех трех координат; оператор префиксного декрементирования как метод для для вычитания выбранных пользователем координат из пользовательского числа.

• Задание 2. Бинарная операция

Создать класс “Стек”. Размер для каждого стека вводится с клавиатуры, заполняется каждый стек случайными числами от 0 до 20. Необходимо перегрузить для работы с другими экземплярами вашего класса следующие операторы сравнения: !=, <, >, <=, >=, ==. Дополнительно требуется перегрузить оператор ( ) для возвращения подстека. На усмотрение студента остается решение какой оператор какой перегрузки требует (метод или дружественная функция).

Вариант 10

• Задание 1. Унарная операция

Создать объект "Дэк". Размер и значения дэка вводятся с клавиатуры пользователем. необходимо перегрузить следующие операторы: оператор ! как метод для получения дэка с отрицательными значениями; оператор префиксного декрементирования как метод для уменьшения значения всех элементов дэка на единицу; оператор префиксного инкрементирования как метод для увеличения всех элементов на 0,5; оператор постфиксного декрементирования как дружественную функцию для извлечения из дэка; оператор постфиксного инкрементирования как дружественная функция для добавления элемента (пользовательское число) в дэк.

• Задание 2. Бинарная операция

Создать класс “Целое число”. Необходимо перегрузить следующие операторы: оператор сложения, вычитания, умножения и деления как для работы с другими экземплярами класса, так и с пользовательскими числами; оператор преобразования в int. Необходимо учитывать то, что пользовательское число и объект класса могут находится и слева и справа от оператора. На усмотрение студента остается решение какой оператор какой перегрузки требует (метод или дружественная функция).

Вариант 11

• Задание 1. Унарная операция

Создать класс “Односвязный линейный список”. Размер для каждого списка вводится с клавиатуры, заполняется каждый список случайными числами от 10 до 20. Необходимо перегрузить следующие операторы: оператор префиксного инкрементирования как метод для добавления в начало списка нового элемента, вводимого пользователем с клавиатуры; оператор постфиксного инкрементирования как метод для добавления нового элемента, вводимого пользователем с клавиатуры, в конец списка; оператор префиксного декрементирования как дружественную функцию для удаления первого элемента из списка; оператор постфиксного декрементирования как дружественную функцию для удаления последнего элемента из списка.

• Задание 2. Бинарная операция

Создать класс “Время”, который включает в себя поля секунда, минута и час. Необходимо перегрузить следующие операторы: оператор сложения для добавления пользовательского числа к одному из параметров; оператор вычитания для вычитания пользовательского значения из одного из параметров; оператор умножения для умножения всех параметров на пользовательское число; оператор деления для деления всех параметров на пользовательское число. На усмотрение студента остается решение какой оператор какой перегрузки требует (метод или дружественная функция).

Вариант 12

• Задание 1. Унарная операция

Создать класс “Стек”. Размер и значения для каждого стека задаются случайными числами от 0 до 15. Необходимо перегрузить следующие операторы: оператор префиксного инкрементирования как дружественную функцию для добавления элемента (пользовательское число) в стек; оператор постфиксного инкрементирования как дружественную функцию для добавления двух элементов (случайные число от 0 до 15) в стек; оператор префиксного декрементирования для удаления элемента из стека; оператор постфиксного декрементирования для удаления двух элементов из стека; оператор преобразования стека в число int.

• Задание 2. Бинарная операция

Создать класс “Вещественное число”. Необходимо перегрузить следующие операторы: оператор сложения для сложения с другим экземпляром класса; оператор вычитания для выполнения функции вычитания с пользовательским числом; оператор деления для выполнения функций деления с пользовательским числом; оператор умножения для умножения на другой экземпляр класса; оператор присваивания для присваивания как пользовательского значения, так и другого экземпляра класса; оператор преобразования в тип double. Необходимо учитывать, что

пользовательские числа могут находится слева и справа от оператора. На усмотрение студента остается решение какой оператор какой перегрузки требует (метод или дружественная функция).

