Министерство образования Республики Беларусь
Учреждение образования «Витебский государственный университет имени П.М. Машерова»
Кафедра прикладного и системного программирования

Е.А. Корчевская, В.А. Степанов

АЛГОРИТМИЗАЦИЯ И ПРОГРАММИРОВАНИЕ

Методические рекомендации

Витебск
ВГУ имени П.М. Машерова
2022

УДК 004.42(075.8)
ББК 32.973.05я73+32.973.432.11я73
К70

Печатается по решению научно-методического совета учреждения образования «Витебский государственный университет имени П.М. Машерова». Протокол № 2 от 05.01.2022.

Авторы: доцент кафедры прикладного и системного программирования ВГУ имени П.М. Машерова, кандидат физико-математических наук, доцент **Е.А. Корчевская;** преподаватель-стажер кафедры прикладного и системного программирования ВГУ имени П.М. Машерова **В.А. Степанов**

Р е ц е н з е н т :
заведующий кафедрой «Информационные системы и автоматизация производства» УО «ВГТУ», кандидат технических наук, доцент *В.Е. Казаков*

Корчевская, Е.А.

К70 Алгоритмизация и программирование : методические рекомендации / Е.А. Корчевская, В.А. Степанов. – Витебск : ВГУ имени П.М. Машерова, 2022. – 48 с.

В методических рекомендациях изложены основные принципы объектно-ориентированного программирования, а также общие подходы при написании программ на языке C++. Даны краткие теоретические сведения, приведён список работ для самостоятельного выполнения, а также список контрольных вопросов.

Предназначается для студентов специальностей «Управление информационными ресурсами» (дисциплина «Алгоритмизация и программирование»), «Прикладная математика» (дисциплина «Основы и методологии программирования»), «Прикладная информатика» (дисциплина «Основы и методологии программирования»), «Программное обеспечение информационных технологий» (дисциплина «Основы алгоритмизации и программирования»), «Информационные системы и технологии» (дисциплина «Основы алгоритмизации и программирования»).

УДК 004.42(075.8)
ББК 32.973.05я73+32.973.432.11я73

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Методические рекомендации посвящены актуальному в настоящее время направлению программирования на языке C++. Материал разбит на 2 раздела.

В 1 разделе собраны сведения о принципах объектно-ориентированного программирования, подробно описан каждый из них. Приведены примеры программ на языке C++, которые демонстрируют основные принципы.

Во 2 разделе описаны классы стандартной библиотеки языка C++: string, vector, list, map, multimap, algorithm. Приведены основные методы стандартной библиотеки, примеры работы, задачи для самостоятельного выполнения и контрольные вопросы.

Материал соответствует отдельным темам рабочих программ курсов: «Алгоритмизация и программирование» (специальность «Управление информационными ресурсами»), «Основы и методологии программирования» (специальность «Прикладная математика»), «Основы и методологии программирования» (специальность «Прикладная информатика»), «Основы алгоритмизации и программирования» (специальность «Программное обеспечение информационных технологий»), «Основы алгоритмизации и программирования» (специальность «Информационные системы и технологии»).

1 ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

1.1 Принципы объектно-ориентированного программирования

Класс является типом данных, определяемым пользователем. В классе задаются свойства и поведение какого-либо предмета или процесса в виде полей данных и функций для работы с ними. Тип задает внутреннее представление данных в памяти компьютера, множество значений, которое могут принимать величины этого типа, а также операции и функции, применяемые к этим величинам. Все это можно задать и в классе. Существенным свойством класса является то, что детали его реализации скрыты от пользователей класса за интерфейсом

Конкретные величины типа данных «класс» называются экземплярами класса, или объектами. Основными свойствами ООП являются инкапсуляция, наследование, полиморфизм и абстракция.

Объединение данных с методами их обработки в сочетании со скрытием ненужной для использования этих данных информации называется инкапсуляцией. Инкапсуляция повышает степень абстракции программы: данные класса и реализация его функций находятся ниже уровня абстракции, и для написания программы информация о них не требуется. Кроме того, инкапсуляция позволяет изменить реализацию класса без модификации основной части программы, если интерфейс остался прежним. Простота модификации является очень важным критерием качества программы.

Абстракция – набор значимых характеристик объекта, исключая из рассмотрения незначимые.

Наследование – это возможность создания иерархии классов, когда потомки наследуют все свойства своих предков, могут их изменять и добавлять новые. Свойства при наследовании повторно не описываются, что сокращает объем программы.

Полиморфизм – возможность использовать в различных классах иерархии одно имя для обозначения сходных по смыслу действий и гибко выбирать требуемое действие во время выполнения программы.

1.2 Понятие класса и объекта

Классы предоставляют разработчику возможность моделировать объекты, которые имеют атрибуты и варианты поведения.

Описание класса выглядит так:

class <имя> {
[private:]
<описание скрытых элементов>
public:
<описание доступных элементов>
}

Метки public: (открытая) и private: (закрытая) называются спецификаторами доступа к элементам. Любые поля и методы, объявленные после спецификатора доступа к элементам public: (и до следующего спецификатора доступа к элементам), доступны при любом обращении программы к объекту класса. Любые поля и методы, объявленные после спецификатора доступа к элементам private: (и до следующего спецификатора доступа к элементам), доступны только внутри этого класса. Спецификаторы доступа к элементам всегда заканчиваются двоеточием (:) и могут появляться в определении класса много раз и в любом порядке.

```
class flower {
   private:
      double length;
      string name;
  public:
     flower(double length, string name) {
          this->length = length;
          this->name = name;
     }
     void set_length(double length) {
        this->length=length;
     }
    void set_name(string name) {
       this->name=name;
    }
    double get_length() {
       return length;
    }
    string get_name()  {
       return name;
   }
}
```

Поля класса могут иметь любой тип, кроме типа этого же класса (но могут быть указателями или ссылками на этот класс).

Классы могут быть глобальными (объявленными вне любого блока) и локальными (объявленными внутри блока, например, функции или другого класса).

Конкретные переменные типа «класс» называются экземплярами класса, или объектами.

```
flower rose(50, “Роза”);
flower *   chamomile = new flower (10, “Ромашка”);
```

При создании каждого объекта выделяется память, достаточная для хранения всех его полей и автоматически вызывается конструктор, выполняющий их инициализацию. Методы класса не тиражируются. При выходе объекта из области действия он уничтожается, при этом автоматически вызывается деструктор.

Для доступа к элементам объекта используются операция. (точка) при обращении к элементу через имя объекта и операция -> при обращении через указатель:

```
double n = rose.get_length();
cout << chamomile ->get_name();
```

Можно создать константный объект, значения полей которого изменять запрещается. К нему должны применяться только константные методы:

```
class flower {
   private:
      double length;
      string name;
  public:
     flower(double length, string name) {
          this->length = length;
          this->name = name;
     }
     void set_length(double length) {
         this->length=length;
     }
    void set_name(string name) {
       this->name=name;
    }
    double get_length() const{
       return length;
    }
    string get_name() const {
       return name;
   }
}
```

```
const monstr Rose(10, “Роза на акции”); // Константный объект
cout <<Rose.get_length();
```

Константный метод:

- объявляется с ключевым словом const после списка параметров;
- не может изменять значения полей класса;
- может вызывать только константные методы;
- может вызываться для любых (не только константных) объектов.

Указатель this

Каждый объект содержит свой экземпляр полей класса. Методы класса находятся в памяти в единственном экземпляре и используются всеми объектами совместно, поэтому необходимо обеспечить работу методов с полями именно того объекта, для которого они были вызваны. Это обеспечивается передачей в функцию скрытого параметра this, в котором хранится константный указатель на вызвавший функцию объект. Указатель this неявно используется внутри метода для ссылок на элементы объекта. В явном виде этот указатель применяется в основном для возвращения из метода указателя (return this;) или ссылки (return *this;) на вызвавший объект.

```
flower& greater( flower& C)  {
   if (get_length() > C.get_length())
      return *this;
   return C;
}
```

Добавим в приведенный выше класс flower новый метод, возвращающий наиболее длинный цветок из двух, один из которых вызывает метод, а другой передается ему в качестве параметра

Указатель this можно также применять для идентификации поля класса в том случае, когда его имя совпадает с именем формального параметра метода. В приведенном примере flower мы использовали указатель this в конструкторе:

```
    flower(double length, string name) {
          this->length = length;
          this->name = name;
     }
```

Конструктор предназначен для инициализации объекта и вызывается автоматически при его создании.

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
```

Приведем пример класса с несколькими конструкторами, добавив в класс flower поле, задающее цвет.

```
      enum color {red, pink, white};
class flower {
```

```
  private:
     double length;
     string name;
     color bloom;
  public:
     flower(double length, string name) {
          this->length = length;
          this->name = name;
          this->bloom = red;
     }
     flower(color bloom){
              switch (bloom){
                case red: length = 100; name= “красная роза”; bloom = red; break;
                case pink: length = 50; name = “розовый тюльпан”; bloom = pink;
break;
                case white: length = 30; name= “белая роза”; bloom = white; break;
    }

}
      void set_length(double length) {
         this->length=length;
      }
     void set_name(string name) {
        this->name=name;
     }
    double get_length() const{
       return length;
    }
    string get_name() const {
       return name;
   }
}
```

Статические элементы класса

С помощью модификатора static можно описать статические поля и методы класса. Их можно рассматривать как глобальные переменные или функции, доступные только в пределах области класса.

Статические поля применяются для хранения данных, общих для всех объектов класса. Эти поля существуют для всех объектов класса в единственном экземпляре, то есть не дублируются. Статические поля доступны как через имя класса, так и через имя объекта:

Статические методы предназначены для обращения к статическим полям класса. Они могут обращаться непосредственно только к статическим полям и вызывать только другие статические методы класса. Обращение к

статическим методам производится так же, как к статическим полям — либо через имя класса, либо, если хотя бы один объект класса уже создан, через имя объекта.

