МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

БАШКИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Практикум на ЭВМ

Программирование на C++

Часть 3

Уфа 2008

Составитель:
Рыков В.И. Практикум на ЭВМ. Программирование на С++.. Часть 2. /Издание Башкирского ун-та. - Уфа 2008. - №№ с.

Работа посвящена методологии программирования на языке С++.

Содержит сведения технологии объектного программирования. Содержит тексты задач и, в необходимых случаях, указания по технологии их решения. Методика программирования и кодирования программ для каждого типа задач изложена в виде законченных примеров.

Является методическим обеспечением лабораторных занятий по дисциплине «Практикум на ЭВМ».

.

2008

Предисловие

Цель практикума – закрепить знания, полученные при изучении теоретической части курсов и получить практические навыки разработки объектно-ориентированных программ. Практикум охватывает все разделы объектно-ориентированного программирования на языке C++ и включает выполнение 10 лабораторных работ. Первые работы связаны с базовыми понятиями C++, такими как объекты и классы, наследование, полиморфизм и виртуальные функции, обработка событий. Следующие работы посвящены профессиональному программированию на C++ и охватывают разделы профессионального программирования, такие как перегрузка операций, шаблоны, потоковые классы, стандартная библиотека шаблонов и исключительные ситуации.

Последняя работа посвящена технологии моделирования конкретной предметной области средствами технологии объектов

1 Наследование и виртуальные функции

1.1 Цель.

Получить практические навыки формирования иерархии классов и использования статических компонентов класса.

1.2 Основное содержание работы.

Написать программу, в которой создается иерархия классов. Включить полиморфные объекты в связанный список, используя статические компоненты класса. Показать использование виртуальных функций.

1.3 Краткие теоретические сведения.

Статические члены класса.

Такие компоненты должны быть определены в классе, как **статические** (**static**). Статические данные классов не дублируются при создании объектов, т.е. каждый статический компонент существует в единственном экземпляре. Доступ к статическому компоненту возможен только после его инициализации. Для инициализации используется конструкция

тип имя_класса : : имя_данного инициализатор;

Например, int complex : : count = 0;

Это предложение должно быть размещено в глобальной области после определения класса. Только при инициализации статическое данное класса получает память и становится доступным. Обращаться к статическому данному класса можно обычным образом через имя объекта

имя_объекта.имя_компонента

К статическим компонентам можно обращаться и тогда, когда объект класса еще не существует. Доступ к статическим компонентам возможен не только через имя объекта, но и через имя класса

имя_класса : : имя_компонента

Однако так можно обращаться только к *public* компонентам.

Для обращения к *private* статической компоненте извне можно с помощью **статических компонентов-функций**. Эти функции можно вызвать через имя класса.

имя_класса : : имя_статической_функции

Пример.

```
#include <iostream>
using namespace std;
class TPoint
{
  double x,y;
  static int N;  // статический компонент – данное : количество точек
   public:

  TPoint(double x1 = 0.0,double y1 = 0.0){N++; x = x1; y = y1;}
  static int& count(){return N;} // статический компонент-функция
};
int TPoint : : N = 0; //инициализация статического компонента-данного
void main(void)
{TPoint A(1.0,2.0);
  TPoint B(4.0,5.0);
  TPoint C(7.0,8.0);
  cout<< \nОпределены ”<<TPoint : : count()<<“точки”; }
```

Указатель this.

Когда функция-член класса вызывается для обработки данных конкретного объекта, этой функции автоматически и неявно передается указатель на тот объект, для которого функция вызвана. Этот указатель имеет имя **this** и неявно определен в каждой функции класса следующим образом:

*имя_класса *const this = адрес_объекта*

Указатель this является дополнительным скрытым параметром каждой нестатической компонентной функции. При входе в тело принадлежащей классу функции this инициализируется значением адреса того объекта, для которого вызвана функция. В результате этого объект становится доступным внутри этой функции.

В большинстве случаев использование **this** является неявным. В частности, каждое обращение к нестатической функции-члену класса неявно использует this для доступа к члену соответствующего объекта.

Примером широко распространенного явного использования this являются операции со связанными списками.

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.

В 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};

В производном классе унаследованные компоненты получают статус доступа **private**, если новый класс определен с помощью ключевого слова **class**, и статус **public**, если с помощью **struct**.

Явно изменить умалчиваемый статус доступа при наследовании можно с помощью атрибутов доступа – *private, protected и public*, которые указываются непосредственно перед именами базовых классов.

Конструкторы и деструкторы производных классов.

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 в определении конструктора производного класса. Таким образом, происходит передача

аргументов от конструктора производного класса конструктору базового класса.

Например.

```
class Basis
{
        int a,b;
        public:
                Basis(int x,int y)
                {
                        a=x;b=y;
                }
};
class Inherit:public Basis
{
        int sum;
        public:
                Inherit(int x,int y, int s):Basis(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**.

Пример.

```
class base
{
  public:
    virtual void print(){cout<<"\nbase";}
};

class dir : public base
{
```

```
   public:
      void print(){cout<<"\ndir";}
};
void main()
{
 base B,*bp = &B;
 dir D,*dp = &D;
 base *p = &D;
 bp –>print(); // base
 dp –>print(); // dir
 p –>print(); // dir
}
```

Таким образом, интерпретация каждого вызова виртуальной функции через указатель на базовый класс зависит от значения этого указателя, т.е. 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, ее 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.

Абстрактные классы.

Абстрактным называется класс, в котором есть хотя бы одна чистая (пустая) виртуальная функция.

Чистой виртуальной функцией называется компонентная функция, которая имеет следующее определение:

virtual тип имя_функции (список_формальных_параметров) = 0;

Чистая виртуальная функция ничего не делает и недоступна для вызовов. Ее назначение – служить основой для подменяющих ее функций в производных классах. Абстрактный класс может использоваться только в качестве базового для производных классов.

Механизм абстрактных классов разработан для представления общих понятий, которые в дальнейшем предполагается конкретизировать. При этом построение иерархии классов выполняется по следующей схеме. Во главе иерархии стоит абстрактный базовый класс. Он используется для наследования интерфейса. Производные классы будут конкретизировать и реализовать этот интерфейс. В абстрактном классе объявлены чистые виртуальные функции, которые по сути есть **абстрактные методы**.

Пример.

