Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

**Пермский национальный исследовательский**

**политехнический университет**

Кафедра «Информационные технологии и автоматизированные системы»

**Структурное и объектно-ориентированное программирование на С++.**

**Лабораторный практикум.**

**Часть II.**

**Объектно-ориентированный анализ и проектирование сложных программных систем.**

Авторы: Бартов Е.И.,   
Ерохин Н.В., Торопицын М.С.

**Пермь, 2018**

Оглавление

[Оглавление 3](#_Toc523770304)

[Лабораторная работа № 1.](#_Toc523770305) [Классы и объекты. Использование конструкторов. Деструкторы 4](#_Toc523770306)

[1.1. Цель работы: 4](#_Toc523770308)

[1.2. Краткие теоретические сведения 4](#_Toc523770309)

[1.3. Постановка задачи 8](#_Toc523770310)

[1.4. Варианты заданий 8](#_Toc523770311)

[1.5. Содержание отчета 9](#_Toc523770312)

[Лабораторная работа № 2.](#_Toc523770313) [Дружественные функции и классы. Перегрузка операций 10](#_Toc523770314)

[2.1. Цель работы: 10](#_Toc523770316)

[2.2. Краткие теоретические сведения 10](#_Toc523770317)

[2.3. Постановка задачи 13](#_Toc523770318)

[2.4. Варианты заданий 13](#_Toc523770319)

[2.5. Содержание отчета 16](#_Toc523770320)

[Лабораторная работа №3.](#_Toc523770321) [Наследование. Принцип подстановки 17](#_Toc523770322)

[3.1. Цель работы: 17](#_Toc523770324)

[3.2. Краткие теоретические сведения 17](#_Toc523770325)

[3.3. Постановка задачи 20](#_Toc523770326)

[3.4. Варианты заданий 20](#_Toc523770327)

[3.5. Содержание отчета 23](#_Toc523770328)

[Лабораторная работа № 4.](#_Toc523770329) [Шаблоны классов 25](#_Toc523770330)

[4.1. Цель работы: 25](#_Toc523770332)

[4.2. Краткие теоретические сведения 25](#_Toc523770333)

[4.3. Постановка задачи 27](#_Toc523770334)

[4.4. Варианты заданий 27](#_Toc523770335)

[4.5. Содержание отчета 31](#_Toc523770336)

[Лабораторная работа № 5.](#_Toc523770337) [Шаблоны функций 31](#_Toc523770338)

[5.1. Цель работы: 31](#_Toc523770340)

[5.2. Шаблоны функций 31](#_Toc523770341)

[5.3. Постановка задачи 32](#_Toc523770342)

[5.4. Варианты заданий 32](#_Toc523770343)

[5.5. Методические указания 32](#_Toc523770344)

[5.6. Содержание отчета 32](#_Toc523770345)

[Лабораторная работа № 6.](#_Toc523770346) [Обработка исключительных ситуаций 34](#_Toc523770347)

[6.1. Цель работы: 34](#_Toc523770349)

[6.2. Краткие теоретические сведения 34](#_Toc523770350)

[6.3. Постановка задачи 37](#_Toc523770351)

[6.4. Варианты заданий 37](#_Toc523770352)

[6.5. Содержание отчета 40](#_Toc523770353)

[Лабораторная работа № 7.](#_Toc523770354) [Контейнерные классы 42](#_Toc523770355)

[7.1. Цель работы: 42](#_Toc523770357)

[7.2. Краткие теоретические сведения 42](#_Toc523770358)

[7.3. Постановка задачи 44](#_Toc523770359)

[7.4. Варианты заданий 44](#_Toc523770360)

[7.5. Содержание отчета 47](#_Toc523770361)

[Лабораторная работа № 8.](#_Toc523770362) [Базы данных 48](#_Toc523770363)

[8.1. Цель работы: 48](#_Toc523770365)

[8.2. Постановка задачи 48](#_Toc523770366)

[8.3. Варианты заданий 48](#_Toc523770367)

[8.4. Методические указания 51](#_Toc523770368)

[8.5. Содержание отчета 51](#_Toc523770369)

[Лабораторная работа № 9.](#_Toc523770370) [Поиск данных с помощью хеш-таблиц 52](#_Toc523770371)

[9.1. Цель работы: 52](#_Toc523770373)

[9.2. Краткие теоретические сведения 52](#_Toc523770374)

[9.3. Постановка задачи 60](#_Toc523770375)

[9.4. Варианты заданий 60](#_Toc523770376)

[9.5. Методические указания 63](#_Toc523770377)

[9.6. Содержание отчета 63](#_Toc523770378)

## Лабораторная работа № 1

## Классы и объекты. Использование конструкторов. Деструкторы



### Цель работы:

1. Создание консольного приложения, состоящего из нескольких файлов в системе программирования Visual Studio.

2. Создание объектов с использованием конструкторов.

### Краткие теоретические сведения

**Три типа конструкторов**

В каждом классе должен быть хотя бы один метод, который предназначен для инициализации объекта. Его имя совпадает с именем класса, и он вызывается автоматически при инициализации объекта. Автоматический вызов конструктора позволяет избежать ошибок, связанных с использованием неинициализированных переменных.

Существует три типа конструкторов:

* конструктор с параметрами, используется для инициализации объекта требуемыми значениями;
* конструктор без параметров, используется для создания «пустого» объекта;
* конструктор копирования, используется для создания объекта, аналогичного тому, который уже существует.

Конструктор без параметров и конструктор копирования создаются по умолчанию.

**Конструктор с параметрами**

Конструктор с параметрами, используется для инициализации объекта требуемыми

значениями;

Пример:

class Person

{

string name;//динамическая строка

int age;

public:

Person(string Name, int Age)//конструктор с параметрами {

name=Name

age=Age;

}

void Set\_Person(string Name, int age)

{

...

}

void Print\_Person()

{

...

}

...

};

void main()

{

Person P(”Иванов”,21);

P.Print\_Person();

}

**Конструктор без параметров**

Конструктор без параметров используется для создания «пустого» объекта.

Пример:

Person::Person()

{

//пустой объект

name=””;//пустая строка

age=0;

}

или

Person::Person()

{

//объект с полями

name=”Ivanov”;//конкретное значение строки age=25;

}

Конструктор без параметров вызывается следующим образом:

Person p;

**Конструктор копирования**

Конструктор копирования — это специальный вид конструктора, получающий в качестве единственного параметра указатель на объект этого же класса:

Т::T(const T&) { ... /\* Тело конструктора \*/ } ,

где Т — имя класса

Пример:

Person::Person(const Person&p)

{

name=p.name;

age=p.age;

}

Этот конструктор вызывается в тех случаях, когда новый объект создается путем копирования существующего:

* при описании нового объекта с инициализацией другим объектом;
* при передаче объекта в функцию по значению;
* при возврате объекта из функции.

Если программист не указал ни одного конструктора копирования, компилятор создает его автоматически. Такой конструктор выполняет поэлементное копирование полей. Если класс содержит указатели или ссылки, это, скорее всего, будет неправильным, поскольку и копия, и оригинал будут указывать на одну и ту же область памяти.

**Основные свойства конструкторов**

* Конструктор не возвращает значение, даже типа void. Нельзя получить указатель на конструктор.
* Класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации (при этом используется механизм перегрузки).
* Конструктор, вызываемый без параметров, называется конструктором по умолчанию.
* Параметры конструктора могут иметь любой тип, кроме этого же класса. Можно задавать значения параметров по умолчанию, но их может содержать только один из конструкторов.
* Если программист не указал ни одного конструктора, компилятор создает его автоматически. Такой конструктор вызывает конструкторы по умолчанию для полей класса. В случае, когда класс содержит константы или ссылки, при попытке создания объекта класса будет выдана ошибка, поскольку их необходимо инициализировать конкретными значениями, а конструктор по умолчанию этого делать не умеет.
* Конструкторы не наследуются.
* Конструкторы нельзя описывать с модификаторами const, virtual и static.
* Конструкторы глобальных объектов вызываются до вызова функции main. Локальные объекты создаются, как только становится активной область их действия. Конструктор запускается и при создании временного объекта (на пример, при передаче объекта из функции).
* Конструктор вызывается, если в программе встретилась какая-либо из синтаксических конструкций:

имя\_класса имя\_объекта [(список параметров)];// Список параметров не должен быть пустым имя класса (список параметров);// Создается объект без имени (список может быть пустым)

имя\_класса имя\_объекта = выражение;// Создается объект без имени и копируется

Примеры:

Person p1(”Ivanov”,23);//конструктор с параметрами

Person p2();//конструктор без параметров Person p3=p1;// конструктор копирования

Person p4=Person (”Sidorov”,20);//создается объект без имени и копируется

Person\* pp1=new(Person);//указатель на пустой объект Person\*pp2=new Person(”Petrov”,32);//указатель на объект

Существует еще один *способ инициализации полей в конструкторе* — с помощью списка инициализаторов, расположенных после двоеточия между заголовком и телом конструктора:

Person;:Person(int Age):age(Age)

{

name=new char[8];

strcpy(name,”Ivanov”);

}

Поля перечисляются через запятую. Для каждого поля в скобках указывается инициализирующее значение, которое может быть выражением. Без этого способа не обойтись при инициализации *полей-констант,* *полей-ссылок* и *полей-объектов.* В последнем случае будет вызван конструктор, соответствующий указанным в скобках параметрам.

**Деструктор**

Деструктор – это особый вид метода, применяющийся для освобождения ресурсов, выделенных конструктором объекту. Деструктор вызывается автоматически, когда объект удаляется из памяти:

- для локальных объектов это происходит при выходе из блока, в котором они объявлены;

- для глобальных *—* как часть процедуры выхода из main;

- для объектов, заданных через указатели*,* деструктор вызывается неявно при использовании операции delete.

Имя деструктора начинается с тильды (~), непосредственно за которой следует имя класса. Свойства деструктора:

- не имеет аргументов и возвращаемого значения;

- не наследуется;

- не может быть объявлен как const или static (далее);

- может быть виртуальным (далее).

Если деструктор явным образом не определен, компилятор автоматически создает пустой деструктор.

Описывать в классе деструктор явным образом требуется в случае, когда объект содержит указатели на память, выделяемую динамически — иначе при уничтожении объекта память, на которую ссылались его поля-указатели, не будет помечена как свободная. Указатель на деструктор определить нельзя.

Деструктор для класса Person, в котором поле name реализуется как динамическая строка будет выглядеть так:

Person::~Person() {delete [] name;}

Без необходимости явно вызывать деструктор объекта не рекомендуется.

**Определение методов класса**

Методы класса имеют неограниченный доступ ко всем элементам класса, независимо от спецификаторов доступа и порядка объявления методов в классе. Методы могут определяться как в классе, так и вне его. Определение метода внутри класса ничем не отличается от определения обычной функции. По умолчанию такой метод считается встроенной функцией (inline). Если метод определяется вне функции, то принадлежность метода классу указывается с помощью имени класса: Имя\_класса::Имя\_метода. В классе присутствует только прототип. Методы могут быть перегружены, могут принимать аргументы по умолчанию. Метод (кроме статических методов) неявно получает в качестве аргумента указатель на тот объект, для которого он вызван. Этот указатель обозначается ключевым словом this и может быть использован в теле метода. В явном виде этот указатель применяется в основном для возвращения из метода указателя (return this;) или ссылки (return \*this;) на вызвавший объект.

Пример:

Person&Old(Person&P)

{

if(P.GetAge()>60) return \*this;

}

Запись \*this представляет собой значение текущего объекта.

Методы могут быть перегружены.

Методы могут быть константными, то есть не изменять значение полей класса. Константный метод обозначается с помощью слова const после списка аргументов метода. Для объекта-константы может быть вызван только константный метод. Для объекта-переменной может быть вызван и константный и неконстантный методы.

### Постановка задачи

Написать программу, в которой создаются и удаляются объекты определенного пользователем класса. Выполнить исследование вызовов конструкторов и деструкторов.

### Варианты заданий

|  |  |
| --- | --- |
| №  варианта | Задание |
| 1 | СТУДЕНТ  имя – char\*  курс – int  пол – int(bool) |
| 2 | СЛУЖАЩИЙ  имя – char\*  возраст – int  рабочий стаж – int |
| 3 | КАДРЫ  имя – char\*  номер цеха – int  разряд – int |
| 4 | ИЗДЕЛИЕ  имя – char\*  шифр – char\*  количество – int |
| 5 | БИБЛИОТЕКА  имя – char\*  автор – char\*  стоимость – float |
| 6 | ЭКЗАМЕН  имя студента – char\*  дата – int  оценка – int |
| 7 | АДРЕС  имя – char\*  улица – char\*  номер дома – int |
| 8 | ТОВАР  имя – char\*  количество – int  стоимость – float |
| 9 | КВИТАНЦИЯ  номер – int  дата – int  сумма – float |
| 10 | ЦЕХ  имя – char\*  начальник – char\*  количество  работающих – int |
| 11 | ПЕРСОНА  имя – char\*  возраст – int  пол – int(bool) |
| 12 | АВТОМОБИЛЬ  марка – char\*  мощность – int  стоимось – float |
| 13 | СТРАНА  имя – char\*  форма правления – char\*  площать – float |
| 14 | ЖИВОТНОЕ  имя – char\*  класс – char\*  средний вес – int |
| 15 | КОРАБЛЬ  имя – char\*  водоизмещение – int  тип – char\* |
| 16 | ВИДЕОИГРА  название – char\*  дата выхода – int  последняя версия – int |
| 17 | ФИЛЬМ  название – char\*  дата выхода – int  длительность – int |
| 18 | КНИГА  название – char\*  автор – char\*  количество страниц – int |
| 19 | ДЕНЬГИ  валюта – char\*  страна – char\*  количество – int |
| 20 | ДОМ  номер – int  улица – char\*  количество этажей – int |

### Содержание отчета

1. Постановка задачи (общая и для конкретного варианта)

2. Анализ задачи

Определения функций для реализации поставленных задач

Определение функции main()

3. Блок-схемы основных функций

4. Текст программы

5. Тесты

## Лабораторная работа № 2

## Дружественные функции и классы. Перегрузка операций



### Цель работы:

1. Создание консольного приложения, состоящего из нескольких файлов в системе программирования Visual Studio.

2. Использование перегруженных операций в классах.

### Краткие теоретические сведения

**Дружественные функции и классы**

Если необходимо иметь доступ извне к скрытым полям класса, то есть расширить интерфейс класса, то можно использовать дружественные функции и дружественные классы.

Дружественные функции применяются для доступа к скрытым полям класса и

представляют собой альтернативу методам. Метод, как правило, описывает свойство объекта, а виде дружественных функций оформляются действия, не являющиеся свойствами класса, но концептуально входящие в его интерфейс и нуждающиеся в доступе к его скрытым полям, например, переопределенные операции вывода объектов.

Правила описания и особенности дружественных функций:

* Дружественная функция объявляется внутри класса, к элементам которого ей нужен доступ, с ключевым словом friend. В качестве параметра ей должен передаваться объект или ссылка на объект класса, поскольку указатель this ей не передается.
* Дружественная функция может быть обычной функцией или методом другого ранее определенного класса. На нее не распространяется действие спецификаторов доступа, место размещения ее объявления в классе безразлично.
* Одна функция может быть дружественной сразу нескольким классам.