```
class A{
        static int count;
    public:
        static void inc_count(){
           count++;
        }
};
  A::int count:
  void f(){
        a.inc_count();
       // или A::inc__count():
}
```

Деструкторы

Деструктор – это особый вид метода, применяющийся для освобождения памяти, занимаемой объектом. Деструктор вызывается автоматически, когда объект выходит из области видимости:

• для локальных объектов – 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.

Деструктор:

• не имеет аргументов и возвращаемого значения;

• не может быть объявлен как const или static;

• не наследуется;

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.

Задания для самостоятельного выполнения

1. Реализовать класс *Matrix*, содержащий методы вывода квадратной матрицы размерности N с вещественными элементами в общепринятом виде, нахождения транспонированной матрицы и определителя матрицы.

2. Реализовать класс *Parallelogram*, определяемый меньшим углом и длиной сторон и содержащий методы нахождения периметра и площади параллелограмма, длин его диагоналей.

3. Реализовать класс *Circle*, определяемый длиной радиуса координатой центра. Класс должен содержать методы вычисления длины окружности и площади круга.

4. Реализовать класс *Fraction*, содержащий методы вывода суммы дробей и произведения и функцию выделения целой части.

5. Реализовать класс *Triangle*, содержащий длины сторон, функцию, определяющую правильность введения данных, т.е. возможность построения треугольника по заданным сторонам, и функцию, вычисляющую площадь треугольника.

6. Реализовать класс *Vector*, содержащий координаты (вектора на плоскости) его начала и конца и метод нахождения угла между ними.

7. Реализовать класс *Polynomial*, содержащий поля степень, аргумент и коэффициенты. Создать метод вычисления значения многочлена от аргумента и вывода многочлена в общем виде на экран.

8. Реализовать класс *Complex*, содержащий методы нахождения аргумента и модуля комплексного числа и вывода числа на экран в общепринятом виде.

9. Реализовать класс *Address*, включающий полный почтовый адрес: индекс, область, город, улица дом, квартира. Определить разные виды конструктора.

10. Реализовать класс *Ellipse*, определяемый координатами фокусов и эксцентриситетом. Класс должен содержать методы нахождения канонического уравнения эллипса и длин его полуосей.

Контрольные вопросы

1. Что такое класс? Чем отличается класс от объекта?
2. Для чего нужны ключевые слова public и private? Можно ли использовать ключевые слова public и private в структуре?
3. Обязательно ли делать поля класса приватными? Как инициализировать приватные поля класса?
4. В чем разница между указателем и ссылкой?
5. Что обозначается ключевым словом this? Для чего может использоваться конструкция *this?

1.3 Наследование

Механизм наследования классов позволяет строить иерархии, в которых производные классы получают элементы родительских, или базовых, классов и могут дополнять их или изменять их свойства. При большом количестве никак не связанных классов управлять ими становится невозможным. Наследование позволяет справиться с этой проблемой путем упорядо-

чивания и ранжирования классов, то есть объединения общих для нескольких классов свойств в одном классе и использования его в качестве базового. Классы, находящиеся ближе к началу иерархии, объединяют в себе наиболее общие черты для всех нижележащих классов. По мере продвижения вниз по иерархии классы приобретают все больше конкретных черт. Множественное наследование позволяет одному классу обладать свойствами двух и более родительских классов.

Ключи доступа. При описании класса в его заголовке перечисляются все классы, являющиеся для него базовыми. Возможность обращения к элементам этих классов регулируется с помощью ключей доступа private (частные), protected (защищенные) и public (публичные):

class имя : [private | protected | public] базовый_класс
{ тело класса };

Если базовых классов несколько, они перечисляются через запятую. Ключ доступа может стоять перед каждым классом, например:

```
class A { ... };
class B { ... };
class C { ... };
class D: A, protected B, public C { ... };
```

По умолчанию для классов используется ключ доступа private.

До сих пор мы рассматривали только применяемые к элементам класса спецификаторы доступа private и public. Для любого элемента класса может также использоваться спецификатор protected, который для одиночных классов, не входящих в иерархию, равносилен private. Разница между ними проявляется при наследовании, что можно видеть из приведенной таблицы:

Ключ доступа	Спецификатор в базовом классе	Доступ в производном классе
Private	private protected public	нет private private
Protected	private protected public	нет protected protected
Public	private protected public	нет protected public

Если базовый класс наследуется с ключом private, можно выборочно сделать некоторые его элементы доступными в производном классе, объявив их в секции public производного класса с помощью операции доступа к области видимости:

```
class Base{
   …
```

```
   public: void f();
}
class Derived : private Base{
   …
   public: Base::void f ();
};
```

Наследование называется простым, если производный класс имеет одного родителя.

Рассмотрим правила наследования различных методов.

•Конструкторы не наследуются, поэтому производный класс должен иметь собственные конструкторы. Порядок вызова конструкторов определяется приведенными ниже правилами.

• Если в конструкторе производного класса явный вызов конструктора базового класса отсутствует, автоматически вызывается конструктор базового класса по умолчанию (то есть тот, который можно вызвать без параметров).

• Для иерархии, состоящей из нескольких уровней, конструкторы базовых классов вызываются, начиная с самого верхнего уровня. После этого выполняются конструкторы тех элементов класса, которые являются объектами, в порядке их объявления в классе, а затем исполняется конструктор класса.

• В случае нескольких базовых классов их конструкторы вызываются в порядке объявления.

Ниже перечислены правила наследования деструкторов,

• Деструкторы не наследуются, и если программист не описал в производном

классе деструктор, он формируется по умолчанию и вызывает деструкторы всех базовых классов.

• В отличие от конструкторов, при написании деструктора производного класса в нем не требуется явно вызывать деструкторы базовых классов, поскольку это будет сделано автоматически.

• Для иерархии классов, состоящей из нескольких уровней, деструкторы вызываются в порядке, строго обратном вызову конструкторов: сначала вызывается деструктор класса, затем – деструкторы элементов класса, а потом деструктор базового класса.

```
#include <iostream>
using namespace std;
class square{
   protected:
      double width;
   public:
      square(double width){
```

```
        this->width = width;
      }
      double area(){
        return width*width;
      }
};
class rectangle: public square{
   protected:
     double height;
   public:
     rectangle(double width, double height):square(width){
        this-> height = height;
     }
     double area(){
        return width*height;
     }
};
class trapeze: public rectangle{
   double length;
   public:
     trapeze(double width, double height, double length):rectangle(width,
height){
        this->length = length;
     }
     double area(){
        return (width+length)*height/2;
     }
};

int main()
{   square Object1(5);
   cout<< Object1.area()<<endl;
   rectangle Object2(7, 8);
   cout<< Object2.area()<<endl;
   trapeze Object3(3,2.5, 7);
   cout<< Object3.area()<<endl;
   return 0;
}
```

Множественное наследование

Множественное наследование означает, что класс имеет несколько базовых классов. Если в базовых классах есть одноименные элементы, при этом может произойти конфликт идентификаторов, который устраняется с помощью операции доступа к области видимости.

Множественное наследование применяется для того, чтобы обеспечить производный класс свойствами двух или более базовых. Чаще всего один из этих классов является основным, а другие обеспечивают некоторые дополнительные свойства, поэтому они называются классами подмешивания. По возможности классы подмешивания должны быть виртуальными и создаваться с помощью конструкторов без параметров, что позволяет избежать многих проблем, возникающих при ромбовидном наследовании (когда у базовых классов есть общий предок).

Задания для самостоятельного выполнения

1. Реализовать класс *Square*, члены класса – длина стороны. Предусмотреть в классе методы вычисления и вывода сведений о фигуре – диагональ, периметр, площадь. Создать производный класс – правильная квадратная призма с высотой *H*, добавить в класс метод определения объема фигуры, расчета площади и вывода сведений о фигуре. Написать программу, демонстрирующую работу с этими классами: дано *S* квадратов и *P* призм, найти квадрат с максимальной площадью и призму с максимальной диагональю.

2. Реализовать класс *Triangle*, члены класса – длины трех сторон. Предусмотреть в классе методы проверки существования треугольника, вычисления и вывода сведений о фигуре – длины сторон, углы, периметр, площадь. Создать производный класс – равносторонний треугольник, добавить в класс проверку, является ли треугольник равносторонним, и метод вывода сведений о фигуре. Написать программу, демонстрирующую работу с классом: дано *T* треугольников и *E* равносторонних треугольников, найти среднюю площадь для *A* треугольников и наибольший равносторонний треугольник.

3. Реализовать класс *Circle*, член класса – радиус *R*. Предусмотреть в классе методы вычисления и вывода сведений о фигуре – площади, длины окружности. Создать производный класс – круглый прямой цилиндр с высотой *h*, добавить в класс метод определения объема фигуры, методы расчета площади и вывода сведений о фигуре. Написать программу, демонстрирующую работу с классом: дано *C* окружностей и *M* цилиндров, найти окружность максимальной площади и средний объем цилиндров.

4. Реализовать класс *Square*, члены класса – длина стороны. Предусмотреть в классе методы вычисления и вывода сведений о фигуре – диагональ, периметр, площадь. Создать производный класс – правильная пирамида с апофемой *a*, добавить в класс метод определения объема фигуры, методы расчета площади и вывода сведений о фигуре. Написать программу, демонстрирующую работу с классом: дано *S* квадратов и *P* пирамид, найти квадрат с минимальной площадью и количество пирамид с высотой более числа *N* (*N* вводит пользователь).

5. Реализовать класс *Quadrilateral* (четырехугольник), члены класса – координаты четырех точек. Предусмотреть в классе методы проверки существования четырехугольника вычисления и вывода сведений о фигуре – длины сторон, диагоналей, периметр, площадь. Создать производный класс – параллелограмм, предусмотреть в классе проверку, является ли фигура параллелограммом. Написать программу, демонстрирующую работу с классом: дано Q четырехугольников и P параллелограммов, найти среднюю площадь C четырехугольников и параллелограммы наименьшей и наибольшей площади.

6. Реализовать класс *Triangle*, члены класса – координаты трех точек. Предусмотреть в классе методы проверки существования треугольника, вычисления и вывода сведений о фигуре – длины сторон, углы, периметр, площадь. Создать производный класс – равносторонний треугольник, предусмотреть в классе проверку, является ли треугольник равносторонним. Написать программу, демонстрирующую работу с классом: дано T треугольников и E равносторонних треугольников, вывести номера одинаковых треугольников и равносторонний треугольник с наименьшей медианой.

7. Реализовать класс *Rectangle*, члены класса – длины сторон a и b. Предусмотреть в классе методы вычисления и вывода сведений о фигуре – длины сторон, диагоналей, периметр, площадь. Создать производный класс – параллелепипед с высотой h, добавить в класс метод определения объема фигуры, методы расчета площади и вывода сведений о фигуре. Написать программу, демонстрирующую работу с классом: дано N прямоугольников и M параллелепипедов, найти количество прямоугольников, у которых площадь больше средней площади прямоугольников и количество кубов (все ребра равны).

8. Реализовать класс *Circle*, член класса – радиус R. Предусмотреть в классе методы вычисления и вывода сведений о фигуре – площади, длины окружности. Создать производный класс – конус с высотой h, добавить в класс метод определения объема фигуры, методы расчета площади и вывода сведений о фигуре. Написать программу, демонстрирующую работу с классом: дано N окружностей и M конусов, найти количество окружностей, у которых площадь меньше средней площади всех окружностей, и наибольший по объему конус.

9. Реализовать класс *Quadrilateral* (четырехугольник), члены класса – координаты четырех точек. Предусмотреть в классе методы вычисления и вывода сведений о фигуре – длины сторон, диагоналей, периметр, площадь. Создать производный класс – равнобедренная трапеция, предусмотреть в классе проверку, является ли фигура равнобедренной трапецией. Написать программу, демонстрирующую работу с классом: дано N четырехугольников и M трапеций, найти максимальную площадь четырехугольников и количество четырехугольников, имеющих максимальную площадь, и трапецию с наименьшей диагональю.

10. Реализовать класс *Equilateral* (равносторонний треугольник), член класса – длина стороны. Предусмотреть в классе методы вычисления и вывода сведений о фигуре – периметр, площадь. Создать производный класс – правильная треугольная призма с высотой *H*, добавить в класс метод определения объема фигуры, методы расчета площади и вывода сведений о фигуре. Написать программу, демонстрирующую работу с классом: дано *N* треугольников и *M* призм. Найти количество треугольников, у которых площадь меньше средней площади треугольников, и призму с наибольшим объемом.

Контрольные вопросы

1. Для чего используется наследование?
2. Чем отличается модификатор доступа protected от модификаторов private и public?
3. Чем открытое наследование отличается от закрытого и защищенного?
4. Какие функции не наследуются?
5. Каков порядок вызова конструкторов? А деструкторов?

1.4 Виртуальные функции

Опишем класс геометрической фигуры квадрат, который содержит метод area(), и производные классы, содержащие метод area().