```
class Base
{
   public:
      Base();                    // конструктор по умолчанию
      Base(const Base&);  // конструктор копирования
      virtual ~Base();     // виртуальный деструктор
      virtual void Show()=0;     //  чистая виртуальная функция
      // другие чистые виртуальные функции
   protected: // защищенные члены класса
   private:
      // часто остается пустым, иначе будет мешать будущим разработкам
};
class Derived: virtual public Base
{
   public:
      Derived();                          // конструктор по умолчанию
      Derived(const Derived&);       // конструктор копирования
      Derived(параметры);            // конструктор с параметрами
   virtual ~Derived();                // виртуальный деструктор
   void Show();                       // переопределенная виртуальная функция
      // другие переопределенные виртуальные функции
      // другие перегруженные операции
   protected:
      // используется вместо private, если ожидается наследование
   private:
      // используется для деталей реализации
};
```

Объект абстрактного класса не может быть формальным параметром функции, однако формальным параметром может быть указатель на абстрактный класс. В этом случае появляется возможность передавать в вызываемую функцию в качестве фактического параметра значение указателя на производный объект, заменяя им указатель на абстрактный базовый класс. Таким образом мы получаем **полиморфные объекты**.

Абстрактный метод может рассматриваться как обобщение *переопределения.* В обоих случаях поведение родительского класса изменяется для потомка. Для абстрактного метода, однако, поведение просто не определено. Любое поведение задается в производном классе.

Одно из преимуществ абстрактного метода является чисто концептуальным: программист может мысленно наделить нужным действием абстракцию сколь угодно высокого уровня. Например, для геометрических фигур мы можем определить метод *Draw*, который их рисует: треугольник *TTriangle*, окружность *TCircle*, квадрат *TSquare.* Мы определим аналогичный метод и для абстрактного родительского класса *TGraphObject*. Однако такой метод не может выполнять полезную работу, поскольку в классе

TGraphObject просто нет достаточной информации для рисования чего бы то ни было. Тем не менее присутствие метода *Draw* позволяет связать функциональность (рисование) только один раз с классом *TGraphObject,* а не вводить три независимые концепции для подклассов *TTriangle*, *TCircle*, *TSquare.*

Имеется и вторая, более актуальная причина использования абстрактного метода. В объектно-ориентированных языках программирования со статическими типами данных, к которым относится и C++, программист может вызвать метод класса, только если компилятор может определить, что класс действительно имеет такой метод. Предположим, что программист хочет определить полиморфную переменную типа *TGraphObject,* которая будет в различные моменты времени содержать фигуры различного типа. Это допустимо для полиморфных объектов. Тем не менее компилятор разрешит использовать метод *Draw* для переменной, только если он сможет гарантировать, что в классе переменной имеется этот метод. Присоединение метода *Draw* к классу *TGraphObject* обеспечивает такую гарантию, даже если метод *Draw* для класса *TGraphObject* никогда не выполняется. Естественно, для того чтобы каждая фигура рисовалась по-своему, метод *Draw* должен быть *виртуальным.*

1.4 Порядок выполнения работы.

1. Определить иерархию классов (в соответствии с вариантом).
2. Определить в классе статическую компоненту - указатель на начало связанного списка объектов и статическую функцию для просмотра списка.
3. Реализовать классы.
4. Написать демонстрационную программу, в которой создаются объекты различных классов и помещаются в список, после чего список просматривается.
5. Сделать соответствующие методы не виртуальными и посмотреть, что будет.
6. Реализовать вариант, когда объект добавляется в список при создании, т.е. в конструкторе (смотри пункт 6 следующего раздела).

1.5 Методические указания.

1. Для определения иерархии классов связать отношением наследования классы, приведенные в приложении (для заданного варианта). Из перечисленных классов выбрать один, который будет стоять во главе иерархии. Это абстрактный класс.
2. Определить в классах все необходимые конструкторы и деструктор.
3. Компонентные данные класса специфицировать как **protected**.
4. Пример определения статических компонентов:
    a. static person* begin; // указатель на начало списка
    b. static void print(void); // просмотр списка

5. Статическую компоненту-данное инициализировать вне определения класса, в глобальной области.
6. Для добавления объекта в список предусмотреть метод класса, т.е. объект сам добавляет себя в список. Например, a.Add() – объект **a** добавляет себя в список.
    a. Включение объекта в список можно выполнять при создании объекта, т.е. поместить операторы включения в конструктор. В случае иерархии классов, включение объекта в список должен выполнять **только** конструктор базового класса. Вы должны продемонстрировать оба этих способа.
7. Список просматривать путем вызова виртуального метода **Show** каждого объекта.
8. Статический метод просмотра списка вызывать не через объект, а через класс.
9. Определение классов, их реализацию, демонстрационную программу поместить в отдельные файлы.

1.6 Содержание отчета.

1. Титульный лист: название дисциплины; номер и наименование работы; фамилия, имя, отчество студента; дата выполнения.
2. Постановка задачи. Следует дать конкретную постановку, т.е. указать, какие классы должны быть реализованы, какие должны быть в них конструкторы, компоненты-функции и т.д.
3. Иерархия классов в виде графа.
4. Определение пользовательских классов с комментариями.
5. .Реализация конструкторов с параметрами и деструктора.
6. Реализация методов для добавления объектов в список.
7. Реализация методов для просмотра списка.
8. Листинг демонстрационной программы.
9. Объяснение необходимости виртуальных функций. Следует показать, какие результаты будут в случае виртуальных и не виртуальных функций.

1.7 Варианты заданий.

Перечень классов:

1. студент, преподаватель, персона, завкафедрой;
2. служащий, персона, рабочий, инженер;
3. рабочий, кадры, инженер, администрация;
4. деталь, механизм, изделие, узел;
5. организация, страховая компания, судостроительная компания, завод;
6. журнал, книга, печатное издание, учебник;
7. тест, экзамен, выпускной экзамен, испытание;
8. место, область, город, мегаполис;
9. игрушка, продукт, товар, молочный продукт;
10. квитанция, накладная, документ, чек;
11. автомобиль, поезд, транспортное средство, экспресс;

12. двигатель, двигатель внутреннего сгорания, дизель, турбореактивный двигатель;
13. республика, монархия, королевство, государство;
14. млекопитающие, парнокопытные, птицы, животное;
15) корабль, пароход, парусник, корвет.

2 Иерархия объектов и группа. итераторы

2.1 Цель.

Получить практические навыки создания объектов-групп и использования методов-итераторов.

2.2 Основные теоретические сведения.

Группа.

Группа – это объект, в который включены другие объекты. Объекты, входящие в группу, называются *элементами группы*. Элементы группы, в свою очередь, могут быть группой.

Примеры групп:

1. Окно в интерактивной программе, которое владеет такими элементами, как поля ввода и редактирования данных, кнопки, списки выбора, диалоговые окна и т.д.

2. Агрегат, состоящий из более мелких узлов.

3. Огород, состоящий из растений, системы полива и плана выращивания.

4. Некая организационная структура (например, ФАКУЛЬТЕТ, КАФЕДРА, СТУДЕНЧЕСКАЯ ГРУППА).

Мы отличаем “группу” от “контейнера”. Контейнер используется для хранения других данных. Примеры контейнеров: объекты контейнерных классов библиотеки **STL** в **C++** (массивы, списки, очереди).

В отличие от контейнера мы понимаем группу как класс, который не только хранит объекты других классов, но и обладает собственными свойствами, не вытекающими из свойств его элементов.

Группа дает второй вид иерархии (первый вид – *иерархия классов*, построенная на основе наследования) – *иерархию объектов* (иерархию типа *целое/часть*), построенную на основе агрегации.

Реализовать группу можно несколькими способами:

1. Класс “группа” содержит поля данных объектного типа. Таким образом, объект “группа” в качестве данных содержит либо непосредственно свои элементы, либо указатели на них

```
class TWindowDialog: public TGroup
    {
     protected:
    TInputLine input1;
    TEdit   edit1;
    TButton button1;
    /*другие члены класса*/
    };
```

Такой способ реализации группы используется в **C++Builder.**

2. Группа содержит член-данное *last* типа *TObject**, который указывает на начало связанного списка объектов, включенных в группу. В этом случае объекты должны иметь поле *next* типа *TObject*,* указывающее на следующий элемент в списке.

3. Создается связанный список структур типа *TItem:*

```
struct TItem
{TObject* item;
 TItem* next;};
```

Поле *item* указывает на объект, включенный в группу. Группа содержит поле *last* типа *TItem* *, которое указывает на начало связанного списка структур типа *TItem*.

Если необходим доступ элементов группы к ее полям и методам, объект типа *TObject* должен иметь поле *owner* типа *TGroup**, которое указывает на собственника этого элемента.

Методы группы.

Имеется два метода, которые необходимы для функционирования группы:

1) *void Insert(TObject* p);*

Вставляет элемент в группу.

2) *void Show();*

Позволяет просмотреть группу.

Кроме этого группа может содержать следующие методы:

1) *int Empty();*

Показывает, есть ли хотя бы один элемент в группе.

2) *TObject* Delete(TObject* p);*

Удаляет элемент из группы, но сохраняет его в памяти.

3) *void DelDisp(TObject* p);*

Удаляет элемент из группы и из памяти.

Иерархия объектов.

Иерархия классов есть иерархия по принципу наследования, т.е. типа “это есть разновидность того”. Например, “рабочий есть разновидность персоны”, “автомобиль” есть разновидность “транспортного средства”. В отличие от этого иерархия объектов – это иерархия по принципу вхождения, т.е. типа “это есть часть того”. Например, “установка – часть завода”, “двигатель” – часть “автомобиля”. Таким образом, объекты нижнего уровня иерархии включаются в объекты более высокого уровня, которые являются для них группой.

Итератор.

Итераторы позволяют выполнять некоторые действия для каждого элемента определенного набора данных.

***For** all элементов набора { действия}*

Такой цикл мог бы быть выполнен для всего набора, например, чтобы напечатать все элементы набора, или мог бы искать некоторый элемент, который удовлетворяет определенному условию, и в этом случае такой цикл может закончиться, как только будет найден требуемый элемент.

Мы будем рассматривать итераторы как специальные методы класса-группы, позволяющие выполнять некоторые действия для всех объектов, включенных в группу. Примером итератора является метод *Show.*

Нам бы хотелось иметь такой итератор, который позволял бы выполнять над всеми элементами группы действия, заданные не одним из методов объекта, а произвольной функцией пользователя. Такой итератор можно реализовать, если эту функцию передавать ему через указатель на функцию.

Определим тип указателя на функцию следующим образом:

*typedef void(*PF)(TObject*,< дополнительные параметры>);*

Функция имеет обязательный параметр типа *TObject* или *TObject**, через который ей передается объект, для которого необходимо выполнить определенные действия.

Метод-итератор объявляется следующим образом:

void TGroup::ForEach(PF action,< дополнительные параметры >);

где

action – единственный обязательный параметр-указатель на функцию, которая вызывается для каждого элемента группы;

дополнительные параметры – передаваемые вызываемой функции параметры.

Затем определяется указатель на функцию и инициализируется передаваемой итератору функцией.

PF pf=myfunc;

Тогда итератор будет вызываться, например, для дополнительного параметра типа int, так:

gr.ForEach(pf,25);

Здесь *gr* – объект-группа.

Динамическая идентификация типов.

Динамическая идентификация типа характерна для языков, в которых поддерживается полиморфизм. В этих языках возможны ситуации, в которых тип объекта на этапе компиляции неизвестен.

В C++ полиморфизм поддерживается через иерархии классов, виртуальные функции и указатели базовых классов. При этом указатель базового класса может использоваться либо для указания на объект базового класса, либо для указания на объект любого класса, производного от этого базового.

Пусть группа содержит объекты различных классов и необходимо выполнить некоторые действия только для объектов определенного класса. Тогда в итераторе мы должны распознавать тип очередного объекта.

В стандарт языка C++ включены средства **RTTI** (Run-Time Type Idendification) – динамическая идентификация типов. Информацию о типе объекта получают с помощью оператора typeid, определение которого содержит заголовочный файл <typeinfo.h>.

Имеется две формы оператора typeid:

typeid (объект)

typeid (имя_типа)

Оператор typeid возвращает ссылку на объект типа type_info.

В классе type_info перегруженные операции == и != обеспечивают сравнение типов.

Функция name () возвращает указатель на имя типа.

Имеется одно ограничение. Оператор typeid работает корректно только с объектами, у которых определены виртуальные функции. Большинство объектов имеют виртуальные функции, хотя бы потому, что обычно деструктор является виртуальным для устранения потенциальных проблем с производными классами. Когда оператор typeid применяют к неполиморфному классу (в классе нет виртуальной функции), получают указатель или ссылку базового типа.

Примеры.

1.