Пример:

class student;//предварительное описание класса

class teacher

{

…

void teach(student &S);

….

};

class student

{

...

friend void teacher::teach(student&);//дружественная функция, //имеет доступ к элементам класса student …

};

Использования дружественных функций нужно по возможности избегать, поскольку они нарушают принцип инкапсуляции и, таким образом, затрудняют отладку и модификацию программы.

Если все методы какого-либо класса должны иметь доступ к скрытым полях другого, весь класс объявляется дружественным с помощью ключевого слова friend.

class student;//предварительное описание класса

class teacher

{

…

void teach(student &S);

….

};

class student

{

...

friend class teacher;//все функции класса teacher являются

дружественными для класса student

…

**};**

**Перегрузка унарных операций**

Унарную операцию можно перегрузить:

* Как компонентную функцию класса
* Как внешнюю (глобальную) функцию

Унарная функция-операция, определяемая внутри класса, должна быть представлена помощью нестатического метода без параметров, при этом операндом является вызвавший ее объект, например:

class Person

{

string name; int age; public:

Person(string, int);//конструктор

.....

//компонентная функция

Person& operator ++() //префиксная операция

{

++age;

return \*this; //указатель на объект, вызвавший метод

}

};

//в основной функции

Person p1(”Ivanov”,20);

++p1;

p1.Show();

Если функция определяется *вне класса,* она должна иметь один параметр типа класса:

class Person

{

string name;

int age;

public:

Person( string, int);//конструктор

…..

//внешняя дружественная функция

friend Person & operator ++(Person&) ;

};

Person & operator ++(Person& p) //префиксная операция

{

++p.age;

return p;

}

//в основной функции

Person p1(”Ivanov”,20);

++p1;

p1.Show();

Операции постфиксного инкремента и декремента должны иметь первый параметр типа int. Он используется только для того, чтобы отличить их от префиксной формы.

**Перегрузка бинарных операций**

Бинарную операцию можно перегрузить:

* Как компонентную функцию класса
* Как внешнюю (глобальную) функцию

*Бинарная функция* - операция, определяемая *внутри класса,* должна быть представлена

* помощью нестатического метода с параметрами, при этом вызвавший ее объект считается первым операндом:

class Person

{

Person & operator перегружаемая операция (тип данных операнда)

{

тело функции

}

};

**Перегрузка операции присваивания**

Операция присваивания определена в любом классе по умолчанию как поэлементное копирование. Эта операция вызывается каждый раз, когда одному существующему объекту присваивается значение другого. Если класс содержит поля, память под которые выделяется динамически, необходимо определить собственную операцию присваивания. Чтобы сохранить семантику присваивания, операция-функция должна возвращать ссылку на объект, для которого она вызвана, и принимать в качестве параметра единственный аргумент — ссылку на присваиваемый объект.

**Перегрузка операций ввода-вывода**

Операции ввода-вывода operator>> и operator<< всегда реализуются как внешние дружественные функции, т. к. левым операндом этих операций являются потоки. Для класса Person соответствующие операции могут выглядеть следующим образом:

class Person

{

string name;

int age;

public:

Person(string, int);//конструктор

…..

//дружественная глобальная функция

friend istream& operator>>(istream&in, Person&p); friend ostream& operator<<(ostream&out, const Person&p); };

……

istream&operator>>(istream&in, Person &p)

{

cout<<"name?"; in>>p.name;

cout<<"age?"; in>>p.age;

return in;

}

ostream&operator<<(ostream&out, const Person&p)

{

return (out<<p.name<<","<<p.age);

}

### Постановка задачи

1. Определить пользовательский класс.

2. Определить в классе следующие конструкторы: без параметров, с параметрами, копирования.

3. Определить в классе деструктор.

4. Определить в классе компоненты-функции для просмотра и установки полей данных (селекторы и модификаторы).

5. Перегрузить операцию присваивания.

6. Перегрузить операции ввода и вывода объектов с помощью потоков.

7. Перегрузить операции указанные в варианте.

8. Написать программу, в которой продемонстрировать создание объектов и работу всех перегруженных операций.

### Варианты заданий

|  |  |
| --- | --- |
| №  варианта | Задание |
| 1 | Создать класс Time для работы с временными интервалами. Интервал должен быть представлен в виде двух полей: минуты типа int и секунды типа int. при выводе минуты отделяются от секунд двоеточием. Реализовать:   * сложение временных интервалов (учесть, что в минуте не может быть более 60 секунд) * сравнение временных интервалов (==) |
| 2 | Создать класс Time для работы с временными интервалами. Интервал должен быть представлен в виде двух полей: минуты типа int и секунды типа int. при выводе минуты отделяются от секунд двоеточием. Реализовать:   * вычитание временных интервалов (учесть, что в минуте не может быть более 60 секунд) * сравнение временных интервалов (!=) |
| 3 | Создать класс Time для работы с временными интервалами. Интервал должен быть представлен в виде двух полей: минуты типа int и секунды типа int. при выводе минуты отделяются от секунд двоеточием. Реализовать:   * добавление секунд (учесть, что в минуте не может быть более 60 секунд) * вычитание секунд |
| 4 | Создать класс Time для работы с временными интервалами. Интервал должен быть представлен в виде двух полей: минуты типа int и секунды типа int. при выводе минуты отделяются от секунд двоеточием. Реализовать:   * добавление секунд (учесть, что в минуте не может быть более 60 секунд) * сравнение временных интервалов (== и !=) |
| 5 | Создать класс Time для работы с временными интервалами. Интервал должен быть представлен в виде двух полей: минуты типа int и секунды типа int. при выводе минуты отделяются от секунд двоеточием. Реализовать:   * вычитание секунд * сравнение временных интервалов (== и !=) |
| 6 | Создать класс Money для работы с денежными суммами. Число должно быть представлено двумя полями: типа long для рублей и типа int для копеек. Дробная часть числа при выводе на экран должна быть отделена от целой части запятой. Реализовать:   * сложение денежных сумм * вычитание денежных сумм |
| 7 | Создать класс Money для работы с денежными суммами. Число должно быть представлено двумя полями: типа long для рублей и типа int для копеек. Дробная часть числа при выводе на экран должна быть отделена от целой части запятой. Реализовать:   * деление сумм * умножение суммы на дробное число |
| 8 | Создать класс Money для работы с денежными суммами. Число должно быть представлено двумя полями: типа long для рублей и типа int для копеек. Дробная часть числа при выводе на экран должна быть отделена от целой части запятой. Реализовать:   * сложение суммы и дробного числа * операции сравнения (>, <, ==) |
| 9 | Создать класс Money для работы с денежными суммами. Число должно быть представлено двумя полями: типа long для рублей и типа int для копеек. Дробная часть числа при выводе на экран должна быть отделена от целой части запятой. Реализовать:   * сложение суммы и дробного числа * операции сравнения (>, <, ==) |
| 10 | Создать класс Money для работы с денежными суммами. Число должно быть представлено двумя полями: типа long для рублей и типа int для копеек. Дробная часть числа при выводе на экран должна быть отделена от целой части запятой. Реализовать:   * операции сравнения (==, !=) * вычитание копеек (--) (постфиксная и префиксная формы) |
| 11 | Создать класс Money для работы с денежными суммами. Число должно быть представлено двумя полями: типа long для рублей и типа int для копеек. Дробная часть числа при выводе на экран должна быть отделена от целой части запятой. Реализовать:   * операции сравнения (<, >) * добавление копеек (++) (постфиксная и префиксная формы) |
| 12 | Создать класс Pair (пара чисел). Пара должна быть представлена двумя полями: типа int для первого числа и типа double для второго. Первое число при выводе на экран должно быть отделено от второго числа двоеточием. Реализовать:   * операции сравнения (<, >) * операция ++, которая работает следующим образом: если форма операции префиксная, то увеличивается первое число, если форма операции постфиксная, то увеличивается второе число |
| 13 | Создать класс Pair (пара чисел). Пара должна быть представлена двумя полями: типа int для первого числа и типа double для второго. Первое число при выводе на экран должно быть отделено от второго числа двоеточием. Реализовать:   * операции сравнения (<, >) * операция --, которая работает следующим образом: если форма операции префиксная, то уменьшается первое число, если форма операции постфиксная, то уменьшается второе число |
| 14 | Создать класс Pair (пара чисел). Пара должна быть представлена двумя полями: типа int для первого числа и типа double для второго. Первое число при выводе на экран должно быть отделено от второго числа двоеточием. Реализовать:   * операции сравнения (==, !=). * вычитание константы из пары (уменьшается первое число, если константа целая, второе, если константа вещественная) |
| 15 | Создать класс Pair (пара чисел). Пара должна быть представлена двумя полями: типа int для первого числа и типа double для второго. Первое число при выводе на экран должно быть отделено от второго числа двоеточием. Реализовать:   * вычитание пар чисел * добавление константы к паре (увеличивается первое число, если константа целая, второе, если константа вещественная) |
| 16 | Создать класс Vector. Вектор должен быть представлен двумя полями типа double для обоих координат (на плоскости). Первое число при выводе на экран должно быть отделено от второго числа двоеточием. Реализовать:   * операцию сложения векторов (+) * операцию умножения числа на вектор (\*) |
| 17 | Создать класс Vector. Вектор должен быть представлен двумя полями типа double для обоих координат (на плоскости). Первое число при выводе на экран должно быть отделено от второго числа двоеточием. Реализовать:   * операцию сравнения длин векторов (==, !=) * операцию умножения числа на вектор (\*) |
| 18 | Создать класс Complex (комплексное число). Комплексное число *x+iy* должно быть представлено в виде пары вещественных чисел x и у типа double, где x – действительная, а y – мнимая части, соответственно.   * сложение комплексных чисел (+) * сравнение комплексных чисел (==, !=) |
| 19 | Создать класс Complex (комплексное число). Комплексное число *x+iy* должно быть представлено в виде пары вещественных чисел x и у типа double, где x – действительная, а y – мнимая части, соответственно.   * умножение комплексных чисел (\*) * сравнение комплексных чисел (==, !=) |
| 20 | Создать класс Complex (комплексное число). Комплексное число *x+iy* должно быть представлено в виде пары вещественных чисел x и у типа double, где x – действительная, а y – мнимая части, соответственно.   * умножение комплексных чисел (\*) * умножение комплексного числа на действительное (\*) |

### Содержание отчета

1. Постановка задачи (общая и для конкретного варианта)

2. Анализ задачи

* Определения функций для реализации поставленных задач
* Определение функции main()

3. Блок-схемы основных функций

4. Текст программы

5. Тесты

## Лабораторная работа №3

## Наследование. Принцип подстановки



### Цель работы:

1. Создание консольного приложения, состоящего из нескольких файлов в системе программирования Visual Studio.

2. Создание иерархии классов с использованием простого наследования.

3. Изучение принципа подстановки.

### Краткие теоретические сведения

**Простое открытое наследование**

Наследование - это механизм получения нового класса на основе уже существующего. Существующий класс может быть дополнен или изменен для создания нового класса.

Существующие классы называются базовыми, а новые – производными. Производный класс наследует описание базового класса; затем он может быть изменен добавлением новых членов, изменением существующих функций-членов и изменением прав доступа. С помощью наследования может быть создана иерархия классов, которые совместно используют код и интерфейсы.

Наследуемые компоненты не перемещаются в производный класс, а остаются в базовых классах.

иерархии производный объект наследует разрешенные для наследования компоненты всех базовых объектов (public, protected).

Допускается множественное наследование – возможность для некоторого класса наследовать компоненты нескольких никак не связанных между собой базовых классов. В иерархии классов соглашение относительно доступности компонентов класса следующее: private – член класса может использоваться только функциями – членами данного

класса и функциями – “друзьями” своего класса. В производном классе он недоступен. protected – то же, что и private, но дополнительно член класса с данным атрибутом

доступа может использоваться функциями-членами и функциями – “друзьями” классов, производных от данного.

public – член класса может использоваться любой функцией, которая является членом данного или производного класса, а также к public - членам возможен доступ извне через имя объекта.

Следует иметь в виду, что объявление friend не является атрибутом доступа и не наследуется.

Синтаксис определения производного класса:

class имя\_класса : список\_базовых\_классов {список\_компонентов\_класса};

**Конструкторы и деструкторы при наследовании**

Поскольку конструкторы не наследуются, при создании производного класса наследуемые им данные-члены должны инициализироваться конструктором базового класса. Конструктор базового класса вызывается автоматически и выполняется до конструктора производного класса. Параметры конструктора базового класса указываются определении конструктора производного класса. Таким образом, происходит передача аргументов от конструктора производного класса конструктору базового класса.

//базовый класс

class Base

{

protected:

int a,b;

public:

Base(int x,int y){a=x;b=y;}

};

//производный класс

class Derive:public Base

{

protected:

int sum;

public:

Derive (int x,int y, int s):Base(x,y){sum=s;}

};

Объекты класса конструируются снизу вверх: сначала базовый, потом компоненты-объекты (если они имеются), а потом сам производный класс. Таким образом, объект производного класса содержит в качестве подобъекта объект базового класса.

Уничтожаются объекты в обратном порядке: сначала производный, потом его компоненты-объекты, а потом базовый объект.

Таким образом, порядок уничтожения объекта противоположен по отношению к порядку его конструирования.

**Виртуальные функции**

К механизму виртуальных функций обращаются в тех случаях, когда в каждом производном классе требуется свой вариант некоторой компонентной функции. Классы, включающие такие функции, называются полиморфными и играют особую роль в ООП.

Виртуальные функции предоставляют механизм позднего (отложенного) или динамического связывания. Любая нестатическая функция базового класса может быть сделана виртуальной, для чего используется ключевое слово virtual.

class Base

{

public:

virtual void print(){cout<<”\nBase”;}

. . .

};

class Derive : public Base

{

public:

void print(){cout<<”\n Derive”;}

};

void main()

{

Base B,\*bp;

Derive D,\*dp;

bp=&B;

dp=&D;

//указатель базового класса ставится на объект //производного класса

Base \*p = &D;

bp –>print(); // вызывается метод для Base

dp –>print(); // вызывается метод для Derive p –>print(); // вызывается метод для Derive

}

Таким образом, интерпретация каждого вызова виртуальной функции через указатель на базовый класс зависит от значения этого указателя, то есть от типа объекта, для которого выполняется вызов.

Выбор того, какую виртуальную функцию вызвать, будет зависеть от типа объекта, на который фактически (в момент выполнения программы) направлен указатель, а не от типа указателя.

Виртуальными могут быть только нестатические функции-члены.

Виртуальность наследуется. После того как функция определена как виртуальная,

повторное определение в производном классе (с тем же самым прототипом) создает в этом классе новую виртуальную функцию, причем спецификатор virtual может не использоваться.

Конструкторы не могут быть виртуальными, в отличие от деструкторов. Практически каждый класс, имеющий виртуальную функцию, должен иметь виртуальный деструктор.