Вариант 13

• Задание 1. Унарная операция

Создать класс “Целое число”. Необходимо перегрузить следующие операторы: оператор префиксного декрементирования как метод для вычитания пользовательского числа; оператор постфиксного декрементирования как метод для вычитания единицы; оператор префиксного инкрементирования как дружественную функцию для прибавления пользовательского числа; оператор постфиксного инкрементирования как дружественную функцию для прибавления единицы; оператор ! как метод для возвращения отрицательного значения, хранящегося в экземпляре класса.

• Задание 2. Бинарная операция

Создать класс "Стек". Размер и значения стека задаются пользователем с клавиатуры. Необходимо перегрузить следующие операторы для работы с числами: +, +=, -, -=, *, *=, /, /=. Необходимо учитывать то, что пользовательское число и объект класса могут находится и слева и справа от оператора. На усмотрение студента остается решение какой оператор какой перегрузки требует (метод или дружественная функция).

Вариант 14

Создать класс "Однонаправленный циклический список". Размер и значения каждого списка задаются случайным образом. Необходимо перегрузить следующие операторы: оператор префиксного инкрементирования для как метод добавления элемента (пользовательское число) в конец списка; оператор постфиксного декрементирования как дружественная функция для удаления первого элемента из списка; оператор постфиксного инкрементирования как метод для добавления элемента (пользовательское число) в начало списка; оператор префиксного декрементирования как метод для удаления последнего элемента из списка; оператор присваивания для присвоения двух экземпляров класса друг другу; операторы сравнения для работы с экземплярами класса как дружественные функции ==, !=, >, <, >=, <=; оператор [ ] как метод для получения элемента списка; оператор ( ) как метод для получения подсписока от первого до пятого элемента; операторы сложения, вычитания, умножения и деления для соответствующего действия работы с другим экземпляром класса; оператор ввода и вывода в поток.

Вариант 15

• Задание 1. Унарная операция

Создать класс "Однонаправленный линейный список". Размер и значения списка задаются пользователем с клавиатуры. Необходимо перегрузить следующие операторы: оператор префиксного инкрементирования как метод для добавления элемента (пользовательское число) в конец списка; оператор префиксного декрементирования как метод для удаления элемента из конца списка; оператор постфиксного инкрементирования как дружественную функцию для добавления элемента (случайное значение от 0 до 10) в начало списка; оператор постфиксного декрементирования как дружественную функцию для удаления элемента из начала списка.

• Задание 2. Бинарная операция

Создать класс “Время”, который содержит параметры секунда, минута и час. Необходимо перегрузить следующие операторы: оператор сложения как метод для увеличения на пользовательское число на выбор пользователя либо секунд, либо минут, либо часа; оператор вычитания как дружественную функцию для аналогичного уменьшения; все операторы сравнения двух экземпляров класса часть как дружественные функции и часть как методы

Вариант 16

• Задание 1 Унарная операция

Создать класс “Вещественное число”. Необходимо перегрузить следующие операторы: оператор префиксного инкрементирования как метод для увеличения значения на пользовательское число; оператор постфиксного инкрементирования как дружественная функция для увеличения значения на случайное число (от 0 до 19.99); оператор префиксного декрементирования как метод для уменьшения значения на случайное число(от 0 до 19.99); оператор постфиксного декрементирования как дружественная функция для для уменьшения значения на пользовательское число; оператор ! как метод для возвращения отрицательного значения; оператор преобразование в тип float.

• Задание 2 Бинарная операция

Создать класс "Стек". Размер и значения каждого стека задаются случайным образом. Необходимо перегрузить следующие операторы для работы с другими экземплярами класса: +, *, =, /, -, +=, *=, /=, -=. На усмотрение студента остается решение какой оператор какой перегрузки требует (метод или дружественная функция).

Вариант 17

• Задание 1 Унарная операция

Создать класс “Координаты”, который содержит три поля данных x, y и z. Необходимо перегрузить операторы следующие операторы: оператор ! как метод для умножения всех данных на (-1); оператор префиксного инкрементирования как дружественная функция для увеличения всех полей данных на наименьшее значение среди них; оператор постфиксного

инкрементирования как дружественная функция для увеличения одного поля данных на выбор пользователя на пользовательское число; оператор префиксного декрементирования для уменьшения всех полей данных на наибольшее значение среди них; оператор постфиксного декрементирования для уменьшения одного поля данных на выбор пользователя на пользовательское число.