```
#include <iostream>
using namespace std;
class square{
   protected:
      double width;
   public:
      square(double width){
         this->width = width;
      }
      double area(){
         return width*width;
      }
};
class rectangle: public square{
   protected:
     double height;
   public:
     rectangle(double width, double height):square(width){
        this-> height = height;
     }
      double area(){
        return width*height;
     }
```

```
};
class trapeze: public rectangle{
   double length;
   public:
     trapeze(double width, double height, double length):rectangle(width,
height){
         this->length = length;
     }
     double area(){
        return (width+length)*height/2;
     }
};

int main()
{   square* Object1 = new rectangle(5,9);
   cout<< Object1->area()<<endl;
   Object1 = new trapeze(3, 2.5, 7);
   cout<< Object1->area()<<endl;

   return 0;
}
```

В результате работы программы будут выведены значения 25 и 9.

Работа с объектами чаще всего производится через указатели. Указателю на базовый класс можно присвоить значение адреса объекта любого производного класса. Вызов методов объекта происходит в соответствии с типом указателя, а не фактическим типом объекта, на который он ссылается, поэтому при выполнении оператора, например,

```
Object1->area();
```

будет вызван метод класса square, а не класса rectangle, поскольку ссылки на методы разрешаются во время компоновки программы. Этот процесс называется ранним связыванием. Чтобы вызвать метод класса daemon, можно использовать явное преобразование типа указателя:

```
(rectangle * Object1)->area();
```

Это не всегда возможно, поскольку в разное время указатель может ссылаться на объекты разных классов иерархии, и во время компиляции программы конкретный класс может быть неизвестен. В качестве примера можно привести функцию, параметром которой является указатель на объект базового класса. На его место во время выполнения программы может быть передан указатель на любой производный класс.

Виртуальная функция – это функция, объявленная с ключевым словом virtual в базовом классе и переопределенная в одном или нескольких производных классах. Виртуальные функции являются особыми функциями, потому что при вызове объекта производного класса с помощью указателя

или ссылки на него C++ определяет во время исполнения программы, какую функцию вызвать, основываясь на типе объекта. Для разных объектов вызываются разные версии одной и той же виртуальной функции. Класс, содержащий одну или более виртуальных функций, называется полиморфным классом.

Для определения виртуального метода используется спецификатор virtual, например:

virtual double area();

Рассмотрим правила описания и использования виртуальных методов.

• Если в базовом классе метод определен как виртуальный, метод, определенный в производном классе с тем же именем и набором параметров, автоматически становится виртуальным, а с отличающимся набором параметров – обычным.

• Виртуальные методы наследуются, то есть переопределять их в производном классе требуется только при необходимости задать отличающиеся действия. Права доступа при переопределении изменить нельзя.

• Если виртуальный метод переопределен в производном классе, объекты этого класса могут получить доступ к методу базового класса с помощью операции доступа к области видимости.

• Виртуальный метод не может объявляться с модификатором static, но может быть объявлен как дружественный.

• Если в классе вводится описание виртуального метода, он должен быть определен хотя бы как чисто виртуальный.

Чисто виртуальный метод содержит признак =0 вместо тела, например:

virtual double area()= 0;

Чисто виртуальный метод должен переопределяться в производном классе (возможно, опять как чисто виртуальный).

Если определить метод area() в классе square как виртуальный, решение о том, метод какого класса вызвать, будет приниматься в зависимости от типа объекта, на который ссылается указатель:

```
#include <iostream>
using namespace std;
class square{
   protected:
      double width;
   public:
      square(double width){
         this->width = width;
      }
     virtual double area(){
         return width*width;
      }
```

```
};
class rectangle: public square{
   protected:
     double height;
   public:
     rectangle(double width, double height):square(width){
        this-> height = height;
     }
      double area(){
        return width*height;
     }
};
class trapeze: public rectangle{
   double length;
   public:
     trapeze(double width, double height, double length):rectangle(width,
height){
        this->length = length;
     }
     double area(){
        return (width+length)*height/2;
     }
};

int main()
{   square* Object1 = new rectangle(5,9);
   cout<< Object1->area()<<endl;
   Object1 = new trapeze(3, 2.5, 7);
   cout<< Object1->area()<<endl;

   return 0;
}
```

В результате работы программа выдаст: 45 и 12.5.

Рекомендуется делать виртуальными деструкторы для того, чтобы гарантировать правильное освобождение памяти из-под динамического объекта, поскольку в этом случае в любой момент времени будет выбран деструктор, соответствующий фактическому типу объекта.

Виртуальный механизм работает только при использовании указателей или ссылок на объекты. Объект, определенный через указатель или ссылку и содержащий виртуальные методы, называется полиморфным. В данном случае полиморфизм состоит в том, что с помощью одного и того же обращения к методу выполняются различные действия в зависимости от типа, на который ссылается указатель в каждый момент времени.

Задания для самостоятельного выполнения

1. Разработать программу с использованием наследования классов, реализующую классы: графический объект; круг; квадрат. Используя виртуальные функции, не зная с объектом какого класса вы работаете, выведите на экран его размер и координаты.

2. Разработать программу с использованием наследования классов, реализующую классы: железнодорожный вагон; вагон для перевозки автомобилей; цистерна. Используя виртуальные функции, не зная с объектом какого класса вы работаете, выведите на экран его вес и количество единиц товара в вагоне.

3. Разработать программу с использованием наследования классов, реализующую классы: массив; стек; очередь. Используя виртуальные функции, не зная с объектом какого класса вы работаете, выведите на экран количество элементов и добавьте элемент.

4. Разработать программу с использованием наследования классов, реализующую классы: воин; пехотинец(винтовка); матрос(кортик). Используя виртуальные функции, не зная с объектом какого класса вы работаете, выведите на экран его возраст и вид оружия.

5. Разработать программу с использованием наследования классов, реализующую классы: точка; линия; круг. Используя виртуальные функции, не зная с объектом какого класса вы работаете, выведите на экран координаты и размер.

6. Разработать программу с использованием наследования классов, реализующую классы: работник больницы; медсестра; хирург. Используя виртуальные функции, не зная с объектом какого класса вы работаете, выведите на экран возраст и название должности.

7. Разработать программу с использованием наследования классов, реализующую классы: точка; квадрат пирамида. Используя виртуальные функции, не зная с объектом какого класса вы работаете, выведите на экран его размер и координаты.

8. Разработать программу с использованием наследования классов, реализующую классы: реагент; углерод; железо. Используя виртуальные функции, не зная с объектом какого класса вы работаете, выведите на экран его количество и свойства (форма кристаллической решетки для углерода и чистота выработки руды для железа).

9. Разработать программу с использованием наследования классов, реализующую классы: работник фирмы; стажер; руководящий сотрудник; директор. Используя виртуальные функции, не зная с объектом какого класса вы работаете, выведите на экран целое число - уровень допуска, и название должности.

10. Разработать программу с использованием наследования классов, реализующую классы: молодой человек; студент; военнослужащий; военный курсант. Используя виртуальное наследование и виртуальные функции, не зная с объектом какого класса вы работаете, выведите на экран сведения о военнообязанности.

Контрольные вопросы

1. Для чего нужны виртуальные функции?
2. Влияет ли наличие виртуальных функций на размер класса?
3. Может ли виртуальная функция быть дружественной функцией класса?
4. Наследуются ли виртуальные функции?
5. Может ли конструктор быть виртуальным? А деструктор?

1.5 Абстрактные типы

Класс, содержащий хотя бы один чисто виртуальный метод, называется абстрактным.

Абстрактные классы предназначены для представления общих понятий, которые предполагается конкретизировать в производных классах. Абстрактный класс может использоваться только в качестве базового для других классов – объекты абстрактного класса создавать нельзя, поскольку прямой или косвенный вызов чисто виртуального метода приводит к ошибке при выполнении.

При определении абстрактного класса необходимо иметь в виду следующее:

• абстрактный класс нельзя использовать при явном приведении типов, для описания типа параметра и типа возвращаемого функцией значения;

• допускается объявлять указатели и ссылки на абстрактный класс, если при инициализации не требуется создавать временный объект;

• если класс, производный от абстрактного, не определяет все чисто виртуальные функции, он также является абстрактным.

Таким образом, можно создать функцию, параметром которой является указатель на абстрактный класс. На место этого параметра при выполнении программы может передаваться указатель на объект любого производного класса. Это позволяет создавать полиморфные функции работающие с объектом любого типа в пределах одной иерархии.

Создадим букет, состоящий из цветов и аксессуаров и посчитаем стоимость букета. Для этого опишем абстрактный класс Элемент букета, содержащий чисто виртуальный метод, который вычисляет стоимость. А в производных классах этот метод будем реализовывать.