**Принцип подстановки**

Открытое наследование устанавливает между классами отношение «является»: класс-наследник является частью класса-родителя. Это означает, что везде, где может быть использован объект базового класса (при присваивании, при передаче параметров и возврате результата), вместо него разрешается использовать объект производного класса. Данное положение называется принципом подстановки. Он работает и для ссылок и для указателей. Обратное неверно. Например, всякий спортсмен (производный класс) является человеком (базовый класс), но не всякий человек является спортсменом.

Закрытое наследование – это наследование реализации, в этом случае принцип подстановки не соблюдается.

class Base

{

public:

void f1();

void f2();

};

class Derive: private Base //закрытое наследование {……};

Программа, использующая класс Derive не может использовать ни f1() ни f2(). В наследнике заново реализуются все методы:

class Derive: private Base

{

public:

void f1(){Base::f1();};

void f2(){Base::f2();};

};

### Постановка задачи

1. Определить пользовательский класс.

2. Определить в классе следующие конструкторы: без параметров, с параметрами, копирования.

3. Определить в классе деструктор.

4. Определить в классе компоненты-функции для просмотра и установки полей данных (селекторы и модификаторы).

5. Перегрузить операцию присваивания.

6. Перегрузить операции ввода и вывода объектов с помощью потоков.

7. Определить производный класс.

8. Написать программу, в которой продемонстрировать создание объектов и работу всех перегруженных операций.

9. Реализовать функции, получающие и возвращающие объект базового класса. Продемонстрировать принцип подстановки.

### Варианты заданий

|  |  |
| --- | --- |
| №  варианта | Задание |
| 1 | Базовый класс:  ПАРА\_ЧИСЕЛ (PAIR)  Первое\_число (first) – int  Второе\_число (second) – int  Определить методы изменения полей и сравнения пар (пара p1 больше пары р2, если (p1.first>p2.first) || (p1.first==p2.first &&p1.second>p2.second).  Создать производный класс ДРОБЬ (FRACTION), с полями Целая\_часть\_числа и Дробная\_часть\_числа. Определить полный набор методов сравнения. |
| 2 | Базовый класс:  ПАРА\_ЧИСЕЛ (PAIR)  Первое\_число (first) – int  Второе\_число (second) – int  Определить методы изменения полей и вычисления произведения чисел.  Создать производный класс ПРЯМОУГОЛЬНИК(RECTANGLE), с полями-сторонами. Определить методы для вычисления площади и периметра прямоугольника. |
| 3 | Базовый класс:  ПАРА\_ЧИСЕЛ (PAIR)  Первое\_число (first) – int  Второе\_число (second) – int  Определить методы изменения полей и вычисления произведения чисел.  Создать производный класс ПРЯМОУГОЛЬНЫЙ\_ТРЕУГОЛЬНИК (RIGHTANGLED), с полями-катетами. Определить метод вычисления гипотенузы. |
| 4 | Базовый класс:  ПАРА\_ЧИСЕЛ (PAIR)  Первое\_число (first) – int  Второе\_число (second) – int  Определить методы изменения полей и вычисления произведения чисел.  Создать производный класс ЭЛЛИПС (ELLIPSE), с полями: малой и большой осями. Определить метод вычисления площади. |
| 5 | Базовый класс:  ПАРА\_ЧИСЕЛ (PAIR)  Первое\_число (first) – int  Второе\_число (second) – int  Определить методы изменения полей и операцию сложения пар (a,b)+(c,d)=(a+b,c+d)  Создать производный класс КОМПЛЕКСНОЕ\_ЧИСЛО(COMPLEX), с полями Действительная\_часть\_числа и Мнимая\_часть\_числа. Определить операции умножения (a,b)\*(c,d)= (a\*c-b\*d, a\*d+b\*c) и вычитания (a,b)-(c,d)= (a-b, с-d) |
| 6 | Базовый класс:  ПАРА\_ЧИСЕЛ (PAIR)  Первое\_число (first) – int  Второе\_число (second) – int  Определить методы изменения полей и операцию сложения пар (a,b)+(c,d)=(a+b,c+d)  Создать производный класс ДЕНЕЖНАЯ\_СУММА(MONEY), с полями Рубли и Копейки. Переопределить операции сложения и определить операции вычитания и деления денежных сумм. |
| 7 | Базовый класс:  ПАРА\_ЧИСЕЛ (PAIR)  Первое\_число (first) – int  Второе\_число (second) – int  Определить методы изменения полей и операцию сложения пар (a,b)+(c,d)=(a+b,c+d)  Создать производный класс ДЛИННОЕ\_ЧИСЛО(LONG), с полями Старшая\_часть\_числа и Младшая\_часть\_числа. Переопределить операцию сложения и определить операции вычитания и умножения. |
| 8 | Базовый класс:  ПАРА\_ЧИСЕЛ (PAIR)  Первое\_число (first) – int  Второе\_число (second) – int  Определить методы проверки на равенство и операцию перемножения полей. Реализовать операцию вычитания пар по формуле (a,b)-(c,d)=(a-b,c-d)  Создать производный класс ПРОСТАЯ\_ДРОБЬ(RATIONAL), с полями Числитель и Знаменатель. Переопределить операцию вычитания и определить операции сложения и умножения простых дробей. |
| 9 | Базовый класс:  ПАРА\_ЧИСЕЛ (PAIR)  Первое\_число (first) – int  Второе\_число (second) – int  Определить методы проверки на равенство и операцию перемножения полей. Реализовать операцию сложения пар по формуле (a,b)+(c,d)=(a+b,c+d)  Создать производный класс СТЕПЕННАЯ\_ФУНКЦИЯ(POWER\_FUNCTION), с полями Основание и Показатель\_степени. Переопределить операцию сложения и определить операции сложения и умножения степенных функций. |
| 10 | Базовый класс:  ПАРА\_ЧИСЕЛ (PAIR)  Первое\_число (first) – int  Второе\_число (second) – int  Определить методы проверки на равенство и операцию перемножения полей. Реализовать операцию сложения пар по формуле (a,b)+(c,d)=(a+b,c+d)  Создать производный класс КОРЕНЬ(ROOT), с полями Основание и Показатель\_корня. Переопределить операцию сложения и определить операции сложения и умножения корней. |
| 11 | Базовый класс:  ПАРА\_ЧИСЕЛ (PAIR)  Первое\_число (first) – int  Второе\_число (second) – int  Определить методы проверки на равенство и операцию перемножения полей. Реализовать операцию вычитания пар по формуле (a,b)-(c,d)=(a-b,c-d)  Создать производный класс ЛОГАРИФМ(LOGARITHM), с полями Основание и Логарифмируемое\_число. Переопределить операцию вычитания и определить операции сложения и умножения логарифмических функций. |
| 12 | Базовый класс:  ТРОЙКА\_ЧИСЕЛ (TRIAD)  Первое\_число (first) – int  Второе\_число (second) – int  Третье\_число (third) – int  Определить методы изменения полей и сравнения триады.  Создать производный класс DISTANCE с полями километр, метр и дециметр. Определить полный набор операций сравнения расстояний. |
| 13 | Базовый класс:  ТРОЙКА\_ЧИСЕЛ (TRIAD)  Первое\_число (first) – int  Второе\_число (second) – int  Третье\_число (third) – int  Определить методы изменения полей и сравнения триады.  Создать производный класс DATE с полями год, месяц и число. Определить полный набор операций сравнения дат. |
| 14 | Базовый класс:  ТРОЙКА\_ЧИСЕЛ (TRIAD)  Первое\_число (first) – int  Второе\_число (second) – int  Третье\_число (third) – int  Определить методы изменения полей и сравнения триады.  Создать производный класс TIME с полями часы, минуты и секунды. Определить полный набор операций сравнения временных промежутков. |
| 15 | Базовый класс:  ТРОЙКА\_ЧИСЕЛ (TRIAD)  Первое\_число (first) – int  Второе\_число (second) – int  Третье\_число (third) – int  Определить методы изменения полей и увеличения полей на 1.  Создать производный класс DATE с полями год, месяц и число. Переопределить методы увеличения полей на 1 и определить метод увеличения даты на n дней. |
| 16 | Базовый класс:  ТРОЙКА\_ЧИСЕЛ (TRIAD)  Первое\_число (first) – int  Второе\_число (second) – int  Третье\_число (third) - int  Определить методы изменения полей и увеличения полей на 1.  Создать производный класс TIME с полями часы, минуты и секунды. Переопределить методы увеличения полей на 1 и определить методы увеличения на n секунд и минут. |
| 17 | Базовый класс:  ШЕСТЕРКА\_ЧИСЕЛ (TRIAD)  Первое\_число (first) – int  Второе\_число (second) – int  Третье\_число (third) – int  Четвертое\_число (fourth) – int  Пятое\_число (fifth) – int  Шестое\_число (sixth) - int  Определить методы изменения полей и увеличения полей на 1.  Создать производный класс FULL\_DATE с полями часы, минуты, секунды, день, месяц, год. Переопределить методы увеличения полей на 1 и определить методы увеличения на n секунд и дней. |
| 18 | Базовый класс:  ЧЕЛОВЕК (PERSON)  Имя (name) – string  Возраст (age) – int  Определить методы изменения полей.  Создать производный класс STUDENT, имеющий поле год обучения. Определить методы изменения и увеличения года обучения. |
| 19 | Базовый класс:  ЧЕЛОВЕК (PERSON)  Имя (name) – string  Возраст (age) – int  Определить методы изменения полей.  Создать производный класс EMPLOYEE, имеющий поля Должность – string и Оклад – double. Определить методы изменения полей и вычисления зарплаты сотрудника по формуле Оклад+Премия (% от оклада). |
| 20 | Базовый класс:  ЧЕЛОВЕК (PERSON)  Имя (name) – string  Возраст (age) – int  Определить методы изменения полей.  Создать производный класс TEACHER, имеющий поля Предмет – string и Количество часов – int. Определить методы изменения полей, а также увеличения и уменьшения часов. |
| 21 | Базовый класс:  ЧЕЛОВЕК (PERSON)  Имя (name) – string  Возраст (age) – int  Определить методы изменения полей.  Создать производный класс STUDENT, имеющий поля Предмет – string и Оценка – int. Определить методы изменения полей и метод, выдающий сообщение о неудовлетворительной оценке. |

### Содержание отчета

1. Постановка задачи (общая и конкретного варианта)

2. Анализ задачи

* Описание класса
* Определение компонентных функций
* Определение глобальных функций
* Функция main()
* Объяснение результатов работы программы

3. Блок-схемы основных функций

4. Текст программы

5. Тесты

## Лабораторная работа № 4

## Шаблоны классов



### Цель работы:

1. Создание консольного приложения, состоящего из нескольких файлов в системе программирования Visual Studio.

2. Реализация шаблона класса-контейнера.

### Краткие теоретические сведения

С помощью шаблона функций можно отделить алгоритм от конкретных типов данных, передавая тип в качестве параметра. Шаблоны классов предоставляют аналогичную возможность, позволяя создавать параметризированные классы. Параметризированный класс создает семейство родственных классов, которые можно применять к любому типу данных, передаваемому в качестве параметра. Если использовать в качестве параметризированного класса контейнер, то такой контейнер можно будет применять к любым типам данных, не переписывая код.

**Шаблоны функций**

Шаблоны вводятся для того, чтобы автоматизировать создание функций, обрабатывающих разнотипные данные. Например, алгоритм сортировки можно использовать для массивов различных типов. При перегрузке функции для каждого используемого типа определяется своя функция. Шаблон функции определяется один раз, но определение параметризируется, т. е. тип данных передается как параметр шаблона. Формат шаблона:

template <параметры\_шаблона>

заголовок\_функции

{тело функции}

Таким образом, шаблон семейства функций состоит из 2 частей – заголовка шаблона: template<список параметров шаблона> и обыкновенного определения функции, котором вместо типа возвращаемого значения и/или типа параметров, записывается имя типа, определенное в заголовке шаблона.

Пример.

//шаблон функции, которая находит абсолютное значение числа любого типа template<class type>//type – имя параметризируемого типа

type abs(type x)

{

if(x<0)return -x; else return x;

}

Шаблон служит для автоматического формирования конкретных описаний функций по тем вызовам, которые компилятор обнаруживает в программе. Например, если в программе вызов функции осуществляется как abs(-1.5), то компилятор сформирует определение функции double abs(double x){. . . }.

**Шаблоны классов**

Шаблоны классов так же как и шаблоны функций поддерживают парадигму обобщенного программирования, т. е. программирования с использованием типов в качестве параметров. Механизм шаблонов в С++ допускает применение абстрактного типа в качестве параметра при определении класса. После того как шаблон класса определен, он может использоваться для определения конкретных классов. Процесс генерации компилятором определения конкретного класса по шаблону класса и аргументам шаблона называется инстанцированием шаблона. Определение шаблонного (обобщенного) класса имеет вид:

template <параметры шаблона>

class имя\_класса

{…};

Пример:

template<class T>

class Point

{

x,y;//координаты точки public:

Point(T X=0,T Y=0):x(X),y(Y){} void Show ();

};

template<class T>

void Point::Show()

{

cout<<”(“<<x<<” , ”<<y<<”)”;

}

};

При включении шаблона класса в программу никакие классы на самом деле не генерируются до тех пор, пока не будет создан экземпляр шаблонного класса, в котором вместо параметра шаблона указывается конкретный тип. Экземпляр создается либо объявлением объекта, либо объявлением указателя на инстанцированный шаблонный тип присваиванием ему адреса с помощью операции new.

Point <int> a(13,15);

Point <float>\*pa=new Point<float>(10.1,0.55);

Встретив такие объявления, компилятор генерирует код исходного класса.

В проекте, состоящем из нескольких файлов, определение шаблона класса обычно выносится в отдельный файл. Но для того, чтобы инстанцировался конкретный экземпляр шаблона класса необходимо, чтобы определение шаблона находилось в одной единице трансляции с этим экземпляром. Поэтому все определение шаблонного класса размещается в заголовочном файле, а затем этот файл подключается к нужным файлам с помощью директивы include. Чтобы этот файл не включался повторно, используется директива ifndef.

Пример.

//Point.h

#ifndef POINT\_H

#define POINT\_H

template<class T>

class Point

{

T x,y;//координаты точки public:

Point(T X=0,T Y=0):x(X),y(Y){} void Show () const;

};

template<class T>

void Point::Show()

{

cout<<”(“<<x<<” , ”<<y<<”)”;

}

};

#endif

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

//main.cpp

#include “Point.h”

…..

void main()

{

Point<double>p1;

Point<int>p2(1,1);

p1.Show();

p2.Show();

}

**Правила описания шаблонов**

* Шаблоны методов (функций) не могут быть виртуальными.
* Шаблоны классов могут содержать статические элементы, дружественные функции и классы.
* Шаблоны могут быть производными как от шаблонов, так и от обычных классов, а также являться базовыми и для шаблонов, и для обычных классов.