• Задание 2 Бинарная операция

Создать класс "Стек". Размер и значения каждого стека задаются случайным образом. Необходимо перегрузить следующие операторы для работы с пользовательскими числами: -, -=, /=, /, =, ==, <, >, !; операторы ввода и вывода в поток; оператор ( ) для получения подстека. На усмотрение студента остается решение какой оператор какой перегрузки требует (метод или дружественная функция).

Вариант 18

• Задание 1 Унарная операция

Создать класс “Время”, который включает три поля данных - секунда, минута и час. Необходимо перегрузить следующие операторы: оператор префиксного инкрементирования как метод для увеличения всех полей данных на наименьшую разницу между значениями секунд, минут и часов; оператор постфиксного инкрементирования как метод для увеличения на выбор пользователя одного из полей на наибольшее значение среди них; оператор префиксного декрементирования как метод для уменьшения всех полей данные на наибольшую разницу между значениями секунд, минут и часов; оператор постфиксного декрементирования для уменьшения на выбор пользователя одного из полей данных класса на пользовательское значение.

• Задание 2 Бинарная операция

Создать класс “Дэк”. Размер каждого дэка определяется пользователем, каждый дэк заполняется случайными числами от 15 до 20. Необходимо перегрузить следующие операторы: оператор сложения как метод для работы с пользовательскими числами (при этом число может находится как слева, так и справа от оператора); оператор умножения как дружественную функцию для работы с пользовательскими числами (при этом число может находится как слева, так и справа от оператора); оператор += как метод для добавления элемента (случайное число от 15 до 20) в дэк; оператор *= как дружественную функцию для умножения верхушки дэка на пользовательское число; оператор присваивания; оператор проверки на равенство и неравенство как методы поэлементное двух дэков друг другу.

Вариант 19

• Задание 1 Унарная операция

Создать класс “Стек”. Размер стека задается пользователем, значения, хранящиеся в стеке вводятся с клавиатуры. Необходимо перегрузить следующие операторы: оператор ! как дружественную функцию для проверки стека на пустоту; оператор префиксного инкрементирования как дружественную функцию для увеличения всех значений в стеке на его максимальное значение; оператор постфиксного инкрементирования как дружественную функцию для увеличения всех значений в стеке на его минимальное значение; оператор префиксного декрементирования как метод для уменьшения всех значений в стеке на его максимальное значение; оператор постфиксного декрементирования как метод для уменьшения все значения в стеке на его минимальное значение.

• Задание 2 Бинарная операция

Создать класс “Вещественное число”. Необходимо перегрузить следующие операторы: оператор вычитания, сложения, умножения и деления для работы с другими экземплярами класса; операторы сравнения <, >, != и == для работы с другими экземплярами класса; операторы преобразования в типы double и float. На усмотрение студента остается решение какой оператор какой перегрузки требует (метод или дружественная функция).

Вариант 20

• Задание 1 Унарная операция

Создать класс "Очередь". Размер и значения очереди определяется случайным образом (значение от 3 до 10). Необходимо перегрузить следующие операторы: оператор ! как дружественную функцию для умножения всех значений очереди на (-1); оператор префиксного инкрементирования как метод для увеличения всех значений очереди на пользовательское число; оператор постфиксного инкрементирования как метод для увеличения всех значений очереди размер очереди; оператор префиксного декрементирования как метод для уменьшения всех значений очереди на пользовательское число; оператор постфиксного декрементирования как метод для уменьшения всех значений очереди на значение вычистываемое как (размер очереди * 2).

• Задание 2 Бинарная операция

Создать класс “Габариты”, который включает три поля данных: высота, ширина и глубина. Необходимо перегрузить все операторы поэлементного сравнения с числом и с другим экземпляром класса (половину как дружественные функции, половину как методы); операторы ввода и вывода в поток; оператор преобразования в тип int и float. На усмотрение студента остается решение какой оператор какой перегрузки требует (метод или дружественная функция).