```
#include <iostream>
#include <string>
using namespace std;
class bouquet_el{
   public:
     virtual double cost()=0;
};
class flower: public bouquet_el{
   protected:
```

```
    double length;
    public:
      flower(double length){
         this->length = length;
      }
      virtual double cost()=0;
};
class rose: public flower{
    string color;
    public:
    rose(double length, string color):flower(length){
       this->color = color;
    }
    double cost(){
       if (color =="pink" && length>50)
          return 79;
       if (color == "red" && length >50)
          return 80;
       return 70;
    }


};
class chamomile: public flower{
    string grade;
    public:
    chamomile(double length, string grade):flower(length){
       this->grade = grade;
    }
    double cost(){
       if (grade =="colored_variety")
          return 100;
       return 50;
    }

};
class wrapping_paper:public bouquet_el{
    int size;
    public:
      wrapping_paper(int size){
         this->size = size;
      }
      double cost(){
```

```
        return size*4.8;
    }
};
int main()
{
   bouquet_el* bouquet[4];
   bouquet[0]=new rose(70,"pink");
   bouquet[1]=new rose(70,"red");
   bouquet[2]=new chamomile(40,"colored_variety");
   bouquet[3]=new wrapping_paper(8);
   double sum = 0;
   for(int i=0; i<4; ++i)
     sum+=bouquet[i]->cost();
   cout<< sum;
   return 0;
}
```

Задания для самостоятельного выполнения

1. Создать абстрактный базовый класс *Number* с виртуальными методами – арифметическими операциями сложения, вычитания, умножения, деления, возведения в степень, получения остатка от деления. Создать производные классы *Integer* и *Real*.

2. Создать абстрактный базовый класс *Figure* с виртуальными методами вычисления площади и периметра. Создать производные классы: *Rectangle*, *Circle*, *Trapezium* со своими функциями площади и периметра.

3. Создать абстрактный базовый класс *Function* с виртуальными методами вычисления значения функции $y = f(x)$ в заданной точке x и вывода результата на экран. Определить производные классы *CubePolinom*, *Hyperbola* с собственными функциями вычисления y в зависимости от входного параметра x.

4. Создать абстрактный базовый класс *Body* с виртуальными функциями вычисления площади поверхности и объема. Создать производные классы: *Parallelepiped* и *Ball* со своими функциями площади поверхности и объема.

5. Создать абстрактный класс *Currency* для работы с денежными суммами. Определить виртуальные функции перевода в рубли и вывода на экран. Реализовать производные классы *Dollar* и *Euro* со своими функциями перевода и вывода на экран.

6. Создать абстрактный базовый класс *Root* с виртуальными методами вычисления корней и вывода результата на экран. Определить производные классы *Linear* (линейное уравнение) и *Square* (квадратное уравнение) с собственными методами вычисления корней и вывода на экран.

7. Создать абстрактный базовый класс *Progression* с виртуальными функциями вычисления *n*-го элемента прогрессии и суммы прогрессии. Определить производные классы: *Arithmetical* (арифметическая) и *Geometrical* (геометрическая).

8. Создать абстрактный класс *Norm* с виртуальной функцией вычисления нормы и модуля. Определить производные классы *Complex, Vector3D* с собственными функциями вычисления нормы и модуля.

9. Создать абстрактный базовый класс *Pair* с виртуальными арифметическими операциями и операциями сравнения. Реализовать производные классы *Complex* (комплексное число) и *Rational* (рациональное число). В классе *Rational* предусмотреть сокращение дроби, используя алгоритм Евклида.

10. Создать абстрактный базовый класс *Pair* с виртуальными арифметическими операциями и операциями сравнения. Создать производные классы *Money* и *Fraction*. Число в классе *Money* представлено двумя полями: типа long для рублей и типа unsigned char для копеек.

Контрольные вопросы

1. Для чего нужны абстрактные классы? Какое ключевое слово используется при объявлении абстрактных методов?
2. Верно ли, что абстрактный класс не может наследоваться от неабстрактного?
3. Какие члены класса могут быть определены как абстрактные?
4. Являются ли абстрактные методы виртуальными?
5. Можно ли создавать цепочку производных классов, используя абстрактный класс?

2. СТАНДАРТНАЯ БИБЛИОТЕКА ЯЗЫКА С++

2.1 Класс string

Язык С++ поддерживает массивы символов, завершаемые нуль-символом, а также, описанный в стандартной библиотеки тип данных string. Базовые операции со строками выполняются в нем с помощью операций и методов, а длина строки изменяется динамически в соответствии с потребностями.

Операции со строками

Операция	Действие
=	присваивание
+	конкатенация
==	равенство
!=	неравенство
<	меньше

<=	меньше или равно
>	больше
>=	больше или равно
[ ]	индексация
<<	вывод
>>	ввод
+=	добавление

Присваивание и добавление частей строк

Для присваивания части одной строки другой строке служит функция **assign:**

assign(const string& s);
assign(const string& s, size_type pos, size_type n);
assign(const char* str, size_type n);

Здесь pos – позиция строки s, n – количество символов строки s.

Для добавления части одной строки к другой служит функция **append:**

append(const string& s);
append(const string& s, size_type pos, size_type n);
append(const char* str, size_type n);

Здесь pos – позиция строки s, n – количество символов строки str.

Для вставки в одну строку части другой строки служит функция **insert:**

insert(size_type pos1, const string& s);
insert(size_type pos1, const string& s, size_type pos2, size_type n);
insert(size_type pos, const char* str, size_type n);

Здесь posl - позиция вызывающей строки, начиная с которой осуществляется вставка n элементов строки s, начиная с позиции pos2.

Для удаления части строки служит функция **erase**:

erase(size_type pos = 0, size_type n = npos);

Она удаляет из вызывающей строки n элементов, начиная с позиции pos.

Очистку всей строки можно выполнить с помощью функции **clear**:

void clear();

Для **замены** части строки служит функция **replace:**

replace(size_type posl, size_type n1, const string& s);
replace(size_type pos1, size_type nl, const string& s, size_type pos2,size_type n2):

Здесь posl – позиция вызывающей строки, начиная с которой выполняется замена, nl – количество удаляемых элементов, pos2 – позиция строки s, начиная с которой она вставляется в вызывающую строку, n2 – количество вставляемых элементов строки s.

Для обмена содержимого двух строк служит функция **swap:**

swap(string& str);

Для выделения части строки служит функция **substr:**

string substr(size_type pos = 0, size_type n = npos) const;

Эта функция возвращает подстроку вызываемой строки длиной n, начиная с позиции pos.

Поиск подстрок

size_type find(const string& s, size_type pos = 0) const;

Ищет самое первое вхождение строки s в вызывающую строку, начиная с позиции pos, и возвращает позицию строки или npos, если строка не найдена.

size_type rfind(const string& s, size_type pos = npos) const;

Ищет самое последнее вхождение строки s в вызывающую строку, до позиции pos, и возвращает позицию строки или npos, если строка не найдена.

size_type find_first_of(const string& s, size_type pos = 0) const;

Ищет самое первое вхождение любого символа строки s в вызывающую строку, начиная с позиции pos, и возвращает позицию символа или npos, если вхождение не найдено.

size_type find_last_of(const string& s, size_type pos = npos) const;

Ищет самое последнее вхождение любого символа строки s в вызывающую строку, начиная с позиции pos, и возвращает позицию символа или npos, если вхождение не найдено.

size_type find_first_not_of(const string& s, size_type pos = 0) const;

Ищет самую первую позицию, начиная с pos, для которой ни один символ строки s не совпадает с символом вызывающей строки.

size_type find_last_not_of(const string& s, size_type pos = npos) const;

Ищет самую последнюю позицию до pos, для которой ни один символ строки s не совпадает с символом вызывающей строки.

Сравнение частей строк

Для сравнения строк целиком применяются перегруженные операции отношения, а если требуется сравнивать части строк, используется функция compare:

int compare(const string& s) const;

int compare(size_type posl, size_type nl, const string& s) const;

int compare(size_type posl, size_type n1, const string& s, size_type pos2, size_type n2) const;

Здесь n1 – количество символов вызывающей строки, pos1 – позиция вызывающей строки, начиная с которой осуществляется сравнение, pos2 – позиция строки s, n2 – количество символов строки s. Функция возвращает значение, равное нулю, если строки одинаковы, и большее нуля – если вызывающая строка больше.

Получение характеристик строк

В классе string определено несколько методов, позволяющих получить длину строки и объем памяти, занимаемый объектом:

Количество элементов строки

size_type size() const;

Длина строки
size_type length() const;
Максимальная длина строки
size_type max_size() const;
Объем памяти, занимаемый строкой
size_type capacity() const;
Проверяет пустая ли строка и возвращает true, если строка пустая
bool empty() const;

Задания для самостоятельного выполнения

1. Дано предложение из W слов. Определить количество различных слов в предложении. (Например, "Мчатся тучи, вьются тучи" → 3).

2. Даны два предложения из W и W_1 слов. Напечатать все слова, которые встречаются в двух предложениях только один раз. (Например, "Я хочу машину. Я хочу мороженое." → "машину" "мороженое").

3. Дана последовательность натуральных чисел N ($N < 10000$). Напечатать эти числа русскими словами. (Например, 3, 101, 23 → "три" "сто один" "двадцать три").

4. Дано предложение из W слов ($W > 10000$). Вывести на экран его слова, начинающиеся и оканчивающиеся на одну и ту же букву. (Например, "Мама мыла мылом Сашу" → "мылом").

5. Дано предложение из некоторого количества слов W. Напечатать его в обратном порядке слов. (Например, "Таня ест кашу" → "ушак тсе янаТ").

6. Дана строка из W слов($W > 10000$). Поменять местами его первое и последнее слово. (Например, "одинокий парус белеет" → "белеет парус одинокий").

7. Дан текст из W слов. Вывести на экран все его слова в порядке невозрастания их длин. (Например, "Он собирается эту книгу прочитать" → "собирается" "прочитать" "книгу" "эту" "он").

8. Дан текст из слов W($W > 10000$). Вывести на экран количество гласных в тексте. (Например, "Вечером должна быть гроза" → 8).

9. Дан текст из W слов ($W > 10000$). Найти длину его самого короткого слова. (Например, "Настало морозное утро. На стёклах появились узоры" → 2).

10. Дано предложение. Найти какое-нибудь его слово, начинающееся на букву, которую вводит пользователь. (Например, "Каждую осень птицы улетают на юг." → "о" "осень").

Контрольные вопросы

1. Перечислите операции над строками.
2. Какие операции сравнения применимы к строкам?
3. Как инициализировать массив строк?
4. Как в теле программы получить аргументы из командной строки?
5. В чем отличия и в чем сходство строк и массивов типа char[]?

2.2 Последовательные контейнеры

Последовательные контейнеры служат для хранения в виде последовательности однотипных элементов. К ним относятся векторы (vector), двусторонние очереди (deque) и списки (list), а также адаптеры, то есть варианты, контейнеров — стеки (stack), очереди (queue) и очереди с приоритетами (priority_queue).

Вектор — это структура, эффективно реализующая произвольный доступ к элементам, добавление в конец и удаление из конца.

Двусторонняя очередь эффективно реализует произвольный доступ к элементам, добавление в оба конца и удаление из обоих концов.

Список эффективно реализует вставку и удаление элементов в произвольное место, но не имеет произвольного доступа к своим элементам.

Векторы (vector), двусторонние очереди (deque) и списки (list) поддерживают разные наборы операций, среди которых есть совпадающие операции. Они могут быть реализованы с разной эффективностью:

Операция	Метод	vector	deque	list
Вставка в начало	push_front	-	+	+
Удаление из начала	pop_front	-	+	+
Вставка в конец	push_back	+	+	+
Удаление из конца	pop_back	+	+	+
Вставка в произвольное место	Insert	+	+	+
Удаление из произвольного места	Erase	+	+	+
Произвольный доступ к элементу	[ ], at	+	+	-