### Постановка задачи

* 1. Определить шаблон класса-контейнера.
  2. Реализовать конструкторы, деструктор, операции ввода-вывода, операцию присваивания.
  3. Перегрузить операции, указанные в варианте.
  4. Инстанцировать шаблон для стандартных типов данных (int, float, double).
  5. Написать тестирующую программу, иллюстрирующую выполнение операций для контейнера, содержащего элементы стандартных типов данных.
  6. Реализовать пользовательский класс.
  7. Перегрузить для пользовательского класса операции ввода-вывода.
  8. Перегрузить операции необходимые для выполнения операций контейнерного класса.
  9. Инстанцировать шаблон для пользовательского класса.
  10. Написать тестирующую программу, иллюстрирующую выполнение операций для контейнера, содержащего элементы пользовательского класса.

### Варианты заданий

|  |  |
| --- | --- |
| №  варианта | Задание |
| 1 | Класс-контейнер ВЕКТОР с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  () – определение размера вектора;  + число – добавляет константу ко всем элементам вектора;  Пользовательский класс Time для работы с временными интервалами. Интервал должен быть представлен в виде двух полей: минуты типа int и секунды типа int. При выводе минуты отделяются от секунд двоеточием. |
| 2 | Класс-контейнер ВЕКТОР с элементами типа int.  Реализовать операции:  []– доступа по индексу;  int() – определение размера вектора;  + вектор – сложение элементов векторов a[i]+b[i];  Пользовательский класс Time для работы с временными интервалами. Интервал должен быть представлен в виде двух полей: минуты типа int и секунды типа int. При выводе минуты отделяются от секунд двоеточием. |
| 3 | Класс-контейнер ВЕКТОР с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  + вектор – сложение элементов векторов a[i]+b[i];  + число – добавляет константу ко всем элементам вектора;  Пользовательский класс Time для работы с временными интервалами. Интервал должен быть представлен в виде двух полей: минуты типа int и секунды типа int. При выводе минуты отделяются от секунд двоеточием. |
| 4 | Класс-контейнер ВЕКТОР с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  () – определение размера вектора;  \* число – умножает все элементы вектора на число;  Пользовательский класс Time для работы с временными интервалами. Интервал должен быть представлен в виде двух полей: минуты типа int и секунды типа int. При выводе минуты отделяются от секунд двоеточием. |
| 5 | Класс-контейнер ВЕКТОР с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  int() – определение размера вектора;  \* вектор – умножение элементов векторов a[i]\*b[i];  Пользовательский класс Time для работы с временными интервалами. Интервал должен быть представлен в виде двух полей: минуты типа int и секунды типа int. При выводе минуты отделяются от секунд двоеточием. |
| 6 | Класс-контейнер ВЕКТОР с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  int() – определение размера вектора;  / вектор – деление элементов векторов a[i]/b[i];  -n – переход влево к элементу с номером n (с помощью класса-итератора).  Пользовательский класс Time для работы с временными интервалами. Интервал должен быть представлен в виде двух полей: минуты типа int и секунды типа int. При выводе минуты отделяются от секунд двоеточием. |
| 7 | Класс-контейнер ВЕКТОР с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  - вектор – вычитание элементов векторов a[i]-b[i];  \* число – умножает на константу все элементам вектора;  ++ - переход к следующему элементу (с помощью класса-итератора).  Пользовательский класс Time для работы с временными интервалами. Интервал должен быть представлен в виде двух полей: минуты типа int и секунды типа int. При выводе минуты отделяются от секунд двоеточием. |
| 8 | Класс-контейнер МНОЖЕСТВО с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  () – определение размера множества;  + – объединение множеств;  Пользовательский класс Money для работы с денежными суммами. Число должно быть представлено двумя полями: типа long для рублей и типа int для копеек. Дробная часть числа при выводе на экран должна быть отделена от целой части запятой. |
| 9 | Класс-контейнер МНОЖЕСТВО с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  int() – определение размера вектора;  \* – пересечение множеств;  Пользовательский класс Money для работы с денежными суммами. Число должно быть представлено двумя полями: типа long для рублей и типа int для копеек. Дробная часть числа при выводе на экран должна быть отделена от целой части запятой. |
| 10 | Класс-контейнер МНОЖЕСТВО с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  == – проверка на равенство;  > число – принадлежность числа множеству;  Пользовательский класс Money для работы с денежными суммами. Число должно быть представлено двумя полями: типа long для рублей и типа int для копеек. Дробная часть числа при выводе на экран должна быть отделена от целой части запятой. |
| 11 | Класс-контейнер МНОЖЕСТВО с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  != - проверка на неравенство;  < число – принадлежность числа множеству;  Пользовательский класс Money для работы с денежными суммами. Число должно быть представлено двумя полями: типа long для рублей и типа int для копеек. Дробная часть числа при выводе на экран должна быть отделена от целой части запятой. |
| 12 | Класс-контейнер МНОЖЕСТВО с элементами типа int. Реализовать операции:  [] – доступа по индексу;  () – определение размера вектора;  - – разность множеств;  Пользовательский класс Money для работы с денежными суммами. Число должно быть представлено двумя полями: типа long для рублей и типа int для копеек. Дробная часть числа при выводе на экран должна быть отделена от целой части запятой. |
| 13 | Класс-контейнер МНОЖЕСТВО с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  == - проверка на равенство;  < число – принадлежность числа множеству;  -n – переход влево к элементу с номером n (с помощью класса-итератора).  Пользовательский класс Money для работы с денежными суммами. Число должно быть представлено двумя полями: типа long для рублей и типа int для копеек. Дробная часть числа при выводе на экран должна быть отделена от целой части запятой. |
| 14 | Класс-контейнер МНОЖЕСТВО с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  int() – определение размера множества;  + – объединение множеств;  -- - переход к предыдущему элементу (с помощью класса-итератора).  Пользовательский класс Money для работы с денежными суммами. Число должно быть представлено двумя полями: типа long для рублей и типа int для копеек. Дробная часть числа при выводе на экран должна быть отделена от целой части запятой. |
| 15 | Класс-контейнер СПИСОК с ключевыми значениями типа int. Реализовать операции:  []– доступа по индексу;  int() – определение размера списка;  + список – сложение элементов списков a[i]+b[i];  Пользовательский класс Money для работы с денежными суммами. Число должно быть представлено двумя полями: типа long для рублей и типа int для копеек. Дробная часть числа при выводе на экран должна быть отделена от целой части запятой. |
| 16 | Класс-контейнер СПИСОК с ключевыми значениями типа int. Реализовать операции:  [] – доступа по индексу;  () – определение размера вектора;  + число – добавляет константу ко всем элементам вектора;  Пользовательский класс Pair (пара чисел). Пара должна быть представлена двумя полями: типа int для первого числа и типа double для второго. Первое число при выводе на экран должно быть отделено от второго числа двоеточием. |
| 17 | Класс- контейнер СПИСОК с ключевыми значениями типа int. Реализовать операции:  [] – доступа по индексу;  + список – сложение элементов списков a[i]+b[i];  + число – добавляет константу ко всем элементам списка;  Пользовательский класс Pair (пара чисел). Пара должна быть представлено двумя полями: типа int для первого числа и типа double для второго. Первое число при выводе на экран должно быть отделено от второго числа двоеточием. |
| 18 | Класс-контейнер СПИСОК с ключевыми значениями типа int. Реализовать операции:  [] – доступа по индексу;  () – определение размера списка;  \* число – умножает все элементы списка на число;  Пользовательский класс Pair (пара чисел). Пара должна быть представлено двумя полями: типа int для первого числа и типа double для второго. Первое число при выводе на экран должно быть отделено от второго числа двоеточием. |
| 19 | Класс-контейнер СПИСОК с ключевыми значениями типа int. Реализовать операции:  [] – доступа по индексу;  int() – определение размера списка;  \* список – умножение элементов списков a[i]\*b[i];  Пользовательский класс Pair (пара чисел). Пара должна быть представлено двумя полями: типа int для первого числа и типа double для второго. Первое число при выводе на экран должно быть отделено от второго числа двоеточием. |
| 20 | Класс-контейнер СПИСОК с ключевыми значениями типа int.  Реализовать операции:  [] – доступа по индексу;  \* список – перемножение элементов списков a[i]+b[i];  - число – вычитает константу из всех элементам списка;  -n - переход влево к элементу с номером n (с помощью класса-итератора).  Пользовательский класс Pair (пара чисел). Пара должна быть представлено двумя полями: типа int для первого числа и типа double для второго. Первое число при выводе на экран должно быть отделено от второго числа двоеточием. |

### Содержание отчета

1. Постановка задачи (общая и конкретного варианта)

2. Анализ задачи

* Описание параметризированного класса-контейнера
* Определение компонентных функций
* Описание пользовательского класса и его компонентных функций
* Функция main()
* Объяснение результатов работы программы

3. Блок-схемы основных функций

4. Текст программы

5. Тесты

## Лабораторная работа № 5

## Шаблоны функций



### Цель работы:

Получить практические навыки работы с шаблонами функций.

### Шаблоны функций

Шаблоны вводятся для того, чтобы автоматизировать создание функций, обрабатывающих разнотипные данные. Например, алгоритм сортировки можно использовать для массивов различных типов. При перегрузке функции для каждого используемого типа определяется своя функция. Шаблон функции определяется один раз, но определение параметризируется, т. е. тип данных передается как параметр шаблона.

template <class имя\_типа [,class имя\_типа]>

заголовок\_функции

{

тело функции

}

Таким образом, шаблон семейства функций состоит из 2 частей – заголовка шаблона: template<список параметров шаблона> и обыкновенного определения функции, в котором вместо типа возвращаемого значения и/или типа параметров, записывается имя типа, определенное в заголовке шаблона.

//сравнение двух чисел любого типа

template<class T>

T max(T a, T b)

{

if (a>b) return a;

else return b;

}

Шаблон служит для автоматического формирования конкретных описаний функций по тем вызовам, которые компилятор обнаруживает в программе. Например, если в программе вызов функции осуществляется как max(1,5), то компилятор сформирует определение функции int max(int a, int b){…}.

### Постановка задачи

Написать шаблон функций для своего варианта. Написать демонстрационную программу для вызова этих функций.

### Варианты заданий

1. Среднее арифметическое массива
2. Количество отрицательных элементов в массиве
3. Максимальный элемент в массиве
4. Минимальный элемент в массиве
5. Сортировка массива методом простого обмена
6. Сортировка массива методом простого выбора
7. Сортировка массива методом простого включения
8. Поиск заданного элемента в массиве
9. Поиск заданного элемента в отсортированном массиве
10. Удаление элемента с заданным номером из динамического массива
11. Удаление элемента с заданным ключом из динамического массива
12. Добавление элемента с заданным номером в динамический массив
13. Добавление элемента после элемента с заданным номером в динамический массив
14. Номер максимального элемента в массиве
15. Среднее арифметическое массива
16. Количество отрицательных элементов в массиве
17. Добавление элемента с заданным номером в динамический массив
18. Сортировка массива методом простого обмена
19. Минимальный элемент в массиве
20. Сортировка массива методом простого выбора
21. Сортировка массива методом простого включения
22. Поиск заданного элемента в массиве
23. Поиск заданного элемента в отсортированном массиве
24. Удаление элемента с заданным номером из динамического массива
25. Удаление элемента с заданным ключом из динамического массива

### Методические указания

1. Использовать в программе классы, так что функция должна принадлежать классу объектов, с которыми будет работать эта функция.

2. Инстанцировать шаблон функции для типов char, int, и double.

### Содержание отчета

1. Постановка задачи (общая и для конкретного варианта)

2. Анализ задачи

* Определения функций для реализации поставленных задач
* Определение функции main()

3. Текст программы

4. Тесты

## Лабораторная работа № 6

## Обработка исключительных ситуаций



### Цель работы:

1. Создание консольного приложения, состоящего из нескольких файлов в системе программирования Visual Studio.
2. Разработка программы, обрабатывающей исключительные ситуации.

### Краткие теоретические сведения

**Механизм обработки исключений**

Исключение – это непредвиденное или аварийное событие.

В С++ исключение – это объект, который система должна генерировать при возникновении исключительной ситуации. Генерация такого объекта и создает исключительную ситуацию.

Исключения позволяют разделить вычислительный процесс на 2 части:

1. обнаружение аварийной ситуации (неизвестно как обрабатывать);

2. обработка аварийной ситуации (неизвестно, где она возникла). Достоинства такого подхода:

1. удобно использовать в программе, которая состоит из нескольких модулей;

2. не требуется возвращать значение в вызывающую функцию

Общая схема:

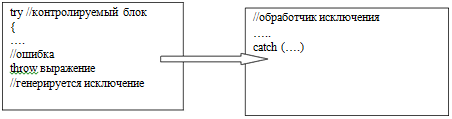


Рисунок 6.1 – общая схема

Исключение генерируется оператором throw <выражение>, где <выражение> - либо константа, либо переменная некоторого типа, либо выражение некоторого типа.

Тип объекта-исключения может быть как встроенным, так и определяемым пользователем. Для представления исключений часто используют пустой класс:

class ZeroDevide{};

class NegativeArg{};

Генерация исключения будет выглядеть:

throw ZeroDevide();//вызывается конструктор без параметров

или

throw new ZeroDevide();

Исключение надо перехватить и обработать. Для проверки возникновения исключения используется контролируемый блок try{}, с которым связана одна или несколько секций-ловушек catch. Все переменные, объявленные в этом блоке, являются локальными для этого блока.

Форма записи секции-ловушки следующая:

catch( спецификация исключения ), где спецификация исключения может иметь три формы:

(тип имя)

(тип)

(…)

Тип – это встроенный тип или тип, определенный программистом.

Формы 1 и 2 обрабатывают конкретные исключения, а форма 3 перехватывает все исключения, такую ловушку надо помещать последней, тогда она будет обрабатывать все исключения, которые еще не были обработаны.

Форма 1 означает, что объект передается в блок обработки, чтобы его каким-то образом там использовать, например, для вывода информации в сообщении об ошибке. Примеры:

catch( exception e) // по значению catch( exception &e) // по ссылке

catch( const exception &e) // по константной ссылке catch( exception \*e) //по указателю

Лучше всего передавать объект по ссылке, т. к. при этом не создается временный объект-исключение.

Для каждой функции, метода, конструктора или деструктора можно в заголовке указать спецификацию исключений. Если в заголовке спецификация исключений не указана, считается, что функция может порождать любое исключение, если указана, то считается, что функция генерирует те исключения, которые явно указаны в этом списке.

Примеры:

void f1()throw(int,double);

void f2()throw(ZeroDivide);

Если спецификация имеет вид:

void f()throw();

то считается, что функция исключений не генерирует.

Наличие спецификаций исключения не является ограничением при реальной генерации исключений. Но если функция генерирует неспецифицированное исключение, то запускается стандартная функция unexpected(), которая вызывает функцию terminate(), что приводит к аварийному завершению программы.

При отсутствии подходящей секции-ловушки осуществляется вызов стандартной функции завершения terminate(). Эта функция вызывается из функции unexpected() при нарушении спецификации завершения.

Обе функции можно подменить собственными реализациями. Для этого необходимо

1. Подключить заголовок #include <expection>

2. Определить собственную функцию с прототипом void F() для подмены стандартной функции terminate().