Вариант 21

• Задание 1 Унарная операция

Создать класс “Габариты”, который содержит три поля данных высоту, ширину и глубину. Необходимо перегрузить следующие операторы: оператор префиксного инкрементирования как метод для увеличения всех полей данных на максимальное значение, хранящееся в объекте класса; оператор постфиксного инкрементирования как дружественная функция для увеличения на выбор пользователя одного из полей данных класса на пользовательское число; оператор постфиксного декрементирования как метод для уменьшения всех данных на минимальное значение объекта класса; оператор префиксного декрементирования как метод для уменьшения на выбор пользователя одно из полей объекта класса на пользовательское число.

• Задание 2 Бинарная операция

Создать класс “Циклический односвязный список”. Размер каждого списка задается пользователем, значения каждого списка задаются случайным образом (максимальное значение 100). Необходимо перегрузить операторы сложения, вычитания, деления и умножения для работы с пользовательскими числами; операторы сравнения на равенство и неравенство для работы с другими экземплярами класса. Необходимо учитывать то, что пользовательское число и объект класса могут находится и слева и справа от оператора. На усмотрение студента остается решение какой оператор какой перегрузки требует (метод или дружественная функция).

Вариант 22

• Задание 1 Унарная операция

Создать класс “Очередь”. Размер очереди задается пользователем с клавиатуры, значения очереди задаются случайными числами от 10 до 30. Необходимо перегрузить следующие операторы: оператор префиксного инкрементирования как метод для увеличения размера очереди на пользовательское число при этом новые элементы генерируются случайным образом; оператор постфиксного инкрементирования как дружественная функция для увеличения всех значений очереди на ее минимальное значение; оператор префиксного декрементирования как метод для уменьшения размера очереди на пользовательское число; оператор постфиксного декрементирования как дружественная функция для уменьшения всех значений очереди на ее максимальное значение.

• Задание 2 Бинарная операция

Создать класс “Целое число”. Необходимо перегрузить все арифметические операторы для работы с числами и с другими экземплярами класса: +, -, /, *, +=, -=, /=, *=; оператор присваивания; оператор преобразования в тип int и double; операторы ввода и вывода в поток. Необходимо учитывать то, что пользовательское число и объект класса могут находится и слева и справа от арифметического оператора. На усмотрение студента остается решение какой оператор какой перегрузки требует (метод или дружественная функция).

Вариант 23

• Задание 1 Унарная операция

Создать класс “Координаты”, который содержит три поля данных: x, y и z. Необходимо перегрузить следующие операторы: оператор префиксного инкрементирования как метод для увеличения на выбор пользователя одного поля данных класса на пользовательское число; оператор постфиксного инкрементирования как дружественная функция для увеличения на выбор пользователя одного поля данных класса на максимальное число среди хранимых классом значений; оператор префиксного декрементирования как дружественная функция для уменьшения на выбор пользователя одного поля данных класса на пользовательское число; оператор постфиксного декрементиорвания как метод для увеличения на выбор пользователя одного поля данных класса на минимальное число среди хранимых классом значений.

• Задание 2 Бинарная операция

Создать класс “Циклическая очередь”. Размер и значения очереди задаются случайными числами (от 5 до 30). Необходимо перегрузить следующие арифметические операторы для работы с другими экземплярами класса: +, -, /, *; арифметические операторы +=, -=, /=, *= для работы с пользовательскими числами. На усмотрение студента остается решение какой оператор какой перегрузки требует (метод или дружественная функция).

4.3. Лабораторная работа №3

Внимательно ознакомьтесь с разделом 1 текущего Приложения. Обязательно к выполнению:

- Необходимо разделение на h и cpp файлы для каждого класса. Функция main должна располагаться в отдельном cpp файле.
- Необходимо использовать динамическое выделение памяти и не использовать STL библиотеку.
- Необходимо написать для каждого класса все необходимые и требуемые конструкторы, деструктор.
- Обратите внимание на формулировку задания, если не указано создание абстрактного класса, то НЕ нужно создавать абстрактный класс. Также это означает, что могут быть созданы объекты базового класса.
- В main необходимо реализовать массив типа базового класса. Пользователь выбирает какой класс-наследника ему необходимо создать, также число наследников программно не ограничено.
- Реализуйте статическую переменную класса для подсчета количества объектов базового класса.
- Используйте модификаторы override и final в своей программе

- Необходимо иметь возможность сохранить данные, которые хранятся в программе в выходной файл.
- Все параметры вводятся пользователем или из входного файла или с клавиатуры, не должно существовать в программе параметров, заданных по умолчанию.
- Должно быть представлено максимально возможное меню пользователя, со всеми действиями, которые может выполнить пользователь при работе с программой, в том числе редактирование, добавление или удаление объектов.
- Номер варианта соответствует номеру студента в списке.