```
#include<iostream>
#include<typeinfo>
using namespace std;
class Base
{
   virtual void f(){};
   //…
};
class Derived: public Base
{
   //…
};
void main()
{
   int i;
   Base ob,*p;
   Derived ob1;
   cout<<typeid(i).name();    //Выводится int
   p=&ob1;
   cout<<typeid(*p).name();  // Выводится Derived
}
```

2.

```
//начало см. выше
```

```
void WhatType(Base& ob)
{
   cout<< typeid(ob).name()<<endl;
}
void main()
{
   Base ob;
   Derived ob1;
   WhatType(ob);    //Выводится Base
   WhatType(ob1);  //Выводится Derived
}
```

3.

```
//начало см. выше
void main()
{
   Base *p;
   Derived ob;
   p=&ob;
   if(typeid(*p)==typeid(Derived)) cout<<“p указывает на объект типа Derived”;
}
```

Если при обращении typeid(*p), p=NULL, то возбуждается исключительная ситуация bad_typeid

2.3 Порядок выполнения работы.

1. Дополнить иерархию классов лабораторной работы № 1 классами “группа”.

Например, для предметной области ФАКУЛЬТЕТ можно предложить классы “*факультет*”, “*студенческая группа*”, “*кафедра*”. Рекомендуется создать абстрактный класс – “*подразделение*”, который будет предком всех групп и абстрактный класс *TObject*, находящийся во главе всей иерархии.

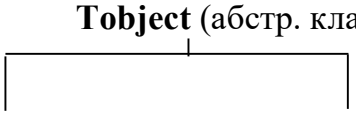
2. Написать для класса-группы метод-итератор.

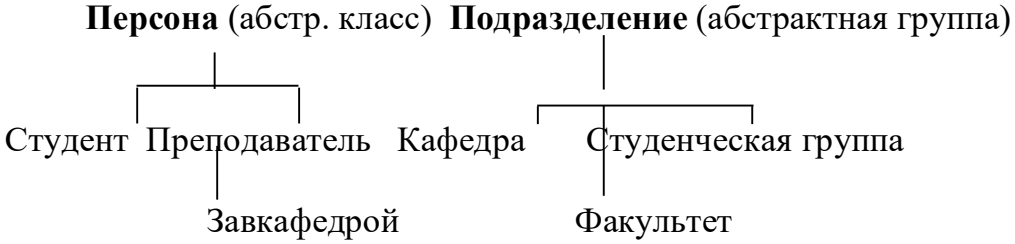
3. Написать процедуру или функцию, которая выполняется для всех объектов, входящих в группу (смотри примеры в приложении).

4. Написать демонстрационную программу, в которой создаются, показываются и разрушаются объекты-группы, а также демонстрируется использование итератора.

2.4 Методические указания.

1. Класс-группа должен соответствовать иерархии классов лабораторной работы № 1, т.е. объекты этих классов должны входить в группу. Например, для варианта 1 может быть предложена следующая иерархия классов:





При этом иерархия объектов будет иметь следующий вид:

2. Для включения объектов в группу следует использовать третий способ (через связанный список структур типа TItem).

3. Пример определения добавленных абстрактных классов:

```
class TObject
{
  public:
    virtual void Show()=0;
};

class TDepartment:public TObject   // абстрактный класс-группа
{
  protected:
     char name[20];  // наименование
     TPerson* head; // руководитель
     TItem* last; // указатель на начало св. списка структур TItem
  public:
     TDepartment(char*,TPerson*);
     TDepardment(TDepartment&);
   ~ TDepartment();
  char* GetName();
  TPerson* GetHead();
  void SetName(char* NAME);
  void  SetHead(TPerson* p);
```

```
    void Insert(TObject* p);
    virtual void Show()=0;
};
```

4. Иерархия объектов создается следующим образом (на примере ФАКУЛЬТЕТА):

а) создается пустой ФАКУЛЬТЕТ,

б) создается пустая КАФЕДРА,

в) создаются ПРЕПОДАВАТЕЛИ и включаются в КАФЕДРУ,

г) КАФЕДРА включается в ФАКУЛЬТЕТ,

д) тоже повторяется для другой кафедры,

е) создается пустая СТУДЕНЧЕСКАЯ ГРУППА,

ж) создаются СТУДЕНТЫ и включаются в СТУДЕНЧЕСКУЮ ГРУППУ,

з) СТУДЕНЧЕСКАЯ ГРУППА включается в ФАКУЛЬТЕТ,

и) тоже повторяется для другой студенческой группы.

5. Удаляется ФАКУЛЬТЕТ (при вызове деструктора) в обратном порядке.

6. Метод-итератор определяется в неабстрактных классах-группах на основе выбранных запросов.

Например, для класса *TStudentGroup* может быть предложен итератор *void TStudentGroup::ForEach(PF action, float parametr);*

где *action* – указатель на функцию, которая должна быть выполнена для всех объектов, включенных в группу (в данном случае для всех СТУДЕНТОВ), *parametr*-передаваемая процедуре дополнительная информация.

В качестве передаваемой методу функции может быть предложена, например, такая: вывести список студентов, имеющих рейтинг не ниже заданного