3. Указать имя этой функции в вызове функции set\_terminate(F);

После этого вместо terminate() будет вызываться наша функция F(). Такая функция не должна возвращать управление оператором return или генерировать исключение throw(),она может только завершить программу функцией exit() или abort(). Аналогично реализуется подмена стандартной функции unexpected():

set\_ unexpected(F);

Функция может сгенерировать неспецифицированное исключение, в этом случае, если в спецификации исключений не указано исключение bad\_exception, вызывается функция terminate(), в противном случае сгенерированное исключение подменяется на bad\_exception и начинается поиск его обработчика.

**Стандартные исключения**

В составе стандартной библиотеки С++ реализован ряд стандартных исключений, которые организованы в иерархию классов.

Эта иерархия может служить основой для создания собственных классов исключений и иерархии исключений. Можно определять собственные исключения, унаследовав их от класса exception.

Класс exception определен в стандартной библиотеке следующим образом:

class exception

{

public:

exception () throw();//конструктор без параметров

exception (const exception&) throw();//конструктор копирования exception& operator= (const exception&) throw();//оператор = virtual ~exception() throw();//деструктор

virtual const char\*what() const throw();//генерирует сообщение об ошибке

};

Все конструкторы и методы имеют спецификацию, запрещающую генерацию исключений.

Предполагается, что исключения типа

logic\_error - ошибки в логике программы, например, невыполнение какого-либо условия;

runtime\_error – ошибки возникают в результате непредвиденных обстоятельств при выполнении программы, например, переполнение при операциях с дробными числами;

Эти исключения программа должна генерировать самостоятельно оператором throw. Пять стандартных исключений порождают различные механизмы С++.

bad\_alloc генерирует операция new, если не может быть выделена память

bad\_cast и bad\_typied генерируются при динамической идентификации типов (RTTI)

ios\_base::failure генерируется системой ввода/вывода.

bad\_exception генерируется, если спецификация исключений содержит

bad\_exception.

**Создание собственной иерархии исключений**

Для создания собственной иерархии исключений надо объявить свой базовый класс-исключение, например:

class BaseException{};

Остальные классы будут наследниками этого класса, аналогично тому, как это сделано в иерархии стандартных исключений:

class Child\_Exception1:public BaseException{}; class Child\_Exception2:public BaseException{};

Класс BaseException можно унаследовать от стандартного класса exception class BaseException: public exception{};

Наследование от стандартных классов позволит использовать метод what для вывода сообщений об ошибках.

Иерархия классов-исключений позволяет вместо нескольких разных блоков-ловушек написать единственный блок с типом аргумента базового класса.

### Постановка задачи

1. Реализовать класс, перегрузить для него операции, указанные в варианте.

2. Определить исключительные ситуации.

3. Предусмотреть генерацию исключительных ситуаций.

### Варианты заданий

|  |  |  |
| --- | --- | --- |
| № | Задание | Вариант реализации |
| 1 | Класс-контейнер ВЕКТОР с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  () – определение размера вектора;  + число – добавляет константу ко всем элементам вектора;  ++ - переход к следующему элементу (с помощью класса-итератора). | 1, 2 |
| 2 | Класс-контейнер ВЕКТОР с элементами типа int.  Реализовать операции:  []– доступа по индексу;  int() – определение размера вектора;  + вектор – сложение элементов векторов a[i]+b[i];  +n - переход вправо к элементу c номером n (с помощью класса-итератора). | 2, 3 |
| 3 | Класс-контейнер ВЕКТОР с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  + вектор – сложение элементов векторов a[i]+b[i];  + число – добавляет константу ко всем элементам вектора;  -- - переход к предыдущему элементу (с помощью класса-итератора). | 3, 1 |
| 4 | Класс-контейнер ВЕКТОР с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  () – определение размера вектора;  \* число – умножает все элементы вектора на число;  -n – переход влево к элементу с номером n (с помощью класса-итератора). | 1, 2 |
| 5 | Класс-контейнер ВЕКТОР с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  int() – определение размера вектора;  \* вектор – умножение элементов векторов a[i]\*b[i];  +n – переход вправо к элементу с номером n (с помощью класса-итератора). | 2, 3 |
| 6 | Класс-контейнер ВЕКТОР с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  int() – определение размера вектора;  / вектор – деление элементов векторов a[i]/b[i];  -n – переход влево к элементу с номером n (с помощью класса-итератора). | 3, 1 |
| 7 | Класс-контейнер ВЕКТОР с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  - вектор – вычитание элементов векторов a[i]-b[i];  \* число – умножает на константу все элементам вектора;  ++ - переход к следующему элементу (с помощью класса-итератора). | 1, 2 |
| 8 | Класс-контейнер МНОЖЕСТВО с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  () – определение размера множества;  + – объединение множеств;  ++ - переход к следующему элементу (с помощью класса-итератора). | 2, 3 |
| 9 | Класс-контейнер МНОЖЕСТВО с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  int() – определение размера множества;  \* – пересечение множеств;  -- - переход к предыдущему элементу (с помощью класса-итератора). | 3, 1 |
| 10 | Класс-контейнер МНОЖЕСТВО с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  == - проверка на равенство;  > число – принадлежность числа множеству;  -n - переход влево к элементу с номером n (с помощью класса-итератора). | 1, 2 |
| 11 | Класс-контейнер МНОЖЕСТВО с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  != - проверка на неравенство;  < число – принадлежность числа множеству;  +n – переход вправо к элементу с номером n (с помощью класса-итератора). | 2, 3 |
| 12 | Класс-контейнер МНОЖЕСТВО с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  () – определение размера множества;  - – разность множеств;  -- – переход к предыдущему элементу ( с помощью класса-итератора). | 1, 3 |
| 13 | Класс-контейнер МНОЖЕСТВО с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  == - проверка на равенство;  < число – принадлежность числа множеству;  -n – переход влево к элементу с номером n (с помощью класса-итератора). | 1, 2 |
| 14 | Класс-контейнер МНОЖЕСТВО с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  int() – определение размера множества;  + – объединение множеств;  -- - переход к предыдущему элементу (с помощью класса-итератора). | 2, 3 |
| 15 | Класс-контейнер СПИСОК с ключевыми значениями типа int.  Реализовать операции:  []– доступа по индексу;  int() – определение размера списка;  + список – сложение элементов списков a[i]+b[i];  -n - переход влево к элементу c номером n (с помощью класса-итератора). | 3, 1 |
| 16 | Класс-контейнер СПИСОК с ключевыми значениями типа int.  Реализовать операции:  [] – доступа по индексу;  () – определение размера вектора;  + число – добавляет константу ко всем элементам вектора;  ++ - переход к следующему элементу (с помощью класса-итератора). | 1, 2 |
| 17 | Класс-контейнер СПИСОК с ключевыми значениями типа int.  Реализовать операции:  [] – доступа по индексу;  + список – сложение элементов списков a[i]+b[i];  + число – добавляет константу ко всем элементам списка;  -- - переход к предыдущему элементу (с помощью класса-итератора). | 2, 3 |
| 18 | Класс-контейнер СПИСОК с ключевыми значениями типа int.  Реализовать операции:  [] – доступа по индексу;  () – определение размера списка;  \* число – умножает все элементы списка на число;  -n – переход влево к элементу с номером n (с помощью класса-итератора). | 3, 1 |
| 19 | Класс-контейнер СПИСОК с ключевыми значениями типа int.  Реализовать операции:  [] – доступа по индексу;  int() – определение размера списка;  \* список – умножение элементов списков a[i]\*b[i];  +n - переход вправо к элементу с номером n (с помощью класса-итератора). | 1, 2 |
| 20 | Класс-контейнер СПИСОК с ключевыми значениями типа int.  Реализовать операции:  [] – доступа по индексу;  \* список – перемножение элементов списков a[i]+b[i];  - число – вычитает константу из всех элементам списка;  -n - переход влево к элементу с номером n (с помощью класса-итератора). | 2, 3 |

Реализовать класс Вектор. Размер вектора ограничен значением MAX\_SIZE=30. Перегрузить для него операции:

* доступ по индексу([int i]),
* добавление элемента (+ int),
* удаление элемента из начала вектора (--).

Предусмотреть генерацию исключительных ситуаций.

Исключительные ситуации генерируются:

1 – в конструкторе с параметром при попытке создать вектор больше максимального размера; 2, 3 – в операции [] – при попытке обратиться к элементу с номером меньше 0 или больше текущего размера вектора; 4 – в операции + – при попытке добавить элемент с номером больше максимального размера; 5 – в операции – при попытке удалить элемент из пустого вектора.

**Варианты реализации заданий**:

1. Информация об исключительных ситуациях передается с помощью стандартного типа данных.

* Создать пустой проект.
* Добавить в него класс Vector.
* В файл Vector.h добавить описание класса Vector.
* В файл Vector.cpp добавить определение методов класса Vector.
* Добавить в проект файл lab9\_main.cpp. В файл записать функцию main(), создающую объекты класса Vector и позволяющую генерировать исключительные ситуации.
* Выполнить тестирование программы с генерацией различных исключительных ситуаций

2. Информация об исключительных ситуациях передается с помощью пользовательского класса.

* Создать пустой проект.
* Добавить в него файл error.h.
* В файл error.h добавить описание класса error.
* Добавить класс Vector. В файл Vector.h добавить описание класса Vector.
* В файл Vector.cpp добавить определение методов класса Vector:
* Добавить в проект файл lab9\_main.cpp. В файл записать функцию main(), создающую объекты класса Vector и позволяющую генерировать исключительные ситуации.
* Выполнить тестирование программы с генерацией различных исключительных ситуаций.

3. Информация об исключительных ситуациях передается с помощью иерархии пользовательских классов.

* Создать пустой проект.
* Добавить в него файл error.h.
* В файл error.h добавить описание иерархии пользовательских классов для определения исключительных ситуаций.
* Добавить класс Vector. В файл Vector.h добавить описание класса Vector.
* В файл Vector.cpp добавить определение методов класса Vector.
* Добавить в проект файл lab9\_main.cpp. В файл записать функцию main(), создающую объекты класса Vector и позволяющую генерировать исключительные ситуации.
* Выполнить тестирование программы с генерацией различных исключительных ситуаций.

### Содержание отчета

1. Постановка задачи (общая и для конкретного варианта)

2. Анализ задачи

* Определения функций для реализации поставленных задач
* Определение функции main()

3. Блок-схемы основных функций

4. Текст программы

5. Тесты

## Лабораторная работа № 7

## Контейнерные классы



### Цель работы:

1. Создание консольного приложения, состоящего из нескольких файлов в системе программирования Visual Studio.

2. Реализация класса-контейнера.

### Краткие теоретические сведения

**Абстрактные типы данных. Контейнеры**

АТД - тип данных, определяемый только через операции, которые могут выполняться над соответствующими объектами безотносительно к способу представления этих объектов.

АТД включает в себя абстракцию как через параметризацию, так и через спецификацию.

Абстракция через параметризацию может быть осуществлена так же, как и для процедур (функций); использованием параметров там, где это имеет смысл.

Абстракция через спецификацию достигается за счет того, что операции представляются как часть типа.

Для реализации АТД необходимо, во-первых, выбрать представление памяти для объектов и, во-вторых, реализовать операции в терминах выбранного представления.

Примером абстрактного типа данных является класс в языке С++.

реальных задачах требуется обрабатывать группы данных большого объема, поэтому в любом языке программирования существуют средства, позволяющие объединять данные в группы. В первую очередь это массивы. В С++ массивы – это очень простые конструкции. В ООП этого недостаточно для работы с группами однородных данных. Поэтому была выработана более общая концепция объединения однородных данных в группу – контейнер.

Контейнер – набор однотипных элементов. Встроенные массивы в С++ - частный случай контейнера.

Контейнер – это объект. Имя контейнера – это имя переменной. Контейнер, так же как и другие объекты, обладает временем жизни. Время жизни контейнера в общем случае не зависит от времени жизни его элементов. Элементами контейнера могут любые объекты, в том числе, и другие контейнеры.

Контейнеры могут быть фиксированного и переменного размера. В контейнере фиксированного размера число элементов постоянное, оно обычно задается при создании контейнера. Примером такого контейнера является массив. Для контейнера переменного размера количество элементов при объявлении обычно не задается. Элементы в таком контейнере добавляются и удаляются во время работы программы. Примером является список.

Если элементы контейнера не упорядочены, то добавление и удаление элементов обычно выполняется в начале и в конце контейнера. Способ вставки и удаления определяет вид контейнера. Если вставка и удаление осуществляется на одном конце, то такой контейнер называется стеком (Last In First Out – последним пришел, первым ушел). Если элементы добавляются на одном конце контейнера, а удаляются на другом, то такой контейнер называется очередью (First In First Out – первым пришел, последним ушел). Можно выполнять вставку и удаление на обоих концах контейнера, тогда такой контейнер будет называться двусторонней очередью (deque – double ended queue).

Если контейнер каким-то образом упорядочен, то операция вставки работает в соответствии с порядком элементов в контейнере. Операция удаления может выполняться по-разному: в начале, в конце или удаление заданного элемента.

**Операции контейнера**

Среди всех операций контейнера можно выделить несколько типовых групп:

* Операции доступа к элементам, которые обеспечивают и операцию замены значений элементов;
* Операции добавления и удаления элементов или групп элементов;
* Операции поиска элементов и групп элементов;
* Операции объединения контейнеров;
* Специальные операции, которые зависят от вида контейнера.

*Доступ к элементам*. Доступ к элементам контейнера бывает: последовательный, прямой и ассоциативный.

Прямой доступ – это доступ по индексу. Например, a[10] – требуется найти элемент контейнера с номером 10. В С++ нумерацию элементов контейнера принято начинать с нуля.

Ассоциативный доступ также выполняется по индексу, но индексом будет являться не номер элемента, а его содержимое. Пусть имеется контейнер –словарь, в котором хранится информация, состоящая, как минимум из двух полей: слово и его перевод. Индексом может служить слово, например, a[“word”]. С этим словом будет связано слово-перевод. Поле, с содержимым которого ассоциируется элемент контейнера, называется ключом или полем доступа. Элемент, с которым ассоциируется ключ, называется значением. Контейнер, который представляет ассоциативный доступ, состоит из пар «ключ-значение». Ассоциативный контейнер каким-то образом должен быть упорядочен по ключу. Например, в словаре упорядочение выполняется по алфавиту.

При последовательном доступе осуществляется перемещение от элемента к элементу контейнера. Операции последовательного доступа включают следующие действия:

* Перейти к первому элементу;
* Перейти к последнему элементу;
* Перейти к следующему элементу;
* Перейти к предыдущему элементу;
* Перейти на n элементов вперед;
* Перейти на n элементов назад;
* Получить текущий элемент.

Если контейнером является массив, то мы можем осуществлять последовательный доступ к его элементам с помощью указателя. И все перечисленные операции можно реализовать, используя указатели. В более общем случае, объект, который перебирает элементы контейнера, называется итератором. Итератор – это объект, который обеспечивает последовательный доступ к элементам контейнера. Итератор может быть реализован как часть класса-контейнера в виде набора методов: v.first() перейти к первому элементу v.last() перейти к последнему элементу v.next() перейти к следующему элементу v.prev() перейти к предыдущему элементу v.skip(n) перейти на n элементов вперед v.skip(-n) перейти на n элементов назад v.current() получить текущий элемент.