Вариант 1

Создать абстрактный базовый класс с виртуальной функцией «Площадь». Создать производные классы «Прямоугольник», «Круг», «Прямоугольный треугольник», «Трапеция» со своими функциями площади и переменными. Для проверки определить массив ссылок на абстрактный класс, которым присваиваются адреса различных объектов.

Вариант 2

Создать абстрактный класс с виртуальной функцией «Норма». Создать производные классы «Комплексные числа», «Вектор из 10 элементов», «Матрица (2x2)». Определить функцию нормы: для комплексных чисел - модуль в квадрате, для вектора - корень квадратный из суммы элементов по модулю, для матрицы - максимальное значение по модулю.

Вариант 3

Создать абстрактный класс «Кривые» для вычисления координаты y для некоторой x. Создать производные классы «Прямая», «Эллипс», «Гипербола» со своими функциями вычисления y в зависимости от входного параметра x.

Вариант 4

Создать абстрактный базовый класс с виртуальной функцией «Сумма прогрессии». Создать производные классы «Арифметическая прогрессия» и «Геометрическая прогрессия». Каждый класс имеет два поля типа double. Первое - первый член прогрессии, второе - постоянная разность (для арифметической) и постоянное отношение (для геометрической). Определить функцию вычисления суммы, где параметром является количество элементов прогрессии.

Вариант 5

Создать абстрактный базовый класс с виртуальной функцией «Площадь поверхности». Создать производные классы «Параллелепипед», «Тетраэдр», «Шар» со своими функциями площади поверхности. Для проверки определить массив ссылок на абстрактный класс, которым присваиваются адреса различных объектов.

Вариант 6

Создать базовый класс «Фигура» и производные классы «Круг», «Прямоугольник», «Трапеция», «Треугольник». Определить виртуальные функции «Площадь», «Периметр» и «Вывод на консоль».

Вариант 7

Создать базовый класс «Работник» и производные классы «Служащий с почасовой оплатой», «Служащий в штате» и «Служащий с процентной ставкой». Базовый класс определяет для каждого работника наличие имени, фамилии, телефона, адреса. Определить функцию начисления зарплаты.

Вариант 8

Создать базовый класс «Список». Реализовать на базе списка стек и очередь с виртуальными функциями вставки и извлечения. Наследники хранят целые числа.

Вариант 9

Определить абстрактный базовый класс «Печатное издание», для которого существует наименование, год издания, наименование издательства, количество страниц, аннотация; определить метод вывода на экран. Определить наследников: журнал, учебник, книга. Для каждого наследника переопределить вывод на экран.

Вариант 10

Создать абстрактный базовый класс с виртуальной функцией «Объем». Создать производные классы «Параллелепипед», «Пирамида», «Тетраэдр», «Шар» со своими функциями объема. Для проверки определить массив ссылок на абстрактный класс, которым присваиваются адреса различных объектов.

Вариант 11

Создать абстрактный класс «Млекопитающие». Определить производные классы «Животные» и «Люди». У животных определить производные классы собак, котов и рыб, которые содержат в себе породу, кличку, окрас, ФИО владельца и размеры. Для класса «Люди» существуют поля: ФИО, дата рождения, национальность. Определить виртуальные функции описания человека, собаки, кота и рыбы.

Вариант 12

Создать базовый класс «Предок», у которого есть ФИО, дата рождения. Определить виртуальную функцию вывода на консоль. Создать производный класс «Ребенок», у которого функция вывода на консоль дополнительно выводит имя его родителя и его собственное, и его дату рождения. Создать производный класс от последнего класса – «Внук», у которого есть ФИО, дата рождения, город рождения; написать свою функцию вывода на консоль всех данных.