```
int main(){
	ifstream in ("text.txt");
	vector<int> w;
	int x_value;
	while ( in >> x_value, !in.eof())
		w.push_back(x_value);
	for (vector<int>::iterator iter = w.begin(); i != w.end(); ++iter)
		cout << *iter << " "<<endl;
	return 0;
}
```

Доступ к элементам вектора осуществляется с помощью следующих операций и методов:

```
reference  operator[](size_type n);//возвращают значение ссылки на элемент
reference at(size_type n);//возвращают значение ссылки на элемент
reference front(); //ссылка на первый элемент вектора
reference back();//ссылка на последний элемент вектора
```

Реализован метод выделения памяти:

void reserve(size_type n);

Определены следующие методы для изменения объектов класса vector:

Метод insert служит для вставки элемента в вектор.

iterator insert(iterator pos, const T& value);

void insert(iterator pos, size_type n, const T& value);

template <class Iter>

void insert(iterator pos, Iter first, Iter last);

Метод erase служит для удаления одного элемента вектора или диапазона, заданного с помощью итераторов.

В классе **list** описаны конструкторы, операция присваивания, метод копирования, операции сравнения и итераторы, аналогичные векторам, а также методы insert, erase, swap, clear. Доступ к элементам ограничен доступом к началу и концу спискас помощью методов:

reference front();

const_reference front() const;

reference back();

const_reference back() const;

Для списков реализованы следующие методы:

void splice(iterator position, list<T>& x); //Помещает в вызывающий список перед элементом, позиция которого указана первым параметром, все элементы списка, указанного вторым параметром.

void splice(iterator position, list<T>& x, iterator i); //переносит элемент, позицию которого содержит третий параметр, из списка x в вызывающий список

void splice(iterator position, list<T>& x, iterator first_iter, iterator last_iter);//перемещает из списка в список элементы, диапазон которых задается третьим и четвертым параметрами.

Можно удалить элемент по значению:

void remove(const T& value);

Если элементов со значением value в списке несколько, все они будут удалены.

Для сортировки элементов списка по возрастанию реализован метод:

void sort();

Метод unique оставляет в списке только первый элемент из каждой серии идущих подряд одинаковых элементов:

void unique();

Для слияния списков служит метод merge:

void merge (list<T>& x);

Метод reverse служит для изменения порядка следования элементов списка на обратный:

void reverse().

Задания для самостоятельного выполнения

1. Дан дек D с количеством элементов N. Удалить средний элемент, если количество элементов нечётное, или два средних элемента, если количество элементов чётное. Использовать один вызов функции-члена erase.

2. Дана очередь Q с нечётным количеством элементов N. Добавить в начало очереди пять его средних элементов в исходном порядке. Использовать один вызов функции-члена insert.

3. Дан вектор V с нечётным количеством элементов. Добавить в середину вектора N нулевых элементов. Использовать один вызов функции-члена insert.

4. Даны список L и дек D; дек D имеет четное количество элементов. Переместить первую половину элементов дека D в начало списка L. Использовать один вызов функций-членов insert и erase

5. Дана последовательность из N действительных чисел. Найти разницу между максимальным и минимальным элементами контейнера и вычесть ее из каждого элемента контейнера.

6. Даны списки L_1 и L_2, имеющие четное количество элементов. Поменять местами первую половину исходного списка L_1 и вторую половину исходного списка L_2.

7. Дан дек D с нечетным количеством элементов N. Удалить средний элемент дека.

8. Даны списки L_1 и L_2. Список L_1 имеет нечетное количество элементов. Переместить средний элемент списка L_1 в конец списка L_2.

9. Дан список L с четным количеством элементов. Вставить после каждого элемента из первой половины исходного списка число -1.

10. Дан список L. Удалить все элементы исходного списка с четными порядковыми номерами (считая, что начальный элемент списка имеет порядковый номер 1).

Контрольные вопросы

1. Для чего служат последовательные контейнеры?
2. Перечислите последовательные контейнеры STL.
3. Перечислите основные операции с последовательными контейнерами.
4. С помощью каких методов и операций осуществляется доступ к элементам вектора?
5. Чем отличается list от vector?
6. Перечислить методы, которые поддерживает последовательный контейнер vector.
7. Перечислить методы, которые поддерживает последовательный контейнер list.
8. Перечислить методы, которые поддерживает последовательный контейнер deque.

2.3 Ассоциативные контейнеры

Ассоциативные контейнеры позволяют получить быстрый доступ к значению по ключу. Существует следующие типы ассоциативных контейнеров: словари (map), словари с дубликатами (multimap), множества (set), множества с дубликатами (multiset) и битовые множества (bitset).

Словарь построен на основе пары, первое значение которой является ключом для идентификации элемента, а второе значение – это элемент. В словарях, описанных в STL, в качестве ключа может использоваться значение произвольного типа. Ассоциативные контейнеры описаны в заголовочных файлах <map> и <set>.

Для хранения пары «ключ-значение» описан шаблон pair:

```
template <class T1, class T2> struct pair{
        typedef Tl    first_type;
        typedef T2 second_type;
        T1 first;
        T2 second;
        pair(const T1& x, const T2& y);
        template <class U, class V>pair (const pair  <U, V>&p);
};
```

Для пары установлены проверка на равенство и операция сравнения на меньше. Пара p1 меньше пары p2. если

p1.first < p2.first или

p1.first == p2.first && p1.second < p2.second

Для присваивания значения паре можно использовать функцию make_pair:

```
template <class T1, class T2>
        pair <T1, T2> make_pair (const T1& x, const T2& y);
```

Ассоциированный контейнер map

Контейнер <map> позволяет хранить данные в виде ключ-значение (книга-количество, продукт-вес\объем). Контейнер <map> является отсортированным массивом. Сортировка произведена по ключу. В словаре (map), в отличие от словаря с дубликатами (multimap), все ключи должны быть уникальны. Элементы в словаре хранятся в отсортированном виде, поэтому для ключей должно быть определено отношение «меньше».

Создание контейнера:

```
map <key type, data type> map_name;
map<string, int> books; //Содержит связку название книги / количество
```

Для доступа к элементам по ключу определена операция [ ]:

```
T& operator [] (const Key & x);
```

Для поиска элементов в словаре определены следующие функции:

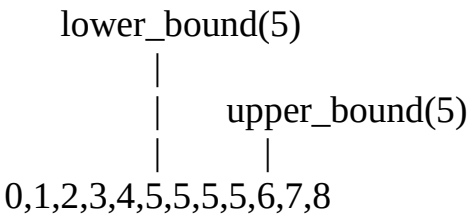
```
iterator find (const key_type& x); //итератор на первый элемент с указанным ключом
const_iterator find (const key_type& x) const;
```

iterator lower_bound (const key_type& x); // возвращает итератор на первый элемент, ключ которого не меньше x, или end(), если такого нет

const_iterator lower_bound (const key__type& x) const;

iterator upper_bound (const key_type& x); // итератор на первый элемент, чей ключ больше указанного ключа

const_iterator upper_bound (const key_type &x) const;



size_type count (const key_type& x) const; // число элементов соответствующих указанному ключу. Для класса map значения 1 или 0

Для вставки и удаления элементов определены функции:

pair <iterator, bool> insert (const value_type& x);// Метод применяется для вставки в словарь пары «ключ-значение» и возвращает пару, состоящую из итератора, указывающего на вставленное значение, и булевого признака результата операции (true, если записи с таким ключом в словаре не было (только в этом случае происходит добавление), и false в противном случае (итератор указывает на существующую запись)).

iterator insert (iterator position, const value_type& x); // первым параметром передается позиция словаря, начиная с которой требуется осуществлять поиск места вставки. Вставка выполняется только в случае отсутствия значения x в словаре. Функция возвращает итератор на элемент словаря с ключом, содержащимся в x.

template <class InputIter>

void insert (InputIter first, InputIter last); // используется для вставки группы элементов, определяемой диапазоном итераторов

void erase (iterator position); // удаляет элемент словаря из позиции, заданной итератором

size_type erase (const key_type& x); // удаляет элемент по заданному ключу

void erase (iterator first, iterator last); // удаляет диапазон элементов

void clear ();

Для обмена всех элементов двух словарей применяется функция swap:

template <class key, class T, class Compare>void swap (map<key, T, Compare>& x, map<key, T, Compare>& y);

Функция equal_range возвращает пару итераторов (lower_bound(x), upper_bound(x)) для переданного ей значения x:

pair<iterator, iterator> equal_range (const key_type& x);

pair<const iterator, const iterator> equal_range (const key_type& x) const;

После вызова функции оба итератора будут указывать на элемент с заданным ключом, если он присутствует в словаре, или на первый элемент, больший него, в противном случае.

Словари с дубликатами (multimap)

Словари с дубликатами (multimap) допускают хранение элементов с одинаковыми ключами. Благодаря этому для них не определена операция доступа по индексу [], а добавление с использованием метода insert выполняется успешно в любом случае. Функция возвращает итератор на вставленный элемент. Элементы с одинаковыми ключами хранятся в словаре в порядке их занесения. При удалении элемента по ключу функция erase возвращает количество удаленных элементов. Функция equal_range возвращает диапазон итераторов, определяющий все вхождения элемента с заданным ключом. Функция count может вернуть значение, которое больше единицы.

Задания для самостоятельного выполнения

1. Написать программу, моделирующую работу с предметным указателем в книге.

Каждая компонента указателя содержит слово и номера страниц, на которых это слово встречается. Количество номеров страниц, относящихся к одному слову, лежит в диапазоне от одного до десяти. Предметный указатель хранится в текстовом файле.

Программа должна обеспечивать выполнение следующих функций:

– начальное формирование предметного указателя;

– вывод предметного указателя;

– вывод номеров страниц для заданного слова.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок при вводе.

Для представления предметного указателя в ОП использовать класс tree, реализующий бинарное дерево.

2. Написать программу работы с базой отдела кадров предприятия. База хранится в текстовом файле, его размер может быть произвольным.

Каждая строка файла содержит запись об одном сотруднике. Формат записи: фамилия и инициалы (30 поз., фамилия должна начинаться с первой позиции), год рождения (5 поз.), оклад (10 поз.).

Программа должна обеспечивать:

– начальное формирование и дополнение базы данных о сотрудниках;

– корректировку сведений о сотрудниках;

– поиск сотрудника в базе по фамилии;

– поиск самого высокооплачиваемого сотрудника;

Для представления базы в ОП использовать класс tree, реализующий бинарное дерево.

3. Написать программу «Моя записная книжка».

Предусмотреть возможность работы с произвольным числом записей, поиска записи по какому-либо признаку (например, по фамилии, дате рождения или номеру телефона), добавления и удаления записей, сортировки по разным полям.

Хранение данных организовать с применением класса list.

4. Написать программу учета книг в библиотеке.

Сведения о книгах содержат: фамилию и инициалы автора, название, год издания, количество экземпляров данной книги в библиотеке.

Программа должна обеспечивать выбор с помощью меню и выполнение следующих функций:

– добавление данных о книгах, вновь поступающих в библиотеку;

– удаление данных о списываемых книгах;

– выдача сведений о всех книгах, упорядоченных по фамилиям авторов;

– выдача сведений о всех книгах, упорядоченных по годам издания.

Хранение данных организовать с применением класса vector.

5. Составить программу учета заявок на авиабилеты.

Каждая заявка содержит: пункт назначения, номер рейса, фамилию и инициалы пассажира, желаемую дату вылета. База хранится в текстовом файле.

Программа должна обеспечивать выбор с помощью меню и выполнение одной из следующих функций:

– добавление заявок в список;

– удаление заявок;

– вывод заявок по заданному номеру рейса и дате вылета;

– вывод всех заявок, упорядоченных по пунктам назначения;

– вывод всех заявок, упорядоченных по датам вылета.

Хранение данных организовать с применением класса vector.

6. На телефонной станции картотека абонентов, содержащая сведения о телефонах и их владельцах, организована в виде линейного списка и хранится в файле.

Написать программу, которая:

– обеспечивает начальное формирование картотеки;

– производит вывод всей картотеки;

– вводит номер телефона, дату и время разговора;

– выводит извещение на оплату телефонного разговора.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок при вводе.

7. Написать программу, моделирующую управление каталогом в файловой системе.

Для каждого файла в каталоге содержатся следующие сведения: имя файла, дата создания, количество обращений к файлу. База хранится в текстовом файле, его размер может быть произвольным.

Программа должна обеспечивать:

– начальное формирование каталога файлов в виде списка;

– вывод каталога файлов;

– обращение к файлу;

– удаление файлов, дата создания которых раньше заданной;

– выборку файла с наибольшим количеством обращений.

Выбор моделируемой функции должен осуществляться с помощью меню. Для представления базы в ОП использовать класс list, реализующий линейный двусвязный список.

8. Написать программу для моделирования работы Т-образного сортировочного узла на железной дороге с использованием стека.

Программа должна разделять на два направления состав, состоящий из вагонов двух типов (на каждое направление формируется состав из вагонов одного типа). Предусмотреть возможность ввода исходных данных с клавиатуры и из файла.

9. Написать программу, моделирующую заполнение гибкого магнитного диска.

Общий объем памяти на диске 360 Кбайт. Файлы имеют произвольную длину от 18 байт до 32 Кбайт. В процессе работы файлы либо записываются на диск, либо удаляются с него.

В начале работы файлы записываются подряд друг за другом. После удаления файла на диске образуется свободный участок памяти, и вновь записываемый файл либо размещается на свободном участке, либо, если файл не помещается в свободный участок, размещается после последнего записанного файла.

В случае, когда файл превосходит длину самого большого свободного участка, выдается аварийное сообщение. Требование на запись или удаление файла задается в строке, которая содержит имя файла, его длину в байтах, признак записи или удаления. Программа должна выдавать по запросу сведения о занятых и свободных участках памяти на диске.

10. Написать программу «Англо-русский и русско-английский словарь».

«База данных» словаря содержит по одному варианту перевода слов и хранится в текстовом файле.

Программа должна обеспечивать выбор с помощью меню и выполнение одной из следующих функций:

Формирование «базы данных» словаря.

Выбор режима работы: англо-русский; русско-английский.

Вывод перевода заданного английского слова.

Вывод перевода заданного русского слова. Базу данных словаря реализовать в виде класса vector.

Контрольные вопросы

1. Для чего служат ассоциативные контейнеры?
2. Перечислите ассоциативные контейнеры STL.
3. Что означает имя iterator в области видимости ассоциативного контейнера set?
4. Что будет, если в Map положить два значения с одинаковым ключом?
5. Перечислить методы, которые поддерживает ассоциативный контейнер Map.

2.4 Итераторы

Итератор – это специальный класс, который обобщает понятие указателя для работы с различными структурами данных единообразным способом.

Прямой итератор(forward) поддерживает все операции входных(input) и выходных(output) итераторов и может использоваться везде, где требуются входные или выходные итераторы. Двунаправленный(bidirectional) итератор поддерживает все операции прямого, а также декремент, и может использоваться везде, где требуется прямой итератор. Итератор произвольного доступа (random access) поддерживает все операции двунаправленного, а кроме того, переход к произвольному элементу последовательности и сравнение итераторов. Можно сказать, что итераторы образуют иерархию, на верхнем уровне которой находятся итераторы произвольного доступа. Чем выше уровень итератора, тем более высокие функциональные требования предъявляются к контейнеру, для которого используется итератор. Например, для списков итераторами произвольного доступа пользоваться нельзя, поскольку список не поддерживает требуемый набор операций итератора.

Так как итераторы применяются для работы с объектами, на которые они указывают (например, для получения значения элемента контейнера), необходимо описать соответствующие типы.

Для этого в заголовочном файле <Iterator>определен шаблон Iterator_traits.