```
void MyProc(TObject* p,float rate)
{
  if (((TStudent*)p) ->GetGrade()>=rate) cout<<(((TStudent*)p) ->GetName());
}
```

7. Студент определяет передаваемую итератору функцию на основе запросов, которые должны быть выполнены вызовом итератора. Варианты запросов приведены в приложении.

2.5 Содержание отчета.

1. Титульный лист.
2. Постановка задачи.
3. Иерархия классов.
4. Иерархия объектов.
5. Определение классов (добавленных или измененных по сравнению с лабораторной работой № 1).
6. Реализация для одного не абстрактного класса-группы всех методов.

7. Реализация итератора.

8. Реализация передаваемой итератору функции.

9. Листинг демонстрационной программы.

2.6 Варианты задания.

1. Имена всех лиц мужского (женского) пола.

2. Имена студентов указанного курса.

3. Имена и должность преподавателей указанной кафедры.

4. Имена служащих со стажем не менее заданного.

5. Имена служащих заданной профессии.

6. Имена рабочих заданного цеха.

7. Имена рабочих заданной профессии.

8. Имена студентов, сдавших все (заданный) экзамены на отлично (хорошо и отлично).

9. Имена студентов, не сдавших все (хотя бы один) экзамен.

10. Имена всех монархов на заданном континенте.

11. Наименование всех деталей (узлов), входящих в заданный узел (механизм).

12. Наименование всех книг в библиотеке (магазине), вышедших не ранее указанного года.

13. Названия всех городов заданной области.

14. Наименование всех товаров в заданном отделе магазина.

15. Количество мужчин (женщин).

16. Количество студентов на указанном курсе.

17. Количество служащих со стажем не менее заданного.

18. Количество рабочих заданной профессии.

19. Количество инженеров в заданном подразделении.

20. Количество товара заданного наименования.

21. Количество студентов, сдавших все экзамены на отлично.

22. Количество студентов, не сдавших хотя бы один экзамен.

23. Количество деталей (узлов), входящих в заданный узел (механизм).

24. Количество указанного транспортного средства в автопарке (на автостоянке).

25. Количество пассажиров во всех вагонах экспресса.

26. Суммарная стоимость товара заданного наименования.

27. Средний балл за сессию заданного студента.

28. Средний балл по предмету для всех студентов.

29. Суммарное количество учебников в библиотеке (магазине).

30. Суммарное количество жителей всех городов в области.

31. Суммарная стоимость продукции заданного наименования по всем накладным.

32. Средняя мощность всех (заданного типа) транспортных средств в организации.

33. Средняя мощность всех дизелей, обслуживаемых заданной фирмой.

34. Средний вес животных заданного вида в зоопарке.

35. Среднее водоизмещение всех парусников на верфи (в порту).

3 Перегрузка операций

3.1 Цель.

Получить практические навыки работы в среде BC++5.02 и создания EasyWin-программы. Получить практические навыки создания абстрактных типов данных и перегрузки операций в языке C++.

3.2 Основное содержание работы.

Определить и реализовать класс – абстрактный тип данных. Определить и реализовать операции над данными этого класса. Написать и выполнить EasyWin-программу полного тестирования этого класса.

3.3 Краткие теоретические сведения.

Абстрактный тип данных (АТД).

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 C++.

Перегрузка операций.

Возможность использовать знаки стандартных операций для записи выражений как для встроенных, так и для АТД.

В языке C++ для перегрузки операций используется ключевое слово **operator**, с помощью которого определяется специальная операция-функция (operator function).

Формат операции-функции:

тип_возвр_значения operator знак_операции (специф_параметров)
{операторы_тела_функции}

Перегрузка унарных операций

Любая унарная операция ⊕ может быть определена двумя способами: либо как компонентная функция без параметров, либо как глобальная (возможно дружественная) функция с одним параметром. В первом случае выражение ⊕ Z означает вызов Z.operator ⊕ (), во втором – вызов operator ⊕(Z).

Унарные операции, перегружаемые в рамках определенного класса, могут перегружаться только через нестатическую компонентную функцию без параметров. Вызываемый объект класса автоматически воспринимается как операнд.

Унарные операции, перегружаемые вне области класса (как глобальные функции), должны иметь один параметр типа класса. Передаваемый через этот параметр объект воспринимается как операнд.

Синтаксис:

а) в первом случае (описание в области класса):

тип_возвр_значения operator знак_операции

б) во втором случае (описание вне области класса):

тип_возвр_значения operator знак_операции(идентификатор_типа)

Примеры.

```
1) class person
{ int age;
  ...
public:
  …
  void operator++(){ ++age;}
  };
void main()
{class person jon;
 ++jon;}
```

```
2) class person
 { int age;
   ...
   public:
   ...
  friend void operator++(person&);
   };
void person::operator++(person& ob)
   {++ob.age;}
   void main()
   {class person jon;
   ++jon;}
```

Перегрузка бинарных операций

- Любая бинарная операция ⊕ может быть определена двумя способами: либо как компонентная функция с одним параметром, либо как глобальная (возможно дружественная) функция с двумя параметрами. В первом случае **x⊕y** означает вызов **x.operator⊕(y)**, во втором – вызов **operator ⊕(x,y).**

- Операции, перегружаемые внутри класса, могут перегружаться только нестатическими компонентными функциями с параметрами.

Вызываемый объект класса автоматически воспринимается в качестве первого операнда.

- Операции, перегружаемые вне области класса, должны иметь два операнда, один из которых должен иметь тип класса.

Примеры.

```
1) class person{…};
class adresbook
{ // содержит в качестве компонентных данных множество объектов типа //person, представляемых как динамический массив, список или дерево
  …
public:
  person&  operator[](int);  //доступ к i-му объекту
};
person& adresbook : : operator[](int i){. . .}
void main()
{class adresbook persons;
 class person record;
 …
 record = persons [3];
}

2) class person{…};
class adresbook
{ // содержит в качестве компонентных данных множество объектов типа//person, представляемых как динамический массив, список или дерево
  …
public:
 friend person& operator[](const adresbook&,int);//доступ к i-му объекту
};
 person&  operator[](const adresbook& ob ,int i){. . .}
void main()
{class adresbook persons;
 class person record;
 …
 record = persons [3];
}
```

Перегрузка операции присваивания

Операция отличается тремя особенностями:

- операция не наследуется;

- операция определена по умолчанию для каждого класса в качестве операции поразрядного копирования объекта, стоящего справа от знака операции, в объект, стоящий слева.
- операция может перегружаться только в области определения класса. Это гарантирует, что первым операндом всегда будет леводопустимое выражение.

Формат перегруженной операции присваивания:

имя_класса& *operator*=(**имя_класса**&);

Отметим две важные особенности функции *operanor*=. Во-первых, в ней используется параметр-ссылка. Это необходимо для предотвращения создания копии объекта, передаваемого через параметр по значению. В случаи создания копии, она удаляется вызовом деструктора при завершении работы функции. Но деструктор освобождает распределенную память, еще необходимую объекту, который является аргументом. Параметр-ссылка помогает решить эту проблему.

Во-вторых, функция operator=() возвращает не объект, а ссылку на него. Смысл этого тот же, что и при использовании параметра-ссылки. Функция возвращает временный объект, который удаляется после завершения ее работы. Это означает, что для временной переменной будет вызван деструктор, который освобождает распределенную память. Но она необходима для присваивания значения объекту. Поэтому, чтобы избежать создания временного объекта, в качестве возвращаемого значения используется ссылка.

3.4 Порядок выполнения работы.

1. Выбрать класс АТД в соответствии с вариантом.
2. Определить и реализовать в классе конструкторы, деструктор, функции Input (ввод с клавиатуры) и Print (вывод на экран), перегрузить операцию присваивания.
3. Написать программу тестирования класса и выполнить тестирование.
4. Дополнить определение класса заданными перегруженными операциями (в соответствии с вариантом).
5. Реализовать эти операции. Выполнить тестирование.

3.5 Методические указания.

1.Класс АТД реализовать как динамический массив. Для этого определение класса должно иметь следующие поля:

– указатель на начало массива;

– максимальный размер массива;

– текущий размер массива.

2. Конструкторы класса размещают массив в памяти и устанавливают его максимальный и текущий размер. Для задания максимального массива использовать константу, определяемую вне класса.

3. Чтобы у вас не возникало проблем, аккуратно работайте с константными объектами. Например:

конструктор копирования следует определить так:

```
MyClass (const MyClass& ob);
```

операцию присваивания перегрузить так:

```
MyClass& operator = (const MyClass& ob);
```

4. Для удобства реализации операций-функций реализовать в классе **private(protected**)-функции, работающие непосредственно с реализацией класса. Например, для класса **множество** это могут быть следующие функции:

– включить элемент в множество;

– найти элемент и возвратить его индекс;

– удалить элемент;

– определить, принадлежит ли элемент множеству.

Указанные функции используются в реализации общедоступных функций-операций (operator).

3.6 Содержание отчета.

1. Титульный лист.
2. Конкретное задание с указанием номера варианта, реализуемого класса и операций.
3. Определение класса.
4. Обоснование включения в класс нескольких конструкторов, деструктора и операции присваивания.
5. Объяснить выбранное представление памяти для объектов реализуемого класса.
6. Реализация перегруженных операций с обоснованием выбранного способа (функция – член класса, внешняя функция, внешняя дружественная функция).
7. Тестовые данные и результаты тестирования.

Вопросы для самоконтроля.

1. Что такое абстрактный тип данных?
2. Приведите примеры абстрактных типов данных.
3. Каковы синтаксис/семантика “операции-функции”?
4. Как можно вызвать операцию-функцию?
5. Нужно ли перегружать операцию присваивания относительно определенного пользователем типа данных, например класса? Почему?
6. Можно ли изменить приоритет перегруженной операции?
7. Можно ли изменить количество операндов перегруженной операции?
8. Можно ли изменить ассоциативность перегруженной операции?

9. Можно ли, используя дружественную функцию, перегрузить оператор присваивания?

10. Все ли операторы языка C++ могут быть перегружены?

11. Какими двумя разными способами определяются перегруженные операции?

12. Все ли операции можно перегрузить с помощью глобальной дружественной функции?

13. В каких случаях операцию можно перегрузить только глобальной функцией?

14. В каких случаях глобальная операция-функция должна быть дружественной?

15. Обязателен ли в функции operator параметр типа “класс” или “ссылка на класс”?

16. Наследуются ли перегруженные операции?

17. Можно ли повторно перегрузить в производном классе операцию, перегруженную в базовом классе?

18. В чем отличие синтаксиса операции-функции унарной и бинарной операции?

19. Приведите примеры перегрузки операций для стандартных типов.

20. Перегрузите операцию “+” для класса “комплексное число”.

21. Перегрузите операции “<”,”>”,”==” для класса “строка символов”.

3.7 Варианты заданий.

1. Множество с элементами типа **char.** Дополнительно перегрузить следующие операции:
   - + – добавить элемент в множество(типа char + set);
   - + – объединение множеств;
   - == – проверка множеств на равенство.
2. Множество с элементами типа **char.** Дополнительно перегрузить следующие операции:
   - - – удалить элемент из множества (типа set-char);
   - пересечение множеств;
   - < – сравнение множеств.
3. Множество с элементами типа **char.** Дополнительно перегрузить следующие операции:
   - - – удалить элемент из множества (типа set-char);
   - проверка на подмножество;
   - != – проверка множеств на неравенство.
4. Множество с элементами типа **char.** Дополнительно перегрузить следующие операции:

- + – добавить элемент в множество (типа set+char);
- пересечение множеств;
- int()– мощность множества.

5. Множество с элементами типа **char.** Дополнительно перегрузить следующие операции:
   - () – конструктор множества (в стиле конструктора Паскаля);
   - + – объединение множеств;
   - <= – сравнение множеств .
6. Множество с элементами типа **char.** Дополнительно перегрузить следующие операции:
   - проверка на принадлежность(char in set Паскаля);
   - пересечение множеств;
   - < – проверка на подмножество.
7. Список с элементами типа **char.** Дополнительно перегрузить следующие операции:
   - + – объединить списки (list+list);
   - -- – удалить элемент из начала (типа --list);
   - == – проверка на равенство.
8. Однонаправленный список с элементами типа **char.** Дополнительно перегрузить следующие операции:
   - + – добавить элемент в начало(char+list);
   - -- – удалить элемент из начала(типа –list);
   - == – проверка на равенство.
9. Однонаправленный список с элементами типа **char.** Дополнительно перегрузить следующие операции:
   - + – добавить элемент в конец (list+char);
   - -- – удалить элемент из конца (типа list--);
   - != – проверка на неравенство.
10. Однонаправленный список с элементами типа **char.** Дополнительно перегрузить следующие операции:
    - [] – доступ к элементу в заданной позиции, например:
    - int i; char c;
    - list L;
    - c=L[i];
    - + – объединить два списка;
    - == – проверка на равенство.
11. Однонаправленный список с элементами типа **char.** Дополнительно перегрузить следующие операции:
    - [] – доступ к элементу в заданной позиции, например:
    - int i; char c;
    - list L;
    - c=L[i];
    - + – объединить два списка;

- != – проверка на неравенство.

12. Однонаправленный список с элементами типа **char.** Дополнительно перегрузить следующие операции:
    - () – удалить элемент в заданной позиции, например :
    - int i;
    - list L;
    - L[i];
    - () – добавить элемент в заданную позицию, например :
    - int i; char c;
    - list L;
    - L[c,i];
    - != – проверка на неравенство.
13. Стек. Дополнительно перегрузить следующие операции:
    - + – добавить элемент в стек;
    - извлечь элемент из стека;
    - bool()– проверка, пустой ли стек.
14. Очередь. Дополнительно перегрузить следующие операции:
    - + – добавить элемент;
    - извлечь элемент;
    - bool() – проверка, пустая ли очередь.
15. Одномерный массив (вектор) вещественных чисел. Дополнительно перегрузить следующие операции:
    - + – сложение векторов (a[i]+b[i] для всех i);
    - [] – доступ по индексу;
    - + – добавить число к вектору (double+vector).
16. Одномерный массив (вектор) вещественных чисел. Дополнительно перегрузить следующие операции:
    - - – вычитание векторов (a[i]-b[i] для всех i);
    - [] – доступ по индексу;
    - - – вычесть из вектора число (vector-double).
17. Одномерный массив (вектор) вещественных чисел. Дополнительно перегрузить следующие операции:
    - умножение векторов (a[i]*b[i] для всех i);
    - [] – доступ по индексу;
    - умножить вектор на число (vector*double).
18. Одномерный массив (вектор) вещественных чисел. Дополнительно перегрузить следующие операции:
    - int() – размер вектора;
    - () – установить новый размер;
    - - – вычесть из вектора число (vector-double);
    - [] – доступ по индексу;
19. Одномерный массив (вектор) вещественных чисел. Дополнительно перегрузить следующие операции:

- = – присвоить всем элементам вектора значение (vector=double);
- [] – доступ по индексу;
- == – проверка на равенство;
- != – проверка на неравенство;

20. Двухмерный массив (матрица) вещественных чисел. Дополнительно перегрузить следующие операции:
    - () – доступ по индексу;
    - умножение матриц;
    - умножение матрицы на число;
    - умножение числа на матрицу.
21. Двухмерный массив (матрица) вещественных чисел. Дополнительно перегрузить следующие операции:
    - () – доступ по индексу;
    - - – разность матриц;
    - - – вычесть из матрицы число;
    - == – проверка матриц на равенство.
22. Двухмерный массив (матрица) вещественных чисел. Дополнительно перегрузить следующие операции:
    - () – доступ по индексу;
    - = – присвоить всем элементам матрицы значение (matr=double);
    - + – сложение матриц;
    - + – сложить матрицу с числом (matr+double).
23. Двухмерный массив (матрица) вещественных чисел. Дополнительно перегрузить следующие операции:
    - () – доступ по индексу;
    - == – проверка матриц на равенство;
    - ++ – транспонировать матрицу.
24. Система плоских геометрических фигур: круг, квадрат. прямоугольник. Предусмотреть методы для создания объектов, перемещения на плоскости, Дополнительно перегрузить следующие операции:
    - ++, -- – изменения размеров
    - +α, -α – вращения на заданный угол.
25. Комплексные чисел с возможностью задания вещественной и мнимой частей как числами типов double, так и целыми числами. Обеспечить выполнение операций сложения, вычитания и умножения комплексных чисел.
26. Цепные списки строк (строки произвольной длины) с операциями:
    - +α – включения в список.
    - -α – удаления из списка элемента с заданным значением данного.
    - / - – удаления всего списка
    - /α - – удаления конца списка. начиная с заданного элемента.
27. Вектора, задаваемых координатами концов в трехмерном пространстве. Обеспечить операции:

- + – сложения
- - – вычитания
- * – вычисления скалярного произведения двух векторов
- |α – длины вектора,
- αcosβ – угла между векторами.

28. Прямоугольники со сторонами, параллельными осям координат. Предусмотреть возможность перемещения прямоугольников на плоскости,
    - ++ – изменение размеров,
    - α+β – построение наименьшего прямоугольника, содержащего два заданных прямоугольника,
    - α-β – прямоугольника, являющегося общей частью (пересечением) двух прямоугольников.
29. Одномерные массивы целых чисел (векторов). Предусмотреть возможность обращения к отдельному элементу массива с контролем выхода за пределы индексов, возможность задания произвольных границ индексов при создании объекта и выполнения операций
    - + – поэлементного сложения
    - - – вычитания массивов с одинаковыми границами индексов
    - *α – умножения
    - /α – деления всех элементов массива на скаляр
    - p() – печати (вывода на экран) элементов массива по индексам
    - p – печати всего массива.
30. Одномерный массив строк фиксированной длины. Предусмотреть возможность обращения к отдельным строкам массива по индексам, контроль выхода за пределы индексов, выполнения операций:
    - + – поэлементного сцепления двух массивов с образованием нового массива
    - ++ – слияния двух массивов с исключением повторяющихся элементов
    - p() – печати (вывода на экран) элементов массива по индексам
    - p – печати всего массива.
31. Многочлены от одной переменной, задаваемых степенью многочлена и массивом коэффициентов. Предусмотреть методы для вычисления значения многочлена для заданного аргумента, операции:
    - + – поэлементного сложения
    - - – вычитания массивов с одинаковыми границами индексов
    - *α – умножения
    - /α – деления всех элементов массива на скаляр
    - p() – печати (вывода на экран) элементов массива по индексам
    - p – печати всего массива.
32. Одномерный массив строк, каждая строка задается длиной и указателем на выделенную для нее память. Предусмотреть возможность обращения к

отдельным строкам массива по индексам, контроль выхода за пределы индексов, выполнения операций:

- + – поэлементного сцепления двух массивов с образованием нового массива
- ++ – слияния двух массивов с исключением повторяющихся элементов
- p(i) – печати (вывода на экран) элементов массива по индексам
- p – печати всего массива.

33. Тип TMatr, обеспечивающий размещение матрицы произвольного размера с операциями:

- ++, -- – изменения числа строк и столбцов
- p(i, j, k, l) – печати (вывода на экран) элементов массива по индексам
- p() – печати всего массива.

4 Шаблоны функций и классов

4.1 Цель.

Получить практические навыки создания шаблонов и использования их в программах C++.

4.2 Основное содержание работы.

Создать шаблон заданного класса и использовать его для данных различных типов.

4.3 Краткие теоретические сведения.

Шаблон функции.

Шаблон функции (иначе параметризированная функция) определяет общий набор операций (алгоритм), которые будут применяться к данным различных типов. При этом тип данных, над которыми функция должна выполнять операции, передается ей в виде параметра на стадии компиляции.

В C++ параметризированная функция создается с помощью ключевого слова **template**. Формат шаблона функции:

template <class тип_данных> тип_возвр_значения имя_функции(список_параметров){тело_функции}

Основные свойства параметров шаблона функции.

Имена параметров шаблона должны быть уникальными во всем определении шаблона.

Список параметров шаблона не может быть пустым.

В списке параметров шаблона может быть несколько параметров, и каждому из них должно предшествовать ключевое слово class.

Имя параметра шаблона имеет все права имени типа в определенной шаблоном функции.

Определенная с помощью шаблона функция может иметь любое количество непараметризованных формальных параметров. Может быть непараметризованно и возвращаемое функцией значение.

В списке параметров прототипа шаблона имена параметров не обязаны совпадать с именами тех же параметров в определении шаблона.

При конкретизации параметризованной функции необходимо, чтобы при вызове функции типы фактических параметров, соответствующие одинаково параметризованным формальным параметрам, были одинаковы.

Шаблон класса.

Шаблон класса (иначе параметризованный класс) используется для построения родового класса. Создавая родовой класс, вы создаете целое семейство родственных классов, которые можно применять к любому типу данных. Таким образом, тип данных, которым оперирует класс, указывается в качестве параметра при создании объекта, принадлежащего к этому классу. Подобно тому, как класс определяет правила построения и формат отдельных объектов, шаблон класса определяет способ построения отдельных классов. В определении класса, входящего в шаблон, имя класса является не именем отдельного класса, а параметризованным именем семейства классов.

Общая форма объявления параметризованного класса:

template <class тип_данных> class имя_класса { . . . };

Основные свойства шаблонов классов.

Компонентные функции параметризованного класса автоматически являются параметризованными. Их не обязательно объявлять как параметризованные с помощью *template.*

Дружественные функции, которые описываются в параметризованном классе, не являются автоматически параметризованными функциями, т.е. по умолчанию такие функции являются дружественными для всех классов, которые организуются по данному шаблону.

Если *friend*-функция содержит в своем описании параметр типа параметризованного класса, то для каждого созданного по данному шаблону класса имеется собственная *friend*-функция.

В рамках параметризованного класса нельзя определить *friend*-шаблоны (дружественные параметризованные классы).

С одной стороны, шаблоны могут быть производными (наследоваться) как от шаблонов, так и от обычных классов, с другой стороны, они могут использоваться в качестве базовых для других шаблонов или классов.

Шаблоны функций, которые являются членами классов, нельзя описывать как *virtual.*

Локальные классы не могут содержать шаблоны в качестве своих элементов.

Компонентные функции параметризованных классов.

Реализация компонентной функции шаблона класса, которая находится вне определения шаблона класса, должна включать дополнительно следующие два элемента:

Определение должно начинаться с ключевого слова *template*, за которым следует такой же *список_параметров_типов* в угловых скобках, какой указан в определении шаблона класса.

За *именем_класса,* предшествующим операции области видимости (**::**), должен следовать *список_имен_параметров* шаблона.

template<список_типов>тип_возвр_значения имя_класса<список_ имен_ параметров> : : имя_функции(список_параметров){ . . . }

Порядок выполнения работы.

1. Создать шаблон заданного класса. Определить конструкторы, деструктор, перегруженную операцию присваивания (“=”) и операции, заданные в варианте задания.
2. Написать программу тестирования, в которой проверяется использование шаблона для стандартных типов данных.
3. Выполнить тестирование.
4. Определить пользовательский класс, который будет использоваться в качестве параметра шаблона. Определить в классе необходимые функции и перегруженные операции.
5. Написать программу тестирования, в которой проверяется использование шаблона для пользовательского типа.
6. Выполнить тестирование.

4.4 Методические указания.

1. Класс АТД реализовать как динамический массив. Для этого определение класса должно иметь следующие поля:

– указатель на начало массива;

– максимальный размер массива;

– текущий размер массива.

2. Для вода и вывода определить в классе функции **input** и **print**.

3. Чтобы у вас не возникало проблем, аккуратно работайте с константными объектами. Например:

конструктор копирования следует определить так:

```
MyTmp (const MyTmp& ob);
```

операцию присваивания перегрузить так:

```
MyTmp& operator = (const MyTmp& ob);
```

4. Для шаблонов множеств, списков, стеков и очередей в качестве стандартных типов использовать символьные, целые и вещественные типы. Для пользовательского типа взять класс из лабораторной работы № 1.

5. Для шаблонов массивов в качестве стандартных типов использо-

вать целые и вещественные типы. Для пользовательского типа взять класс “комплексное число” *complex*.

```
class complex{
int re;          // действительная часть
int im;          // мнимая часть
public;
// необходимые функции и перегруженные операции
};
```

6. Реализацию шаблона следует разместить вместе с определением в заголовочном файле.

7. Программа создается как EasyWin-приложение в Borland C++5.02.

8. Тестирование должно быть выполнено для всех типов данных и для всех операций.

4.5 Содержание отчета.

1. Титульный лист: название дисциплины, номер и наименование работы, фамилия, имя, отчество студента, дата выполнения.

2. Постановка задачи.

Следует дать конкретную постановку, т.е указать шаблон какого класса должен быть создан, какие должны быть в нем конструкторы, компоненты-функции, перегруженные операции и т.д.

То же самое следует указать для пользовательского класса.

3. Определение шаблона класса с комментариями.

4. Определение пользовательского класса с комментариями.

5. Реализация конструкторов, деструктора, операции присваивания и операций, которые заданы в варианте задания.

6. То же самое для пользовательского класса.

7. Результаты тестирования. Следует указать для каких типов и какие операции проверены и какие выявлены ошибки (или не выявлены)

Вопросы для самоконтроля.

1. В чем смысл использования шаблонов?
2. Каковы синтаксис/семантика шаблонов функций?
3. Каковы синтаксис/семантика шаблонов классов?
4. Напишите параметризованную функцию сортировки массива методом обмена.
5. Определите шаблон класса “вектор” – одномерный массив.
6. Что такое параметры шаблона функции?
7. Перечислите основные свойства параметров шаблона функции.
8. Как записывать параметр шаблона?
9. Можно ли перегружать параметризованные функции?
10. Перечислите основные свойства параметризованных классов.
11. Может ли быть пустым список параметров шаблона? Объясните.

12. Как вызвать параметризованную функцию без параметров?

13. Все ли компонентные функции параметризованного класса являются параметризованными?

14. Являются ли дружественные функции, описанные в параметризованном классе, параметризованными?

15. Могут ли шаблоны классов содержать виртуальные компонентные функции?

16. Как определяются компонентные функции параметризованных классов вне определения шаблона класса?

4.6 Варианты заданий.

1.Класс – одномерный массив. Дополнительно перегрузить следующие операции:

* – умножение массивов;

[] – доступ по индексу.

2. Класс – одномерный массив. Дополнительно перегрузить следующие операции:

int() – размер массива;

[] – доступ по индексу.

3. Класс – одномерный массив. Дополнительно перегрузить следующие операции:

[] – доступ по индексу;

== – проверка на равенство;

!= – проверка на неравенство.

4. Класс – множество set**.** Дополнительно перегрузить следующие операции:

+ – добавить элемент в множество (типа set+item);

+ – объединение множеств;

* – пересечение множеств;

5. Класс – множество set**.** Дополнительно перегрузить следующие операции:

+ – добавить элемент в множество (типа item + set);

+ – объединение множеств;

== – проверка множеств на равенство.

6. Класс – множество set**.** Дополнительно перегрузить следующие операции:

- – удалить элемент из множества (типа set-item);

* – пересечение множеств;

< – сравнение множеств.

7. Класс – множество set**.** Дополнительно перегрузить следующие операции:

- – удалить элемент из множества (типа set-item);

> – проверка на подмножество;

!= – проверка множеств на неравенство.

8.Класс – множество set. Дополнительно перегрузить следующие операции:

+ – добавить элемент в множество (типа set+item);

* – пересечение множеств;

int()– мощность множества.

9. Класс – множество set**.** Дополнительно перегрузить следующие операции:

() – конструктор множества (в стиле конструктора для множественного типа в языке Pascal);

+ – объединение множеств;

<= – сравнение множеств.

10. Класс – множество set**.** Дополнительно перегрузить следующие операции:

> – проверка на принадлежность (типа операции **in** множественного типа в языке Pascal);

* – пересечение множеств;

< – проверка на подмножество.

11. Класс – однонаправленный список list**.** Дополнительно перегрузить следующие операции:

+ – добавить элемент в начало (list+item);

-- – удалить элемент из начала (--list);

== – проверка на равенство.

12. Класс – однонаправленный список list**.** Дополнительно перегрузить следующие операции:

+ – добавить элемент в начало (item+list);

-- – удалить элемент из начала (--list);

!= – проверка на неравенство.

13. Класс – однонаправленный список list**.** Дополнительно перегрузить следующие операции:

+ – добавить элемент в конец (list+item);
-- – удалить элемент из конца (типа list--);
!= – проверка на неравенство.

14. Класс – однонаправленный список list**.** Дополнительно перегрузить следующие операции:

[] – доступ к элементу в заданной позиции, например:
Type c;
int i;
list L;
c=L[i];
+ – объединить два списка;
== – проверка на равенство.

15. Класс – однонаправленный список list**.** Дополнительно перегрузить следующие операции:

[] – доступ к элементу в заданной позиции, например:
int i; Type c;
list L;
c=L[i];
+ – объединить два списка;
!= – проверка на неравенство.

16. Класс – однонаправленный список list**.** Дополнительно перегрузить следующие операции:

() – удалить элемент в заданной позиции, например:
int i;
list L;
L(i);
() – добавить элемент в заданную позицию, например:
int i;
Type c;
list L;
L(c,i);
!= – проверка на неравенство.

17. Класс – стек stack**.** Дополнительно перегрузить следующие операции:

+ – добавить элемент в стек;
-- – извлечь элемент из стека;
bool() – проверка, пустой ли стек.

18. Класс – очередь queue. Дополнительно перегрузить следующие операции:

+ – добавить элемент;

-- – извлечь элемент;

bool() – проверка, пустая ли очередь.

19. Класс – одномерный массив. Дополнительно перегрузить следующие операции:

+ – сложение массивов;

[] – доступ по индексу;

+ – сложить элемент с массивом.

20. Класс – одномерный массив. Дополнительно перегрузить следующие операции:

- – вычитание массивов;

[] – доступ по индексу;

- – вычесть из массива элемент.

5 Потоковые классы

5.1 Цель.

Научиться программировать ввод и вывод в C++, используя объекты потоковых классов стандартной библиотеки C++.

5.2 Основное содержание работы.

Создание пользовательского типа данных, создание и сохранение объектов этого типа в файле, чтение их из файла, удаление из файла, корректировка в файле, создание пользовательских манипуляторов.

5.3 Основные теоретические сведения.

Понятие потока.

Потоковые классы представляют объектно-ориентированный вариант функций ANSI-C. Поток данных между источником и приемником при этом обладает следующими свойствами.

– Источник или приемник данных определяется объектом потокового класса.

– Потоки используются для ввода-вывода высокого уровня.

– Общепринятые стандартные С-функции ввода/вывода разработаны как функции потоковых классов, чтобы облегчить переход от С-функций к C++ классам.

– Потоковые классы делятся на три группы (шаблонов классов):

basic_istream, basic_ostream – общие потоковые классы, которые могут быть связаны с любым буферным объектом;

basic_ifstream, basic_iostream – потоковые классы для считывания и записи файлов;

basic_istringstream, basic_ostringstream – потоковые классы для объектов-строк.