Вариант 13

Определить класс для хранения массива целочисленных чисел. Определить функцию для вывода на экран значений. Создать 2 производных класса, которые переводят целочисленные значения в двоичную и восьмеричную формы. В производных классах переопределить вывод на экран в соответствии с задачей наследника.

Вариант 14

Создать абстрактный базовый класс с виртуальной функцией «Корни уравнения». Создать производные классы «Класс линейных уравнений» и «Класс квадратных уравнений». Определить функцию вычисления корней уравнений.

Вариант 15

Создать базовый абстрактный класс «Персона». Наследниками являются: студент, преподаватель, заведующий кафедрой. Каждый класс содержит общие переменные: ФИО, дата рождения, так и специфические, для студента: номер группы, дата поступления, средний балл; для преподавателя: перечень дисциплин, общее количество студентов; заведующий кафедрой: наименование кафедры. Определить функцию вывода всей информации о любой персоне на экран.

Вариант 16

Создать базовый абстрактный класс «Рубль», наследниками являются «Евро», «Доллар», «Фунт стерлингов» и «Японская иена». С клавиатуры вводятся соотношения валют друг относительно друга. Определить функции преобразования рублей в соответствующую валюту и наоборот, и вывод на экран.

4.4. Лабораторная работа №4

Внимательно ознакомьтесь с разделом 1 текущего Приложения. Обязательно к выполнению:

- Функция main должна располагаться в отдельном cpp файле.
- Реализовать динамическое выделение памяти и очищение. Не использовать библиотеку STL.
- Если требуется реализовать стек/очередь/дек/список/дерево, то элемент такой дисциплины обязан быть выполнен в виде класса.
- Обязательно реализовать работу с несколькими исключительными ситуациями для объекта класса. Если в ЛР студенту не очевидно какую исключительную ситуацию следует обрабатывать, следует обратиться к преподавателю. Обработка исключений должна производиться в обоих заданиях.
- Реализовать пользовательское меню согласно заданию. Обязательно реализовать возможность выбора типа данных, с которыми возможно взаимодействие в каждом задании: int, char, float, double, char*. Не должно быть в программе параметров, которые задаются в main, все,

что может задать пользователь должно задаваться с клавиатуры, если в задании не указано иначе (имеется в виду заполнение случайными данными).

- Класс должен содержать все необходимые конструкторы и деструктор.
- По списку студент определяет свой порядковый номер. Необходимо выполнить оба задания.

Вариант 1

Задание 1

Написать класс шаблон последовательного поиска в массиве по ключу. Основная функция поиска должна возвращать индексы всех элементов, найденных в массиве, равных ключу.

Задание 2

Создать параметризованный класс «множество» и перегрузить операторы != проверка на неравенство множеств, == проверка на равенство множеств, [] для доступа по индексу.

Вариант 2

Задание 1

Написать функцию-шаблон, вычисляющую среднее арифметическое по значениям в массиве. Размер массива задается пользователем, значения массива генерируются случайным образом (от 0 до 50).

Задание 2

Создать параметризованный стек с перегруженными операторами потокового ввода/вывода, оператор присваивания =, оператор += для добавления в стек, оператор – для извлечения из стека.

Вариант 3

Задание 1

Написать функцию-шаблон, переставляющую элементы в массиве с шагом, заданным пользователем. Размер и значения массива генерируются случайным образом (до 20).

Задание 2

Создать параметризованный массив с перегруженными операторами [] для доступа по индексу, = для присваивания массивов друг другу, вывода и ввода в поток, == для сравнения массивов.

Вариант 4

Задание 1

Написать функцию-шаблон двоичного поиска.

Задание 2

Создать параметризованную очередь с перегруженным оператором потокового ввода/вывода и перегруженным оператором + для сложения двух очередей, ! для проверки на пустоту и – для вычитания двух очередей.

Вариант 5

Задание 1

Написать функцию-шаблон для инверсии массива. Размер массива и значения задаются случайным образом (до 30).

Задание 2

Создать параметризованный стек. Начальный размер стека вводится с клавиатуры, заполняется случайными значениями. Перегрузить оператор = для присваивания двух стеков друг другу, + для сложения двух стеков, == для сравнения двух стеков, - для вычитания стеков.