Итератор можно реализовать как класс, представляющий такой же набор операций.

С++ итератор реализуется как класс, который имеет такой же интерфейс, как и указатель для совместимости с массивами.

Если объект-итератор имеет имя iterv, то операции могут быть представлены следующим образом:

iterv=v.first() перейти к первому элементу

iterv=v.last() перейти к последнему элементу

iterv++ перейти к следующему элементу

iterv-- перейти к предыдущему элементу

iterv+=n перейти на n элементов вперед

v.skip-=n перейти на n элементов назад

\*iterv получить текущий элемент

При наличии последовательного доступа с помощью итератора обычными операциями являются операции.

Вставки элемента перед текущим элементом (после текущего);

Изменение значения текущего элемента;

Удаление текущего элемента;

Поиск элемента с заданным значением (возвращает итератор на текущий элемент).

*Операции объединения контейнеров*. Наиболее часто используется операция объединения двух контейнеров с получением нового контейнера. Она может быть реализована в разных вариантах:

Простое сцепление двух контейнеров: в новый контейнер попадают сначала элементы первого контейнера, потом второго, операция не коммутативна.

Объединение упорядоченных контейнеров, новый контейнер тоже будет упорядочен, операция коммутативна.

Объединение контейнеров как объединение множеств, в новый контейнер попадают только те элементы, которые есть хотя бы в одном контейнере, операция коммутативна.

Объединение контейнеров как пересечение множеств, в новый контейнер попадают только те элементы, которые есть в обоих контейнерах, операция коммутативна.

Для контейнеров-множеств может быть еще реализована операция вычитания, в контейнер попадают только те элементы первого контейнера, которых нет во втором, операция не коммутативна.

Извлечение части элементов из контейнера и создание нового контейнера. Эта операция может быть выполнена с помощью конструктора, а часть контейнера задается двумя итераторами.

### Постановка задачи

1. Определить класс-контейнер.

2. Реализовать конструкторы, деструктор, операции ввода-вывода, операцию присваивания.

3. Перегрузить операции, указанные в варианте.

4. Реализовать класс-итератор. Реализовать с его помощью операции последовательного доступа.

5. Написать тестирующую программу, иллюстрирующую выполнение операций.

### Варианты заданий

|  |  |
| --- | --- |
| №  варианта | Задание |
| 1 | Класс-контейнер ВЕКТОР с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  () – определение размера вектора;  + число – добавляет константу ко всем элементам вектора;  ++ - переход к следующему элементу (с помощью класса-итератора). |
| 2 | Класс-контейнер ВЕКТОР с элементами типа int.  Реализовать операции:  []– доступа по индексу;  int() – определение размера вектора;  + вектор – сложение элементов векторов a[i]+b[i];  +n - переход вправо к элементу c номером n (с помощью класса-итератора). |
| 3 | Класс-контейнер ВЕКТОР с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  + вектор – сложение элементов векторов a[i]+b[i];  + число – добавляет константу ко всем элементам вектора;  -- - переход к предыдущему элементу (с помощью класса-итератора). |
| 4 | Класс-контейнер ВЕКТОР с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  () – определение размера вектора;  \* число – умножает все элементы вектора на число;  -n – переход влево к элементу с номером n (с помощью класса-итератора). |
| 5 | Класс-контейнер ВЕКТОР с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  int() – определение размера вектора;  \* вектор – умножение элементов векторов a[i]\*b[i];  +n – переход вправо к элементу с номером n (с помощью класса-итератора). |
| 6 | Класс-контейнер ВЕКТОР с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  int() – определение размера вектора;  / вектор – деление элементов векторов a[i]/b[i];  -n – переход влево к элементу с номером n (с помощью класса-итератора). |
| 7 | Класс-контейнер ВЕКТОР с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  - вектор – вычитание элементов векторов a[i]-b[i];  \* число – умножает на константу все элементам вектора;  ++ - переход к следующему элементу (с помощью класса-итератора). |
| 8 | Класс-контейнер МНОЖЕСТВО с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  () – определение размера множества;  + – объединение множеств;  ++ - переход к следующему элементу (с помощью класса-итератора). |
| 9 | Класс-контейнер МНОЖЕСТВО с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  int() – определение размера множества;  \* – пересечение множеств;  -- - переход к предыдущему элементу (с помощью класса-итератора). |
| 10 | Класс-контейнер МНОЖЕСТВО с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  == - проверка на равенство;  > число – принадлежность числа множеству;  -n - переход влево к элементу с номером n (с помощью класса-итератора). |
| 11 | Класс-контейнер МНОЖЕСТВО с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  != - проверка на неравенство;  < число – принадлежность числа множеству;  +n – переход вправо к элементу с номером n (с помощью класса-итератора). |
| 12 | Класс-контейнер МНОЖЕСТВО с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  () – определение размера множества;  - – разность множеств;  -- – переход к предыдущему элементу ( с помощью класса-итератора). |
| 13 | Класс-контейнер МНОЖЕСТВО с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  == - проверка на равенство;  < число – принадлежность числа множеству;  -n – переход влево к элементу с номером n (с помощью класса-итератора). |
| 14 | Класс-контейнер МНОЖЕСТВО с элементами типа int.  Реализовать операции:  [] – доступа по индексу;  int() – определение размера множества;  + – объединение множеств;  -- - переход к предыдущему элементу (с помощью класса-итератора). |
| 15 | Класс-контейнер СПИСОК с ключевыми значениями типа int.  Реализовать операции:  []– доступа по индексу;  int() – определение размера списка;  + список – сложение элементов списков a[i]+b[i];  -n - переход влево к элементу c номером n (с помощью класса-итератора). |
| 16 | Класс-контейнер СПИСОК с ключевыми значениями типа int.  Реализовать операции:  [] – доступа по индексу;  () – определение размера вектора;  + число – добавляет константу ко всем элементам вектора;  ++ - переход к следующему элементу (с помощью класса-итератора). |
| 17 | Класс-контейнер СПИСОК с ключевыми значениями типа int.  Реализовать операции:  [] – доступа по индексу;  + список – сложение элементов списков a[i]+b[i];  + число – добавляет константу ко всем элементам списка;  -- - переход к предыдущему элементу (с помощью класса-итератора). |
| 18 | Класс-контейнер СПИСОК с ключевыми значениями типа int.  Реализовать операции:  [] – доступа по индексу;  () – определение размера списка;  \* число – умножает все элементы списка на число;  -n – переход влево к элементу с номером n (с помощью класса-итератора). |
| 19 | Класс-контейнер СПИСОК с ключевыми значениями типа int.  Реализовать операции:  [] – доступа по индексу;  int() – определение размера списка;  \* список – умножение элементов списков a[i]\*b[i];  +n - переход вправо к элементу с номером n (с помощью класса-итератора). |
| 20 | Класс-контейнер СПИСОК с ключевыми значениями типа int.  Реализовать операции:  [] – доступа по индексу;  \* список – перемножение элементов списков a[i]+b[i];  - число – вычитает константу из всех элементам списка;  -n - переход влево к элементу с номером n (с помощью класса-итератора). |

### Содержание отчета

1. Постановка задачи (общая и конкретного варианта)

2. Анализ задачи

* Описание класса-контейнера
* Определение компонентных функций
* Описание класса-итератора и его компонентных функций
* Функция main()
* Объяснение результатов работы программы

3. Блок-схемы создания контейнера и реализации операций

4. Текст программы

5. Тесты

## Лабораторная работа № 8

## Базы данных



### Цель работы:

Написать приложение для работы с простой базой данных, хранящей информацию об объекте на внешнем носителе. Приложение должно выполнять следующие функции:

1. Создание базы данных, содержащей записи указанного формата.

2. Просмотр базы данных.

3. Удаление элементов из базы данных (по ключу/ по номеру).

4. Корректировка элементов в базе данных (по ключу / по номеру).

5. Добавление элементов в базу данных (в начало / в конец/ с заданным номером).

6. Выполнение задания, указанного в варианте.

### Постановка задачи

Написать приложение для работы с простой базой данных, хранящей информацию об объекте на внешнем носителе. Приложение должно выполнять следующие функции:

1. Создание базы данных, содержащей записи указанного формата.

2. Просмотр базы данных.

3. Удаление элементов из базы данных (по ключу/ по номеру).

4. Корректировка элементов в базе данных (по ключу / по номеру).

5. Добавление элементов в базу данных (в начало / в конец/ с заданным номером).

6. Выполнение задания, указанного в варианте.

### Варианты заданий

|  |  |
| --- | --- |
| № варианта | Задание |
| 1 | В типизированном файле хранится информация о средней температуре за месяц. Программа должна:  • Добавлять, удалять, корректировать, позволять просматривать записи файла.  • Определять: дни, в которые температура поднималась выше средней за месяц;  • Самый длинный отрезок между днями с отрицательной температурой. |
| 2 | Типизированный файл содержит данные о ежемесячных доходах подразделений фирмы «Феникс» за пять лет. Программа должна  • Добавлять, удалять, корректировать, позволять просматривать записи файла.  • Определять: для каждого подразделения самые прибыльные годы;  • Наиболее длинный период каждого подразделения с доходом ниже среднего по всей фирме. |
| 3 | Списки студентов нескольких групп 1 курса хранятся в отдельных файлах – один файл на одну группу. Программа должна  • Добавлять, удалять, корректировать, позволять просматривать записи файлов.  • Создавать новый файл, в котором будет храниться общий список студентов 1 курса, отсортированный по алфавиту.  • Создавать новый файл, в котором будет храниться общий список студентов 1 курса, отсортированный по убыванию номеров групп. |
| 4 | Типизированный файл, имеет следующую структуру: Автор, Название, Год издания, Издательство. Программа должна  • Добавлять, удалять, корректировать, позволять просматривать записи файла.  • Выдавать по запросу пользователя: список литературы, указанного пользователем автора;  • Список литературы, изданной в указанный пользователем период. |
| 5 | Типизированный файл содержит данные о клиентах банка, получивших кредит (ФИО клиента, сумма кредита, вид кредита, срок, на который выдан кредит). Клиент может взять несколько кредитов. Программа должна  • Добавлять, удалять, корректировать, позволять просматривать записи файлов.  • Выполнять: вывод списка клиентов, получивших указанный пользователем вид кредита;  • Определение клиентов, взявших самый большой суммарный кредит. |
| 6 | В типизированном файле хранится информация о средней температуре за месяц. Программа должна  • Добавлять, удалять, корректировать, позволять просматривать записи файла.  • Определять: среднюю температуру каждой недели;  • Самые тёплые и самые холодные дни; |
| 7 | Типизированный файл содержит данные о ежемесячных доходах подразделений фирмы «Феникс» за пять лет. Программа должна  • Добавлять, удалять, корректировать, позволять просматривать записи файла.  • Определять: для каждого подразделения самые прибыльные годы;  • Наиболее длинный период каждого подразделения с доходом ниже среднего по всей фирме. |
| 8 | Списки студентов нескольких групп 1 курса хранятся в отдельных файлах – один файл на одну группу. Программа должна  • Добавлять, удалять, корректировать, позволять просматривать записи файлов.  • Создавать новый файл, в котором будет храниться общий список студентов 1 курса, отсортированный по убыванию номеров групп. |
| 9 | Типизированный файл, имеет следующую структуру: Автор, Название, Год издания, Издательство. Программа должна  • Добавлять, удалять, корректировать, позволять просматривать записи файла.  • Выдавать по запросу пользователя: список литературы, указанного пользователем автора;  • Список литературы, изданной в указанный пользователем период. |
| 10 | Типизированный файл содержит данные о клиентах банка, получивших кредит (ФИО клиента, сумма кредита, вид кредита, срок, на который выдан кредит). Клиент может взять несколько кредитов. Программа должна  • Добавлять, удалять, корректировать, позволять просматривать записи файлов.  • Выполнять: вывод списка клиентов, получивших указанный пользователем вид кредита;  • Определение клиентов, взявших самый большой суммарный кредит. |
| 11 | В типизированном файле хранится информация о средней температуре за месяц. Программа должна  • Добавлять, удалять, корректировать, позволять просматривать записи файла.  • Определять: среднюю температуру каждой недели;  • Самый длинный отрезок между днями с отрицательной температурой. |
| 12 | Типизированный файл содержит данные о ежемесячных доходах подразделений фирмы «Феникс» за пять лет. Программа должна  • Добавлять, удалять, корректировать, позволять просматривать записи файла.  • Определять: средний доход каждого подразделения за пять лет;  • Наиболее длинный период каждого подразделения с доходом ниже среднего по всей фирме. |
| 13 | Списки студентов нескольких групп 1 курса хранятся в отдельных файлах – один файл на одну группу. Программа должна  • Добавлять, удалять, корректировать, позволять просматривать записи файлов.  • Создавать новый файл, в котором будет храниться общий список студентов 1 курса, отсортированный по алфавиту. |
| 14 | Типизированный файл, имеет следующую структуру: Автор, Название, Год издания, Издательство. Программа должна  • Добавлять, удалять, корректировать, позволять просматривать записи файла.  • Выдавать по запросу пользователя: список имеющейся литературы;  • Список литературы, изданной в указанный пользователем период. |
| 15 | Типизированный файл содержит данные о клиентах банка, получивших кредит (ФИО клиента, сумма кредита, вид кредита, срок, на который выдан кредит). Клиент может взять несколько кредитов. Программа должна  • Добавлять, удалять, корректировать, позволять просматривать записи файлов.  • Выполнять: поиск клиентов, взявших кредит на N месяцев;  • Поиск клиентов по шаблону (3 первые буквы фамилии); |
| 16 | В типизированном файле хранится информация об автобусах, находящихся в автобусном парке (номер маршрута, ФИО водителя, номер автобуса, состояние автобуса (в парке или на маршруте)). Программа должна  • Добавлять, удалять, корректировать, позволять просматривать записи файла.  • Определять: количество автобусов на маршруте;  • Формировать список водителей автобусов, находящихся в парке. |
| 17 | Типизированный файл содержит данные о сотрудниках фирмы «Феникс» (ФИО, адрес, телефон, образование, название подразделения, в котором работает сотрудник). Программа должна  • Добавлять, удалять, корректировать, позволять просматривать записи файла.  • Определять: количество сотрудников с высшим образованием в каждом подразделении;  • Список сотрудников для заданного подразделения. |
| 18 | Типизированный файл содержит данные о студентах 1 курса (ФИО, группа, адрес, дата рождения, итоговые оценки по дисциплинам). Программа должна  • Добавлять, удалять, корректировать, позволять просматривать записи файлов.  • Выводить список студентов, имеющих хотя бы одну задолженность.  • Выводить список предметов, по которым есть хотя бы одна задолженность. |
| 19 | Типизированный файл хранит информацию о товарах в магазине (наименование, цена, дата поступления, срок годности товара, название отдела, в котором хранится товар). Программа должна  • Добавлять, удалять, корректировать, позволять просматривать записи файла.  • Выдавать по запросу пользователя: список просроченных продуктов в заданном отделе.  • Суммарную стоимость продуктов указанного наименования. |
| 20 | Типизированный файл содержит данные о клиентах банка, получивших кредит (ФИО клиента, сумма кредита, вид кредита, срок, на который выдан кредит). Клиент может взять несколько кредитов. Программа должна  • Добавлять, удалять, корректировать, позволять просматривать записи файлов.  • Выполнять: вывод списка клиентов, получивших указанный пользователем вид кредита;  • Вывод фамилии клиента, взявшего самый большой суммарный кредит. |
| 21 | В типизированном файле хранится информация об автобусах, находящихся в автобусном парке (номер маршрута, ФИО водителя, номер автобуса, состояние автобуса (в парке или на маршруте)). Программа должна  • Добавлять, удалять, корректировать, позволять просматривать записи файла.  • Выводить: информацию о водителе по шаблону (2 первые буквы фамилии);  • Список водителей автобусов, находящихся на маршруте. |
| 22 | Типизированный файл содержит данные о сотрудниках фирмы «Феникс» (ФИО, адрес, телефон, образование, название подразделения, в котором работает сотрудник). Программа должна  • Добавлять, удалять, корректировать, позволять просматривать записи файла.  • Выводить: информацию о заданном сотруднике по шаблону (первые 3 буквы фамилии);  • Фамилии сотрудников с высшим образованием для заданного подразделения. |
| 23 | Типизированный файл содержит данные о студентах 1 курса (ФИО, группа, адрес, дата рождения, итоговые оценки по дисциплинам). Программа должна  • Добавлять, удалять, корректировать, позволять просматривать записи файлов.  • Выводить: список студентов, не имеющих задолженностей.  • Средний балл за сессию заданного студента. |
| 24 | Типизированный файл хранит информацию о товарах в магазине (наименование, цена, дата поступления, срок годности товара, название отдела, в котором хранится товар). Программа должна  • Добавлять, удалять, корректировать, позволять просматривать записи файла.  • Выдавать по запросу пользователя: информацию о продуктах, поступивших в определенную дату.  • Список продуктов в заданном отделе. |
| 25 | Типизированный файл содержит данные о машинах, находящихся в гараже (ФИО владельца, сумма платежа в месяц, внесена ли плата за текущий месяц, марка машины, модель машины, год выпуска, номер машины, оплаченная сумма). У владельца может быть несколько машин. Программа должна  • Добавлять, удалять, корректировать, позволять просматривать записи файлов.  • Выполнять: формирование списка к оплате за текущий месяц;  • Поиск машины. |