```
template <class Iter>struct iterator_traits{
typedef typename Iter: :difference_type difference_type;
 typedef typename Iter::value_type value_type;
typedef typename Iter::pointer pointer;
 typedef typename Iter::reference reference;
 typedef typename Iter: :iterator_category iterator_category;
}
```

Ключевое слово typename требуется для того, чтобы компилятор мог идентифицировать Iter как имя типа. Iterator_category — это категория итератора, регламентирующая, какие он поддерживает операции. Тип difference_type предназначен для определения разности между двумя итераторами.

Так как только итераторы random_access поддерживают операции + и -, в библиотеке описаны функции distance и advance. Функция distance служит для определения расстояния между элементами контейнера:

distance (Inputlterator first, Inputlterator last);

Она возвращает значение типа difference_type, который является разностью между двумя итераторами. Эта функция для всех итераторов, кроме итераторов произвольного доступа, реализлована с помощью операции инкремент за время, которое пропорционально расстоянию между элементами контейнера.

Аналогично работает и функция advance, которая применяется для реализации операции i += n:

void advance(InputIterator& I, Distance n);

Параметр n может быть отрицательным только для двунаправленных итераторов и итераторов произвольного доступа.

Обратные итераторы

Для двунаправленных итераторов и итераторов произвольного доступа определены разновидности, называемые адаптерами итераторов.

Адаптер, просматривающий последовательность в обратном порядке, называется reverse_ iterator.

Reverse_ iterator содержит защищенное поле данных, называемое текущим итератором - current. Операция инкремента реализуется путем декремента этого итератора:

```
template <class Iter>
  reverse_iterator<Iter>& reverse_iterator<Iter> operator++(){
      --current;
  return *this;
}
```

Итератор может указывать на все элементы контейнера, включая следующий за последним, но для обратного итератора таким элементом будет элемент, стоящий перед первым, а его не существует.

Поэтому current указывает на элемент, следующий за тем, на который указывает обратный итератор. Между прямым и обратным итератором существует зависимость:

&*(reverse_iterator(i)) == &*(i - 1).

Для обратных итераторов определены операции отношения =, !=, <= и >=.

Обратные итераторы описаны в контейнерных классах для перебора их элементов в обратном порядке. Также реализованы методы rbegin() и rend(), возвращающие reverse_iterator.

Итераторы вставки

Итераторы вставки являются адаптерами итераторов. Они используются для добавления новых элементов в начало, конец или произвольное

место контейнера. В стандартной библиотеке определено три шаблона классов итераторов вставки, построенных на основе выходных итераторов:

back_insert_iterator,
front_insert iterator и
insert_iterator.

Также реализованы три метода вставки:

```
template <class c> back_insert_iterator <c>
        back_inserter(C& x);
 template <class c> front_insert_iterator<c>
 front_inserter(C& x);
template <class c> insert_iterator <c>
inserter(C& x, Iter i);
```

Потоковые итераторы

Итератор входного потока извлекает данные из потока, для которого был создан, и затем к ним можно обращаться через операцию разадресации.

Например, для чтения целого числа из файла с именем temp можно использовать следующий фрагмент:

```
ifstream in(“temp");
istream_iterator i(in);
int buf = *i;
```

Как только достигается конец входного потока, итератор принимает значение конца ввода. Это же значение имеет конструктор итератора по умолчанию, поэтому цикл чтения из файла можно организовать следующим образом:

```
while ( i != istream_iterator () )
cout << *i++ << " ";
```

Для итераторов входного потока предопределены операции сравнения на равенство и неравенство. Итераторы, которые равны концу ввода,являются равными между собой. Сравнивать не равные концу ввода итераторы можно только в случае, что они сконструированы для одного и того же потока. Спецификой итераторов входного потока является то, что они не сохраняют равенство после инкрементации, то есть если i == j, то не обязательно, что ++i == ++j.

Итератор выходного потока записывает с помощью операции << элементы в выходной поток, для которого он был создан. Вторым параметром конструктора может быть строка символов, которая выводится после каждого значения:

```
ostream_iterator ostr(cout, " см");
*ostr = 59: // Будет выведено: 59 см
++ostr;
 *ostr = 27; // Будет выведено: 27 см
```

Задания для самостоятельного выполнения

1. Дан текстовый файл number.txt, содержащий строковые представления вещественных чисел. Используя итератор istream_iterator, найти количество неотрицательных чисел в исходном файле.

2. Дана строка *S* и некоторый набор символов. Используя итераторы ptin_iterator и ostream_iterator, записать в текстовый файл symbol.txt исходный набор символов в том же порядке, добавляя после каждого символа пробел.

3. Дано натуральное число *N*, текстовый файл words.txt, содержащий русские слова, и строка *S*. Используя итераторы istream_iterator и ostream_iterator записать в текстовый файл words2.txt все слова из исходного файла, длина которых не превосходит *N*, сохранив исходный порядок их следования и располагая каждое слово на новой строке.

4. Дан некоторый набор символов. Используя итераторы ptin_iterator и ptout_iterator, вывести все символы из исходного набора в том же порядке, заменяя цифровые символы на символ подчеркивания.

5. Дан текст *W* и натуральное число *N*. Используя итератор ostream_iterator записать в текстовый файл w2.txt *N* символов «#».

6. Дана некоторая строка *S* и набор целых чисел. Используя итераторы ptin_iterator и ostream_iterator, записать в текстовый файл с именем S.txt все числа из исходного набора в том же порядке, заменяя каждое число 1 на число 0 и добавляя после каждого числа три пробела.

7. Даны вещественные числа *A, D* и натуральное число *N*. Используя итераторы ptout_iterator, вывести *N* первых членов арифметической прогрессии с первым элементом *A* и разностью *D* и сумму арифметической прогрессии.

8. Дана строка *S* и набор символов. Используя итераторы ptin_iterator и ostream_iterator, записать в текстовый файл S.txt утроенные кодовые значения всех символов из исходного набора в том же порядке, добавляя после каждого числа два пробела.

9. Даны вещественные числа *G, Q* и целое число *N*. Используя итератор ptout_iterator, вывести *N* первых членов геометрической прогрессии с первым элементом *G* и знаменателем *Q*.

10. Дан некоторый набор вещественных чисел *N*. Используя итераторы ptin_iterator и ptout_iterator, вывести числа из исходного набора с нечетными порядковыми номерами.

Контрольные вопросы

1. Что такое итератор?
2. В чем разница между итераторами и указателями?
3. Каким образом может быть реализован итератор?
4. Какие типы итераторов существуют?
5. Что задает второй аргумент ostream_iterator?

2.5 Алгоритмы

Для использования стандартных алгоритмов необходимо подключить заголовочный файл <algorithm>.

Алгоритм adjacent_find ищет пары соседних значений.

```
template <class For> For adjacent_find(For first, For last);
template <class For, class BinPred>
For adjacent_find(For first, For last, BinPred pred);
```

Первая реализация выполняет поиск в последовательном контейнере пары соседних совпадающих значений и возвращает итератор на первое из них или следующий за последним, если значения не найдены. Второй вариант находит соседние элементы, удовлетворяющие условию, который задан предикатом pred в виде функции или функционального объекта.

Алгоритм count выполняет подсчет количества вхождений значения в последовательность:

```
template <class In, class T>
    typename iterator_traits<In>::difference_type
      count(In first, In last, const T& value);
```

Данная реализация вычисляет в последовательном контейнере количество вхождений заданного значения value. Результат имеет тип разности между двумя итераторами difference_type.

Алгоритм count_if выполняет подсчет количества выполнений условия в последовательности:

```
template <class In, class Pred>
    typename iterator_traits<In>::difference_type
      count_if(In first, In last, Pred pred);
```

Эта форма находит в последовательном контейнере количество элементов, удовлетворяющих условию, заданному предикатом pred в виде функции или функционального объекта.

Алгоритм equal попарно сравнивает элементы двух последовательностей. Пользователь может задать предикат, который определяет, что считать равенством:

```
template <class In1, class In2>
   bool equal(In1 first1, In1 last1, In2 first2);
template <class In1, class In2, class BinPred>
    bool equal(In1 first1, In1 last1, In2 first2, BinPred pred);
```

Алгоритмы семейства find выполняют поиск в последовательности. Алгоритм find осуществляет поиск заданного значения value:

```
template <class In, class T>
    In find (In first, In last, const T& value);
```

Алгоритм find_if выполняет поиск элемента, удовлетворяющего заданному предикату pred:

```
template <class In, class Pred>
    In find_if (In first, In last, Pred pred);
```

Алгоритм find_first_of находит первое вхождение в первую последовательность элемента из второй последовательности:

template<class For1, class For2>

For1 find_first_of(Forl firstl, For1 lastl, For2 first2, For2 last2);

template<class For1, class For2,class BinPred>

For1 find_first_of(Forl firstl, For1 lastl, For2 first2, For2 last2, BinPred pred);

Границы последовательностей заданы с помощью итераторов. Первый вариант алгоритма находит вхождение любого элемента, а вторая - элемента, для которого выполняется бинарный предикат, анализирующий соответствующие элементы первой и второй последовательности. В случае неудачного поиска возвращается lastl.

Алгоритм find_end находит первое вхождение в первую последовательность второй последовательности (с анализом предиката или без) и возвращает итератор на последний совпадающий элемент:

template<class For1, class For2>

For1 find_end(Forl firstl, For1 lastl, For2 first2, For2 last2);

template<class For1, class For2,class BinPred>

For1 find_end(Forl firstl, For1 lastl, For2 first2, For2 last2, BinPred pred);

В случае неудачного поиска возвращается lastl.

Алгоритм for_each для всех элемента последовательности применяет заданную функцию:

template <class In, class Function>

Function for_each(In first, In last, Function f);

Алгоритм mismatch осуществлякт поиск первой пары несовпадающих элементов двух последовательностей и возвращает итераторы на эту пару:

template<class In1, class In2>

pair<In1, In2> mismatch (Inl first1, Inl lastl, In2 first2);

template<class In1, class In2, class BinPred>

pair<In1, In2> mismatch (Inl first1, Inl lastl, In2 first2, BinPred pred);

Длина второй последовательности должна быть большей или равной длине первой. Пользователь может задать предикат, определяющий, что считать несовпадением.

Алгоритм search находит первое вхождение в первую последовательность второй последовательности (с анализом предиката или без) и возвращает итератор на первый совпадающий элемент:

template<class For1, class For2>

For1 search(Forl firstl, For1 lastl, For2 first2, For2 last2);

template<class For1, class For2, class BinPred>

For1 search(Forl firstl, For1 lastl, For2 first2, For2 last2, BinPred pred);

В случае неудачного поиска возвращается lastl.

Алгоритм search_n находит в последовательности подпоследовательность, которая состоит из по крайней мере n значений value (с анализом предиката или без) и возвращает итератор на первый совпадающий элемент:

template<class For, class Size, class T>

For search_n(For first, For last, Size count, const T& value);

template<class For, class Size, class T, class BinPred>

For search_n(For first, For last, Size count, const T& value, BinPred pred);

Алгоритм copy копирует, начиная с первого элемента последовательности, пределы которой задаются итераторами first и last, в выходную последовательность, для которой задается итератор начала result:

template<class In, class Out>

Out copy(In first, In last, Out result);

Алгоритм copy_backward копирует, начиная с последнего элемента заданной последовательности. Третий параметр должен указывать на элемент, следующий за последним элементом приемника, поскольку его значение уменьшается на шаг перед операцией копирования каждого элемента:

template<class Bi1, class Bi2>

Bi2 copy_backward(Bi1 first, Bi1 last, Bi2 result);

```
int main(){
  int b[4], a[5]={1, 2, 3, 4, 5}, i;
 copy (a + 1, a + 5, b);
 for (i = 0; i < 4; i++)
 cout << b[i] ; //234 5
 copy (a + 1, a + 5, a);
for (i = 0;i < 5; i++)
        cout << a[i] ; //2345 5
copy_backward (b, b + 3, b + 4);
for (i = 0; i < 4; i++)
        cout << b[i] ; //223 4
return 0;
}
```

Алгоритм fill заменяет все элементы последовательности, определенной с помощью итераторов first и last, значением value. Алгоритм fill_n выполняет замену n элементов заданным значением:

template <class For, class T>

void fill(For first, For last, const T& value);

template <class Out, class Size, class T>

void fill_n(Out first, Size n, const T& value);

Алгоритм generate выполняет замену всех элементов результатом операции. Это позволяет заполнить контейнер не одинаковыми значениями, а вычисленными с помощью функции или функционального объекта gen, заданного третьим параметром.

```
template <class For, class Generator>
        void generate(For first, For last, Generator gen);
        #include<iostream>
        #include<algorithm>
         using namespace std;
         int func(){
            static int i = 1;
            return (++i) * 3;
        }
        int main(){
           int a[5], i;
           generate(a, a + 5, func);
           for (i=0 ; i<5; ++i)//6 9 12 15 18
                cout<< a[i];
           return 0;
       }
```

Алгоритмы семейства remove выполняют перемещение в конец последовательности элементов с заданным значением value или по предикату pred. Те элементы, которые остались, перемещаются в начало последовательности с сохранением их относительного порядка. Алгоритм возвращает границу их расположения. Элементы, которые размещаются после границы, не удаляются, размер последовательности не изменяется. Формы алгоритма, содержащие слово copy, перед обработкой копируют последовательность на место, заданное итератором Out, и обрабатывают копию последовательности.

```
 template <class For, class T>
     For remove(For first, For last, const T& value);
 template <class For, class Pred>
     For remove_if(For first, For last, Pred pred);
 template <class In, class Out, class T>
   Out remove_copy(In first, In last, Out result, const T& value);
 template <class In, class Out, class Pred>
  Out remove_copy_if(In first, In last, Out result, Pred pred);

vector::iterator  p = remove(a.begin(), a.end(), 2);
bool In_10_50 (int X) {
return x > 10 && x < 50;
}
vector::iterator new_end = remove_if(a.begin(), a.end(), In_10_50);
```

Алгоритмы семейства replace осуществляют замену элементов с заданным значением на новое значение или в соответствии с предикатом.

Формы алгоритма, содержащие слово copy, предварительно копируют последовательность на место, заданное итератором Out, и обрабатывают копию последовательности.

```
template  <class For, class T>
    void replace(For first, For last, const T& old_value, const T& new_value);
template <class For, class Pred, class T>
     void replace_if (For first, For last, Pred pred, const T& new_value);
 template < class In, class Out, class T>
     Out replace_copy(In first, In last, Out result, const T& old_value, const T& new_value);
 template <class Iterator, class Out, class Pred, class T>
    Out replace_copy_if(Iterator first, Iterator last, Out result, Pred pred, const T& new_value);
```

Задания для самостоятельного выполнения

1. Дан список *L* из *N* действительных чисел. Используя вызовы алгоритма find и вызовы функции-члена erase, удалить все нулевые элементы списка *L*. Если нулевых элементов нет, то список не изменять, если список состоит из всех нулей, то вывести пустую строку.

2. Дан список *L* из *N* действительных чисел. Используя вызовы алгоритма find_if и вызовы функции-члена erase, удалить все элементы списка, которые делятся на 4, а остальные оставить в таком же порядке. Если список не содержит элементов делящихся на 4, то вывести все элементы, если список состоит из всех элементов делящихся на 4, то вывести пустую строку.

3. Дан список *L* из *N* действительных чисел. Используя алгоритм partition и вызов функции distance для итераторов, вывести количество отрицательных и неотрицательных чисел в исходном списке.

4. Дан вектор *V* с чётным количеством элементов из 2*N* действительных чисел. Используя два вызова алгоритма accumulate найти среднее арифметическое элементов из первой и среднее геометрическое из второй половины вектора.

5. Дано натуральное число *N* и вектор *V*, содержащий только единицы и нули. Используя алгоритм search_n и функцию-член erase, а также обратные итераторы, удалить в векторе *V* первый набор из *N* подряд расположенных единиц (если в этом наборе имеется больше *N* единиц, то требуется удалить только последние *N* из них). Если вектор не содержит требуемого набора единиц, то не изменять его.

6. Дан список *L* из *N* действительных чисел. Используя два вызова алгоритма stable_partition, переставить элементы списка так, чтобы сначала шли отрицательные, затем нулевые, а затем положительные элементы.

7. Дан дек *D* из *W* слов, элементами которого являются русские слова. Используя алгоритм sort и алгоритм stable_sort, отсортировать его элементы

по возрастанию их длин в алфавитном порядке, а элементы одинаковой длины – в алфавитном порядке.

8. Дан вектор *V* состоящий из *W* русских слов. Используя алгоритм accumulate с параметром (функциональным объектом), получить строку, содержащую начальные символы всех элементов вектора, расположенные в обратном порядке.

9. Дан вектор *V* с чётным количеством элементов. Известно, что первая половина вектора уже отсортирована по возрастанию. Отсортировать все элементы вектора по возрастанию, выполнив вначале сортировку его второй половины алгоритмом sort, а затем слияние обеих половин алгоритмом inplace_merge. Выводить новое содержимое вектора *V* после применения каждого алгоритма.

10. Дан список *L*, содержащий как отрицательные, так и положительные элементы. Вставить нулевой элемент после первого отрицательного элемента и перед последним положительным элементом. Использовать два вызова алгоритма find_if и два вызова функции-члена insert.

Контрольные вопросы

1. Перечислите группы алгоритмов STL.

2. Можно ли в алгоритме merge использовать не отсортированные последовательности данных?

3. Верно ли, что одной из функций итераторов STL является связывание алгоритмов и контейнеров?

4. Какая сущность зачастую используется для изменения поведения алгоритма?

ЛИТЕРАТУРА

1. Страуструп, Б. Язык программирования C++ / Б. Страуструп. – М: Бином, 2022.
2. Шупляк, В.И. C++. Практический курс: учеб. пособие / В.И. Шупляк. – Минск: Новое знание, 2011.
3. Шилдт, Г. Искусство программирования на C++/ Г. Шилдт. – СПб: БХВ-Петербург, 2005.
4. Керниган, Б. Язык программирования C / Б. Керниган, У. Ритчи, М. Денис. – 2-е изд.; пер. с англ. – М.: Издат. дом «Вильямс», 2008.
5. Павловская, Т.А. C/ C++. Программирование на языке высокого уровня. – СПб.: Питер, 2021.

Учебное издание

КОРЧЕВСКАЯ Елена Алексеевна
СТЕПАНОВ Владимир Александрович

АЛГОРИТМИЗАЦИЯ И ПРОГРАММИРОВАНИЕ

Методические рекомендации

Технический редактор *Г.В. Разбоева*
Компьютерный дизайн *В.Л. Пугач*

Подписано в печать 18.07.2022. Формат 60x84 $^{1}/_{16}$. Бумага офсетная.
Усл. печ. л. 2,79. Уч.-изд. л. 2,12. Тираж 55 экз. Заказ 117.

Издатель и полиграфическое исполнение – учреждение образования
«Витебский государственный университет имени П.М. Машерова».

Свидетельство о государственной регистрации в качестве издателя,
изготовителя, распространителя печатных изданий
№ 1/255 от 31.03.2014.

Отпечатано на ризографе учреждения образования
«Витебский государственный университет имени П.М. Машерова».
210038, г. Витебск, Московский проспект, 33.