Вариант 6

Задание 1

Написать функцию-шаблон, вычисляющую среднее значение в массиве. Размер массива и значения задаются случайным образом (до 100).

Задание 2

Создать параметризованный дэк. Начальный размер дэка генерируется случайным образом (до 30), значения также генерируются случайным образом (до 10). Перегрузить оператор =, потокового ввода/вывода, - для извлечения последнего элемента, + для сложения двух дэков, < для сравнения двух дэков.

Вариант 7

Задание 1

Написать функцию-шаблон, которая ищет разницу между максимальным и минимальным элементом в неупорядоченном массиве. Размер массива и значения задаются случайным образом (от 50 до 100).

Задание 2

Создать параметризированный класс «множество», перегрузить оператор * для пересечения множеств, + для объединения множеств, < для сравнения множеств.

Вариант 8

Задание 1

Написать функцию-шаблон, которая выполняет линейный поиск заданного пользователем значения, выводит на экран его индекс. Массив значений не упорядочен, размер задается случайным образом (от 30 до 100). Значения массива также задаются случайным образом (до 100).

Задание 2

Создать параметризованный класс бинарного дерева. С методами - добавить элемент в дерево, прохождение по дереву в нисходящем и в восходящем порядке. Осуществить поиск по дереву.

Вариант 9

Задание 1

Опишите параметризованную функцию для возведения в квадрат всех данных в массиве. Размер массива изначально неизвестен, пользователь вводит значения массива с клавиатуры.

Задание 2

Создайте шаблон класса массива, в котором есть методы для вычисления суммы и среднего значения хранимых в массиве чисел. Перегрузите потоковый ввод/вывод, оператор + для сложения двух массивов, - для вычитания массивов, != для сравнения массивов, < для сравнения массивов.

Вариант 10

Задание 1

Написать функцию-шаблон, которая выполняет сортировку выбором как для сортировки по возрастанию, так и по убыванию.

Задание 2

Создать параметризованный список, перегрузить оператор потокового ввода/вывода, [] для доступа к элементу заданной позиции, + для объединения двух списков.

Вариант 11

Задание 1

Написать функцию-шаблон, реализующую сортировку слиянием.

Задание 2

Создать параметризированный класс циклической очереди. Размер очереди вводится с клавиатуры, очередь заполняется случайными числами (от 0 до 50). Перегрузить оператор + для добавления элемента в очередь, ! для проверки очереди на пустоту, - для извлечения элемента очереди.

Вариант 12

Задание 1

Написать функцию-шаблон, которая выполняет сортировку вставками. Размер массива задается случайным образом (от 10 до 20 элементов). Значения заполняются случайным образом.

Задание 2.

Создать параметризованный массив с перегруженными операторами ввода/вывода в поток, оператором индексирования [] для доступа по индексу, операторами поэлементного сравнения < и >.

Вариант 13

Задание 1

Написать параметризованную функцию - сортировка методом Шелла.

Задание 2

Написать параметризованный класс двусвязный список элементов. Определить операции сравнения != и ==, + для добавления элемента в конец списка.

Вариант 14

Задание 1

Написать функцию-шаблон двоичного поиска.

Задание 2

Создать параметризованный класс односвязного списка. Перегрузить операторы – для извлечения первого элемента из списка, + для добавления элемента в начало списка, < и > для сравнения списков.

Список литературы

1. Курицын К.А., Рабин А.В., Рождественская К.Н., Технология программирования. Введение в ООП: учеб.пособие. - СПб.:ГУАП, 2018. - 116 стр.

2. Robert B. Murray, C++ Strategies and Tactics 1st Edition.- Publisher: Addison-Wesley Professional, 1993. - 304 p.

3. Шилдт Г., Самоучитель C++, -СПб.: БХВ-Петербург, 2006. - 638 с.

4. Элджер Дж., C++: Библиотека программиста. - СПб.: Питер, 1999. - 320 с.

5. Scott Schacon, Pro Git, 2nd edition, 2014. - 425 с. Перевод: https://git-scm.com/book/ru/v2/