### Методические указания

1. Для выбора действий использовать меню.

2. Для добавления элементов в файл использовать вспомогательную структуру для хранения элементов в оперативной памяти (список). Добавление должно осуществляться в начало, в конец файла и на позицию с заданным номером.

3. Удаление должно осуществляться по ключевому полю и по номеру. Удаляемые из файла записи помечаются как удаленные, когда их количество превысит половину файла, их можно удалять. При удалении использовать вспомогательную структуру для хранения элементов в оперативной памяти (список).

4. Предусмотреть возможность отмены последней операции удаления.

5. Корректировка выполняется по ключу и по номеру.

6. Предусмотреть команду «Сохранить изменения», по которой измененные данные из списка переписываются в файл.

### Содержание отчета

1. Постановка задачи (общая и для конкретного варианта)

2. Анализ задачи

* Определения функций для реализации поставленных задач
* Определение функции main()

3. Блок-схемы основных функций

4. Текст программы

5. Тесты

## Лабораторная работа № 9

## Поиск данных с помощью хеш-таблиц



### Цель работы:

1. Изучить построение функции хеширования и алгоритмов хеширования данных

2. Научиться разрабатывать алгоритмы открытого и закрытого хеширования при решении задач на языке C++.

### Краткие теоретические сведения

Поиск данных в большой базе может требовать достаточно большие временные затраты. Действительно, необходимо просмотреть значительное число элементов и каждый из них сравнить с ключом поиска. Для решения данной проблемы в настоящее время достаточно популярно применяется хеширование, значительно упрощающее поиск элементов.

Хеширование – процесс превращения некоторых данных на входе в некоторую битовую последовательность фиксированной длины на выходе. Эта строка позволяет отличить ее от большинства других строк, так же полученных путем хеширования. Другое название таких преобразований – хеш-функции или функции свертки, а результат работы – хеш или хеш-код.

Результат работы хеш-функции помогает различать одни данные от других, приводя в сравнение только значения функций хеширования, обработавших соответствующие данные, что очень удобно, так как конечные данные весят гораздо меньше информации, нежели данные, которым они соответствуют.

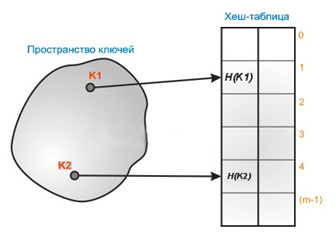


Рисунок 9.1 – Пример работы ключей

Хеширование используется для создания ассоциативных массивов, в которых индексами являются не числа, а произвольные значения. Оно применяется в поиске дубликатов в текстах, то есть поиска фрагментов, у которых имеется одинаковое значения хеш-функций. Кроме того, хеширование часто используется для хранения паролей; для создания уникальных идентификаторов, например, если для файла требуется уникальное название, можно посчитать результат обработки хешированием этого файла и сделать его названием. Оно также очень важно для подсчета контрольной суммы текста. Допустим, пользователю необходимо передать по сети какой-либо текст. Вместе с текстом передается контрольная сумма, которая, когда будет принята, сверяется с исходной. И, если суммы не совпадут, это будет означать, что при передаче текста возникли некоторые ошибки.

Однако, нередко нескольким различным данным, соответственно, различной длины на входе могут соответствовать одинаковые данные на выходе. Ситуации, когда разные данные имеют одинаковое значение хеш-функций, называются коллизиями (рисунок 9.1). При этом алгоритм хеширования должен стремиться к тому, чтобы разные данные имели разные значения. В редких случаях удается избежать коллизий вообще.

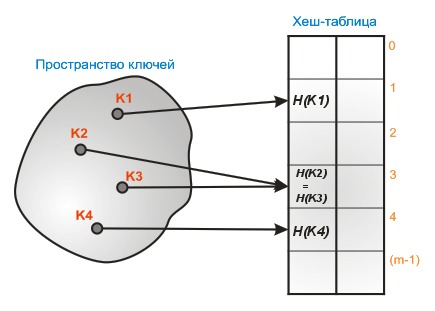


Рисунок 9.2 – Пример коллизии: ключи К2 и K3 в результате хеширования приобрели одинаковые значения хеш-функций

Хеш-таблица – структура данных, реализующая интерфейс ассоциативного массива. В ней хранятся пары, состоящие из ключа и хешированного ключа. Она позволяет добавлять новые пары, искать и удалять пары по ключу. Хеш-таблица формируется в определенном порядке хеш-функцией.

Хеш-таблицы часто применяются в базах данных, и, особенно, в языковых процессорах типа компиляторов и ассемблеров, где они повышают скорость обработки таблицы идентификаторов.

**Реализация хеш-таблицы и выбор хеш-функции**

Хеш-таблицы должны соответствовать следующим свойствам.

Прежде чем выполнять операции в хеш-таблице, вычисляется хеш-функция от ключа, результат которой является индексом в исходном массиве.

Коэффициент заполнения хеш-таблицы - количество хранимых элементов массива, деленное на число возможных значений хеш-функции. Является важным параметром, от которого зависит среднее время выполнения операций.

Принято считать, что хорошей является такая хеш-функция, которая удовлетворяет следующим условиям. Функция должна: быть простой с вычислительной точки зрения (это зависит от характеристик компьютера), распределять ключи в хеш-таблице наиболее равномерно (зависит от значений данных), стремиться уменьшить число коллизий. Функция не должна отображать какую-либо связь между значениями ключей в связь между значениями адресов.

**Составление функции хеширования в качестве примера**

Пусть имеется некоторый текст. Он состоит из большой последовательности символов, и нужно разработать функцию, которая выдаст практически уникальное значение хеш-функции для данного текста.

Для простоты возьмем набор символов, представленный на рисунке 9.3. Каждому символу ниже соответствует свой код по таблице ASCII.

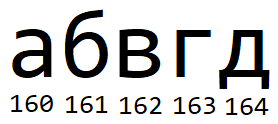


Рисунок 26.3 – Символы и соответствующие им коды из таблицы ASCII

По данной последовательности численных значений каждого символа необходимо составить значение хеш-функции. Составлением будет заниматься хеш-функция, для которой нужно придумать механизм обработки набора символов.

Необходимо учесть, что хешированный ключ имеет фиксированную длину и, желательно, небольшую. Пусть ключ после хеширования будет состоять из 8 разрядов, то есть 8 битов, принимающих значения от 0 до 1. Соответственно, количество различных значений хеш-функции может достигать 28=256 вариаций (от 0 до 255). На рисунке 9.4 представлен общий вид значения хеш-функции из восьми разрядов.

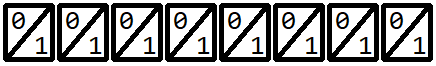


Рисунок 9.4 – Общий вид хеш-функции из восьми разрядов

Исходя из этого, не рационально использовать всю последовательность символов текста в качестве хешированного ключа. Ее необходимо каким-либо образом обработать, например:

c1, c2, c3, …, cn, где c – численное значение символа, а n – количество символов в тексте.

Можно просуммировать все численные значения:

S=c1+c2+c3+…+cn

В нашем случае сумма будет равна: (рисунок 9.3):

S=160+161+162+163+164=810

810 гораздо больше 255 – максимально возможного значения данной хеш-функции. Поэтому необходима дополнительная обработка.

Можно поделить сумму на 2N, где N – количество разрядов числа - результата работы хеш-функции, и найти остаток от деления:

H=S mod 2N

Тогда:

H=810 mod 256=42

При нахождении остатка от деления любого целого числа на 2 в степени количества разрядов конечных данных всегда будет получаться число меньшее, чем максимально возможное значение конечных данных. То есть в этой ситуации никогда не будет получено число, превышающее 255.

Но возникает проблема – возможно появление коллизии: если поменять местами хотя бы два слагаемых в сумме или прибавить какое-либо число к одному из слагаемых, а из другого вычесть его же, сумма не изменится, и, следовательно, конечные данные тоже не изменятся. Действительно:

S1=c1+c2+c3+…+cn; S2=c2+c1+c3+…+cn; S1=S2; H1=H2

S1=c1+c2+c3+…+cn; S2=c1+k+c2-k+c3+…+cn=c1+c2+c3+…+cn; S1=S2; H1=H2

Это означает, что тексты, отличающиеся друг от друга незначительно приобретут одно и то же значение хеш-функции. На рисунке 26.4 показан первоначальный текст и два текста, полученного из него путем небольших манипуляций. Во втором тексте поменялись местами два символа из первого текста, а в третьем – к численному значению символа г добавилась 3 (г стало ж), а из численного значения д было вычтено 3 (д стало б). Суммы численных значений не изменились, поэтому значение хеш-функции осталось прежним:

S1=160+161+162+163+164=810

S2=161+160+162+163+164=810

S3=160+161+162+166+161=810

S1=S2=S3

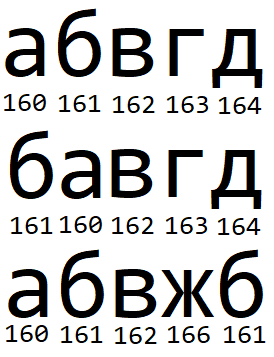


Рисунок 9.5 – Эквивалентные тексты по сумме численных значений символов

Недопустимо, чтобы текстам, отличающимся незначительно, соответствовали одни и те же значения хеш-функции. Поэтому необходимо модифицировать эту конструкцию, например, введением полиномиального хеша. Сумма будет находиться иным способом и будет представлять собой многочлен степени (n-1).

S\*=c1+c2p+c3p2+…+cnpn-1, где p≠0 и выбирается программистом по усмотрению в качестве константы.

Например:

S\*=160+161∙3+162∙32+163∙33+164∙34=19786

И, если пересчитать аналогично для двух похожих текстов, результат получится другим:

S\*2=161+160∙3+162∙32+163∙33+164∙34=19784

S\*3=160+161∙3+162∙32+166∙33+161∙34=19625

Далее, полученную сумму нужно так же поделить на 2N и найти остаток от деления:

H=S\* mod 2N

Таким образом:

H=19786 mod 256=74

Полученная хеш-функция не решает проблему коллизий полностью, но она избавляется от них лучше, чем версия, предложенная ранее (без использования полинома). Относительно предыдущего варианта она значительно сокращает вероятность возникновения коллизий, хотя допускает их.

**Реализация хеш-таблицы в программе**

Предположим, в хеш-таблице будут храниться некоторые данные о людях. Например, ФИО, дата рождения, № телефона.

Подготовим класс Hash.

Начнем с его приватных полей и функций. Пусть таблица имеет фиксированный размер, который измеряется максимальным количеством ячеек:

int max\_size = 20; //максимальный размер таблицы (необязательно)

Понадобится три списка, заполненных, соответственно, именами, фамилиями, отчествами – впоследствии они вместе с датой и номером телефона случайно будут формировать элементы будущей таблицы случайным образом.

vector<string> surnames = { "Иванов","Петров","Сидоров","Тесла","Маск","Эйнштейн","Ньютон","Гук","Кюри","Сталин","Ленин","Маркс"};

vector<string> names = { "Иван","Петр","Сидор","Никола","Илон","Альберт","Исаак","Роберт","Мария","Иосиф","Владимир","Карл","Алексей","Михаил","Дмитрий"};

vector<string> patronymics = { "Иванович","Петрович","Сидорович","Николаевич","Илонович","Альбертович","Исаакович","Робертович","Маркович","Иосифович","Владимирович","Карлович","Алексеевич","Михаилович","Дмитриевич"};

**Разрешение коллизий**

Коллизии необходимо разрешать, потому что они осложняют использование хеш-таблицы, нарушая однозначность соответствия между данными и их хешированными аналогами.

Для этого отводится место для хранения ключей, которые претендуют на ячейку хеш-таблицы, которая уже занята ключом, добавленным ранее. Этот механизм называется методом цепочек. Либо, если все ключи элементов известны заранее, соответственно, нет необходимости распределять элементы по ячейкам непосредственно в процессе добавления в таблицу, можно создать некоторую инъективную хеш-функцию, которая будет распределять ключи по ячейкам хеш-таблицы без коллизии. Такие хеш-таблицы, не нуждающиеся в механизме разрешения коллизий, называются хеш-таблицами с прямой или открытой адресацией.

Рассмотрим два предложенных способа подробнее.

**Метод цепочек**

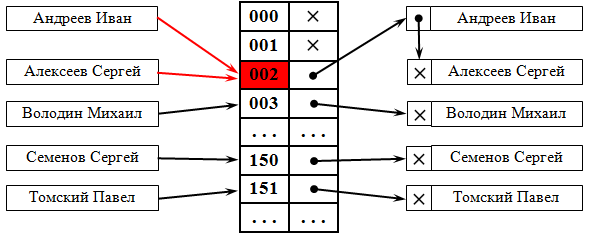


Рисунок 9.6 – Метод цепочек

Каждая ячейка массива является указателем на связный список (цепочку) пар ключ-значение, соответствующих одному и тому же хеш-значению ключа (рисунок 9.6). Коллизии приводят к тому, что появляются цепочки длиной более одного элемента.

Следовательно, если посчитать элементы для каждой из цепочек, состоящих более, чем из одного элемента, вычтя единицу из каждой такой суммы, а затем сложить все эти результаты, то получится общее число коллизий.

Чтобы внести в таблицу данные, нужно добавить элемент с предварительно найденным значением хеш-функции в конец или начало цепочки с соответствующим хеш-значением.

Чтобы найти и удалить какие-либо данные в таблице, достаточно найти цепочку элементов, хеш-значение которой совпадает с хеш-значением исходных данных. Затем, если цепочка состоит из одного элемента, можно удалить всю цепочку, в противном случае необходимо организовать поиск и в самой цепочке уже по ключу, а не по хешированным данным и удалить элемент.

**Метод открытой адресации**

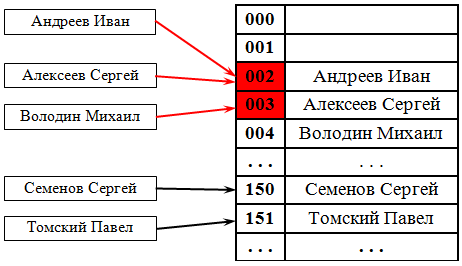


Рисунок 9.7 – Метод открытой адресации

При открытой адресации, если ячейка с вычисленным индексом занята, то просматриваются следующие записи таблицы по порядку до тех пор, пока не будет найден ключ K или пустая позиция в таблице. На рисунке 9.7 разрешение коллизий осуществляется методом открытой адресации. Два значения претендуют на ключ 002, для одного из них находится первое свободное (еще не занятое) место в таблице.

При любом методе разрешения коллизий необходимо ограничить длину поиска элемента. Если для поиска элемента необходимо более трех-четырех сравнений, то эффективность использования такой хеш-таблицы пропадает и ее следует реструктуризировать (найти другую хеш-функцию), чтобы минимизировать количество сравнений для поиска элемента.

Работа алгоритма поиска считается успешной, если все ячейки хеш-таблицы просмотрены ровно по одному разу.

Удаление элементов в такой схеме несколько затруднено. В таких случаях обычно заводится логический флажок для каждой из ячеек. Флажок обозначает, удален элемент либо нет. Необходимо изменить процедуру поиска существующего элемента так, чтобы удаленные ячейки считались занятыми. Процедура добавления должна учитывать удаленные элементы, принимая их в качестве свободных и сбрасывая значения флага при добавлении.

**Алгоритмы хеширования**

Существует несколько типов функций хеширования, каждая из которых имеет свои преимущества и недостатки и основана на представлении данных. Приведем обзор и анализ некоторых наиболее простых из применяемых на практике хеш-функций.

**Таблица прямого доступа**

Таблица прямого доступа является простейшей организацией таблицы, так как обеспечивает идеально быстрый поиск. В ней ключ – это уникальный адрес записи или то, что может быть преобразовано в адрес, так что никогда не возникнет ситуации, когда два разных ключа будут давать одинаковое значение хеш-функции в результате хеширования.

Перед созданием таблицы выделяется память для хранения всех ее элементов. Ячейки изначально заполнены пустыми записями. Затем записи будут внесены в таблицу в то место, которое определил исходный ключ.

При поиске ключ используется как адрес и по этому адресу выбирается запись. Если выбранная запись пустая, то записи с таким ключом вообще нет в таблице. Таблицы прямого доступа очень эффективны в использовании, но, к сожалению, область их применения весьма ограничена.

Но таблицы прямого доступа применимы только для тех задач, в которых множество всех теоретических возможных значений ключей записи совпадает по численности с множеством ячеек памяти, которые должны быть выделены для этих записей. Другими словами, сколько ключей, столько и записей.

На практике же чаще всего приходится сталкиваться с ситуацией, когда записей получается намного меньше, чем ключей. Например, если ключом будет фамилия, то, даже если ограничить ее длину десятью символами кириллицы, получится 3310 возможных значений ключей. Пусть ресурсы вычислительной системы и позволяют выделить место для хранения такого количества записей, но значительная часть этого пространства все равно окажется заполненной пустыми записями. Фактическое множество ключей не сможет прикрыть полностью пространство ключей.

В целях экономии памяти можно назначать размер пространства записей равным размеру фактического множества записей или превосходящим его незначительно. В этом случае необходимо иметь некоторую функцию, обеспечивающую отображение точки из пространства ключей в точку в пространстве записей, то есть, преобразование ключа в адрес записи: A=H(X), где A – адрес, X – ключ.

Идеальной хеш-функцией является инъективная функция, которая для любых двух неодинаковых ключей дает неодинаковые адреса.

**Метод остатков от деления**

Простейшей хеш-функцией является деление по модулю числового значения ключа Key на размер пространства записи HashTableSize. Результат интерпретируется как адрес записи. Следует иметь в виду, что такая функция хорошо соответствует первому, но плохо – последним трем требованиям к хеш-функции и сама по себе может быть применена лишь в очень ограниченном диапазоне реальных задач.

Если ключей меньше, чем элементов массива, то в качестве хеш-функции можно использовать деление по модулю, то есть остаток от деления целочисленного ключа Key на размерность массива HashTableSize, то есть: Key % HashTableSize.

На практике, метод деления – самый распространенный.

// функция создания хеш-таблицы метод деления по модулю

int Hash(int Key, int HashTableSize) {

return Key % HashTableSize;

}

**Метод функции середины квадрата**

Следующей хеш-функцией является функция середины квадрата. Значение ключа преобразуется в число, это число затем возводится в квадрат, из него выбираются несколько средних цифр и интерпретируются как адрес записи.

Пусть ключ будет четырехзначным числом. Его можно возвести в квадрат и взять из получившегося числа все разряды, начиная, например, с сотен и заканчивая сотнями тысяч. Эти разряды составят новое число, которое и будет являться результатом хеш-функци. Так, число 1234 при возведении в квадрат даст число 1522756, из которого в качестве значения хеш-функции можно извлечь число 5227.

**Метод свертки**

Еще одна из хеш-функций – это функция свертки. Ключ в своем цифровом представлении должен быть разбит на несколько частей. Длина каждой части равна длине требуемого адреса. Над частями могут быть произведены различные арифметические или логические операции, затрагивающие все число или только его разряды. В результате части при компоновке могут быть интерпретированы как адрес.

Пусть ключ – это строка, состоящая из фамилии, имени и отчества, разделенными пробелами. Функция хеширования разделит строку на части по пробелам. То есть имя будет находиться в одной части, фамилия – во второй, отчество – в третьей. В дальнейшем, с каждой из частей можно выполнить различные операции, применительно к численным кодам каждого из символов. Затем скомпоновать получившиеся результаты, выбрав определенный способ и получить адрес на выходе.

**Метод перевода в другую систему счисления**

В качестве хеш-функции также применяют функцию преобразования системы счисления. Ключ, записанный как число в некоторой системе счисления P, интерпретируется как число в системе счисления Q<P. Например, Q=P-1. Далее, можно выделить разряды из получившегося числа, взяв их в качестве нового числа, и провести с ними какие-либо манипуляции.

Простейший пример: 12345678910= 25937424510. Выделим четыре наименьших разряда, поделим 4510 на 10000, получив 0.4510, округлим вверх до 0.5 и умножим на 4510, отбросив мантиссу: 2255.

**Виды хеширования**

*Открытое хеширование.* При открытом хешировании множество значений разбивается на конечное число классов (сегментов). Для N классов строится определенная хеш-функция таким образом, что для любого элемента Х исходного множества хеш-функция обязательно принимает в качестве одного из аргументов номер класса (они могут быть пронумерованы от 0 до N-1). Таблица классов имеет N записей, содержащих заголовки для N цепочек (списков), в которые помещаются элементы соответствующего класса.

Если в каждом из классов примерно одинаковое количество элементов, то в этом случае списки должны быть наиболее короткими относительно общего числа классов. Если имеется M элементов, то средняя длина списка будет составлять M/N элементов.

*Закрытое хеширование.* При закрытом (внутреннем) хешировании в хеш-таблице хранятся непосредственно сами элементы, а не заголовки списков элементов, и пустые, пока не занятые ничем сегменты. Поэтому в каждой записи может храниться не более одного элемента. При закрытом хешировании применяется методика повторного хеширования.

При поиске элемента Х необходимо просмотреть все местоположения, пока не будет найден элемент Х или пока не встретится пустой сегмент.

Если осуществляется попытка добавления элемента в класс, в котором уже имеется элемент - возникает коллизия, то происходит повторное хеширование относительно нового количества элементов так, чтобы перераспределить номера элементов. Каждое местоположение последовательно проверяется, пока для добавляемого элемента не найдется свободное. Если свободных мест не окажется, это будет означать, что таблица заполнена, соответственно, элемент Х добавить нельзя.

Существует три метода повторного хеширования, помогающих определить местоположение добавляемых элементов, чьи хеш-значения совпали с одним из элементов в таблице.

1. Линейное опробование сводится к последовательному перебору сегментов таблицы с некоторым фиксированным шагом:

A=H(X)+Ci, где i – номер попытки разрешить коллизию, C – константа, определяющая шаг перебора.

Если шаг равен единице, происходит последовательный перебор всех сегментов после текущего.

1. Квадратичное опробование отличается формулой и зависит от попытки разрешить коллизию нелинейно.

A=H(X)+Ci +Di2, где i – номер попытки разрешить коллизию, C и D – константы.

Благодаря нелинейности квадратичного опробования уменьшается число проб при большом числе похожих значений хеш-функций. Но чем больше проб, тем больше вероятность того (растет в геометрической прогрессии), что получившееся значение адреса будет вне табличного пространства.

1. Двойное хеширование заключается в суммировании значений двух хеш-функций, работающих с одним и тем же элементом:

A=H1(X)+H2(X)

По мере заполнения хещ-таблицы могут возникнуть коллизии. Вследствие этого численное значение адреса может быть за пределами допустимых адресов. Эту проблему можно решить увеличением длины таблицы. Однако это хоть и сократит число коллизий и ускорит работу с хеш-таблицей, но приведет к нерациональному расходованию памяти.

### Постановка задачи

1. Создать динамический массив из записей (в соответствии с вариантом), содержащий не менее 100 элементов. Для заполнения элементов массива использовать ДСЧ.

2. Выполнить поиск элемента в массиве по ключу в соответствии с вариантом. Для поиска использовать хеш-таблицу.

3. Подсчитать количество коллизий при размере хеш-таблицы 40, 75 и 90 элементов.

### Варианты заданий

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № варианта | Данные | Ключ (string) | Хэш-функция | Метод рехеширования |
| 1 | ФИО, группа, рейтинг | ФИО | H(k)=k mod M | Метод цепочек |
| 2 | ФИО, №счета, сумма | №счета | H(k)= [M (kAmod1)], 0<A<1, mod1 – получение дробной части, [] – получение целой части | Метод открытой адресации |
| 3 | ФИО, №счета, сумма | ФИО | H(k)=k mod M | Метод открытой адресации |
| 4 | ФИО, №паспорта, адрес | №паспорта | H(k)= [M (kAmod1)], 0<A<1, mod1 – получение дробной части, [] – получение целой части | Метод цепочек |
| 5 | ФИО, №паспорта, адрес | ФИО | H(k)=k mod M | Метод цепочек |
| 6 | ФИО, №паспорта, адрес | Адрес | H(k)= [M (kAmod1)], 0<A<1, mod1 – получение дробной части, [] – получение целой части | Метод открытой адресации |
| 7 | ФИО, №телефона, адрес | №телефона | H(k)=k mod M | Метод открытой адресации |
| 8 | ФИО, №телефона, адрес | ФИО | H(k)= [M (kAmod1)], 0<A<1, mod1 – получение дробной части, [] – получение целой части | Метод цепочек |
| 9 | ФИО, №телефона, адрес | Адрес | H(k)=k mod M | Метод цепочек |
| 10 | ФИО, №паспорта, №телефона | №телефона | H(k)= [M (kAmod1)], 0<A<1, mod1 – получение дробной части, [] – получение целой части | Метод открытой адресации |
| 11 | ФИО, №паспорта, №телефона | №паспорта | H(k)=k mod M | Метод открытой адресации |
| 12 | ФИО, №паспорта, №телефона | ФИО | H(k)= [M (kAmod1)], 0<A<1, mod1 – получение дробной части, [] – получение целой части | Метод цепочек |
| 13 | ФИО, дата\_рождения, адрес | дата\_рождения | H(k)=k mod M | Метод цепочек |
| 14 | ФИО, дата\_рождения, адрес | адрес | H(k)= [M (kAmod1)], 0<A<1, mod1 – получение дробной части, [] – получение целой части | Метод открытой адресации |
| 15 | ФИО, дата\_рождения, адрес | ФИО | H(k)=k mod M | Метод открытой адресации |
| 16 | ФИО, дата\_рождения, №телефона | дата\_рождения | H(k)= [M (kAmod1)], 0<A<1, mod1 – получение дробной части, [] – получение целой части | Метод цепочек |
| 17 | ФИО, дата\_рождения, №телефона | ФИО | H(k)=k mod M | Метод цепочек |
| 18 | ФИО, дата\_рождения, №телефона | №телефона | H(k)= [M (kAmod1)], 0<A<1, mod1 – получение дробной части, [] – получение целой части | Метод открытой адресации |
| 19 | ФИО, дата\_рождения, №паспорта, | №паспорта, | H(k)=k mod M | Метод открытой адресации |
| 20 | ФИО, дата\_рождения, №паспорта, | дата\_рождения, | H(k)= [M (kAmod1)], 0<A<1, mod1 – получение дробной части, [] – получение целой части | Метод цепочек |
| 21 | ФИО,  дата\_рождения, №телефона, №паспорта | №телефона | H(k)=k mod M | Метод открытой адресации |
| 22 | ФИО,  дата\_рождения, №телефона, №паспорта | №паспорта | H(k)= [M (kAmod1)], 0<A<1, mod1 – получение дробной части, [] – получение целой части | Метод цепочек |
| 23 | ФИО,  дата\_рождения, №телефона, №паспорта | дата\_рождения | H(k)=k mod M | Метод открытой адресации |
| 24 | ФИО,  дата\_рождения, №телефона, №паспорта | №телефона | H(k)= [M (kAmod1)], 0<A<1, mod1 – получение дробной части, [] – получение целой части | Метод цепочек |
| 25 | ФИО,  дата\_рождения, №телефона, №паспорта | дата\_рождения | H(k)= [M (kAmod1)], 0<A<1, mod1 – получение дробной части, [] – получение целой части | Метод открытой адресации |

### Методические указания

1. Для выбора действий использовать меню.

2. Удаление должно осуществляться по ключевому полю и по номеру.

3. Корректировка выполняется по ключу и по номеру.

### Содержание отчета

1. Постановка задачи (общая и для конкретного варианта)

2. Анализ задачи

* Определения функций для реализации поставленных задач
* Определение функции main()

3. Блок-схемы основных функций

4. Текст программы

5. Тесты