– Каждый потоковый класс поддерживает буферный объект, который предоставляет память для передаваемых данных, а также важнейшие функции ввода/вывода низкого уровня для их обработки.

– Базовым шаблоном классов basic_ios (для потоковых классов) и basic_streambuf (для буферных классов) передаются по два параметра шаблона:

первый параметр (charT) определяет символьный тип;

второй параметр (traits) – объект типа ios_traits (шаблон класса), в котором заданы тип и функции, специфичные для используемого символьного типа;

для типов char и wchar_t образованы соответствующие объекты типа ios_traits и потоковые классы.

Пример шаблона потокового класса.

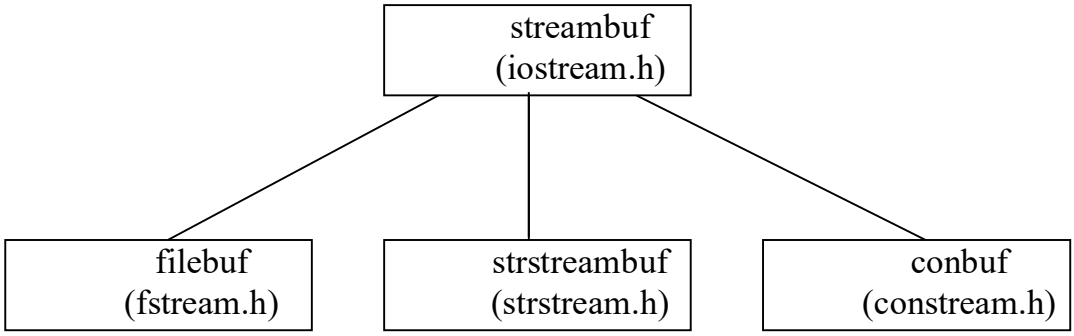
template <class charT, class traits = ios_traits <charT>> class basic_istream: virtual public basic_ios <charT, traits>;

Потоковые классы в C++.

Библиотека потоковых классов C++ построена на основе двух базовых классов: **ios** и **streambuf** .

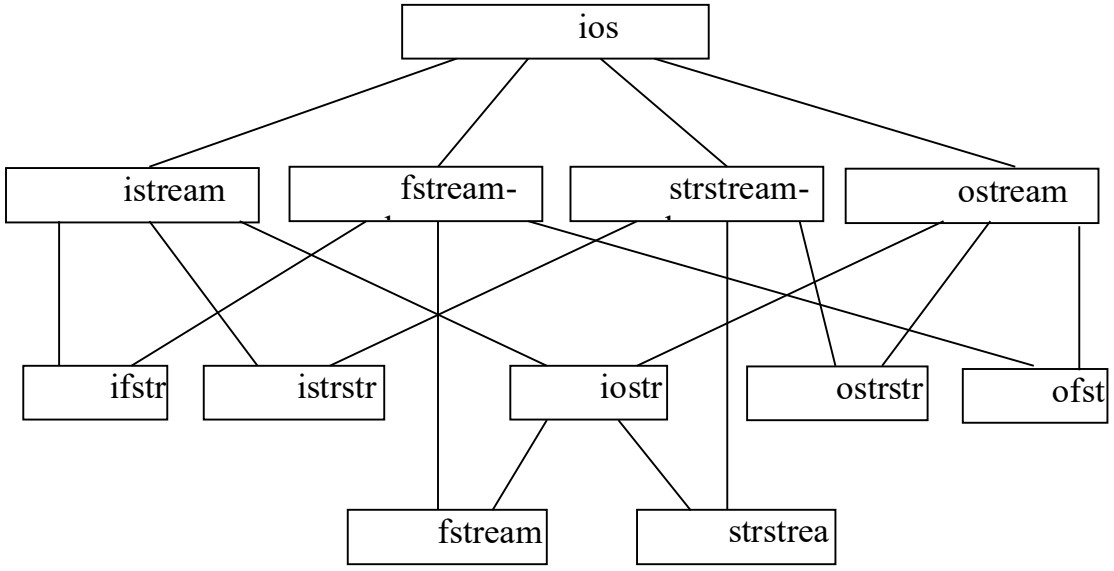
Класс streambuf обеспечивает организацию и взаимосвязь буферов ввода-вывода, размещаемых в памяти, с физическими устройствами ввода-вывода. Методы и данные класса streambuf программист явно обычно не использует. Этот класс нужен другим классам библиотеки ввода-вывода. Он доступен и программисту для создания новых классов на основе уже существующих.

Схема иерархии



Класс ios содержит средства для форматированного ввода-вывода и проверки ошибок.

Схема иерархии



istream – класс входных потоков;
ostream – класс выходных потоков;
iostream – класс ввода-вывода;
istrstream – класс входных строковых потоков;
ifstream – класс входных файловых потоков и т.д.

Потоковые классы, их методы и данные становятся доступными в программе, если в неё включен нужный заголовочный файл.

iostream.h – для ios, ostream, istream.
strstream.h – для strstream, istrstream, ostrstream
fstream.h – для fstream, ifstream, ofstream

Базовые потоки ввода-вывода.

Для ввода с потока используются объекты класса istream, для вывода в поток – объекты класса ostream.

В классе istream определены следующие функции:

istream& get(char* buffer,int size,char delimiter='\n');

Эта функция извлекает символы из istream и копирует их в буфер. Операция прекращается при достижении конца файла, либо при скопировании size символов, либо при обнаружении указанного разделителя.

Сам разделитель не копируется и остается в streambuf. Последовательность прочитанных символов всегда завершается нулевым символом.

istream& read(char* buffer,int size);

Не поддерживает разделителей, и считанные в буфер символы не завершаются нулевым символом.

istream& getline(char* buffer,int size, char delimiter='\n');

Разделитель извлекается из потока, но в буфер не заносится. Это основная функция для извлечения строк из потока. Считанные символы завершаются нулевым символом.

istream& get(streambuf& s,char delimiter='\n');

Копирует данные из istream в streambuf до тех пор, пока не обнаружит конец файла или символ-разделитель, который не извлекается из istream. В s нулевой символ не записывается.

istream get (char& C);

Читает символ из istream в C. В случае ошибки C принимает значение 0XFF.

int get();

Извлекает из istream очередной символ. При обнаружении конца файла возвращает EOF.

int peek();

Возвращает очередной символ из istream, не извлекая его из istream.

int gcount();

Возвращает количество символов, считанных во время последней операции неформатированного ввода.

istream& putback(C)

Если в области get объекта streambuf есть свободное пространство, то туда помещается символ C.

istream& ignore(int count=1,int target=EOF);

Извлекает символ из istream, пока не произойдет следующее:

– функция не извлечет count символов;

– не будет обнаружен символ target;

– не будет достигнуто конца файла.

В классе ostream определены следующие функции:

ostream& put(char C);

Помещает в ostream символ C.

ostream& write(const char* buffer,int size);

Записывает в ostream содержимое буфера. Символы копируются до тех пор, пока не возникнет ошибка или не будет скопировано size символов. Буфер записывается без форматирования. Обработка нулевых символов ничем не отличается от обработки других. Данная функция осуществляет передачу необработанных данных (бинарных или текстовых) в ostream.

ostream& flush();

Сбрасывает буфер streambuf.

Для прямого доступа используются следующие функции установки позиции чтения - записи.

При чтении

istream& seekg(long p);

Устанавливает указатель потока get (не путать с функцией) со смещением p от начала потока.

istream& seekg(long p,seek_dir point);

Указывается начальная точка перемещения.

enum seek_dir{beg,curr,end}

Положительное значение p перемещает указатель get вперед (к концу потока), отрицательное значение p – назад (к началу потока).

long tellg();

Возвращает текущее положение указателя get.

При записи

ostream& seekp(long p);

Перемещает указатель put в streambuf на позицию p от начала буфера streambuf.

ostream& seekp(long p,seek_dir point);

Указывает точка отсчета.

long tellp();

Возвращает текущее положение указателя put.

Помимо этих функций в классе istream перегружена операция >>, а в классе ostream <<. Операции << и >> имеют два операнда Левым операндом является объект класса istream (ostream), а правым – данное, тип которого задан в языке.

Для того чтобы использовать операции << и >> для всех стандартных типов данных используется соответствующее число перегруженных функций operator<< и operator>>. При выполнении операций ввода-вывода в зависимости от типа правого операнда вызывается та или иная перегруженная функция operator.

Поддерживаются следующие типы данных: целые, вещественные, строки (char*). Для вывода – void* (все указатели, отличные от char*, автоматически переводятся к void*). Перегрузка операции >> и << не изменяет их приоритета.

Функции operator<< и operator>>возвращают ссылку на тот потоковый объект, который указан слева от знака операции. Таким образом, можно формировать “цепочки ” операций.

cout << a << b << c;

cin >> i >> j >> k;

При вводе-выводе можно выполнять форматирование данных.

Чтобы использовать операции >> и << с данными пользовательских типов, определяемых пользователем, необходимо расширить действие этих

операций, введя новые операции-функции. Первым параметром операции-функции должна быть ссылка на объект потокового типа, вторым – ссылка или объект пользовательского типа.

В файле iostream.h определены следующие объекты, связанные со стандартными потоками ввода-вывода:

cin – объект класса istream, связанный со стандартным буферизированным входным потоком;

cout – объект класса ostream, связанный со стандартным буферизированным выходным потоком;

cerr – не буферизированный выходной поток для сообщения об ошибках;

clog – буферизированный выходной поток для сообщения об ошибках.

Форматирование.

Непосредственное применение операций ввода << и вывода >> к стандартным потокам cout, cin, cerr, clog для данных базовых типов приводит к использованию “умалчиваемых” форматов внешнего представления пересылаемых значений.

Форматы представления выводимой информации и правила восприятия данных при вводе могут быть изменены программистом с помощью флагов форматирования. Эти флаги унаследованы всеми потоками из базового класса ios. Флаги форматирования реализованы в виде отдельных фиксированных битов и хранятся в protected компоненте класса long x_flags. Для доступа к ним имеются соответствующие public функции.

Кроме флагов форматирования используются следующие protected компонентные данные класса ios:

int x_width – минимальная ширина поля вывода.

int x_precision – точность представления вещественных чисел (количество цифр дробной части) при выводе;

int x_fill – символ-заполнитель при выводе, пробел – по умолчанию.

Для получения (установки) значений этих полей используются следующие компонентные функции:

```
int width();
int width(int);
int precision();
int precision(int);
char fill();
char fill(char);
```

Манипуляторы.

Несмотря на гибкость и большие возможности управления форматами с помощью компонентных функций класса ios, их применение достаточно громоздко. Более простой способ изменения параметров и флагов форматирования обеспечивают манипуляторы.

Манипуляторами называются специальные функции, позволяющие модифицировать работу потока. Особенность манипуляторов состоит в том, что их можно использовать в качестве правого операнда операции >> или <<. В качестве левого операнда, как обычно, используется поток (ссылка на поток), и именно на этот поток воздействует манипулятор.

Для обеспечения работы с манипуляторами в классах istream и ostream имеются следующие перегруженные функции operator.

istream& operator>>(istream&(*_f)(istream&));

ostream& operator<<(ostream&(*_f)(ostream&));

При использовании манипуляторов следует включить заголовочный файл <iomanip.h>, в котором определены встроенные манипуляторы.

Определение пользовательских манипуляторов.

Порядок создания пользовательского манипулятора с параметрами, например для вывода, следующий:

1. Определить класс (my_manip) с полями: параметры манипулятора, указатель на функцию типа

ostream& (*f)(ostream&,<параметры манипулятора>);

2. Определить конструктор этого класса (my_manip) с инициализацией полей.

3. Определить, в этом классе дружественную функцию – operator<<. Эта функция в качестве правого аргумента принимает объект класса my_manip, левого аргумента (операнда) поток ostream и возвращает поток ostream как результат выполнения функции *f. Например,

```
typedef far ostream&(far *PTF)(ostream&,int,int,char);
class my_man{
int w;int n;char fill;
PTF f;
public:
//конструктор
my_man(PTF F,int W,int N,char FILL):f(F),w(W),n(N),fill(FILL){}
friend ostream& operator<<(ostream&,my_man);
};
ostream& operator<<(ostream& out,my_man my)
{return my.f(out,my.w,my.n,my.fill);}
```

4. Определить функцию типа *f (fmanip), принимающую поток и параметры манипулятора и возвращающую поток. Эта функция собственно и выполняет форматирование. Например,

```
ostream& fmanip(ostream& s,int w,int n,char fill)
{s.width(w);
s.flags(ios::fixed);
s.precision(n);
s.fill(fill);
return s;}
```

5. Определить собственно манипулятор (wp) как функцию, принимающую параметры манипулятора и возвращающую объект my_manip, поле f которого содержит указатель на функцию fmanip. Например,

```
my_man wp(int w,int n,char fill)
{return my_man(fmanip,w,n,fill);}
```

Для создания пользовательских манипуляторов с параметрами можно использовать макросы, которые содержатся в файле <iomanip.h>:

```
OMANIP(int)
IMANIP(int)
IOMANIP(int)
```

Состояние потока.

Каждый поток имеет связанное с ним состояние. Состояния потока описываются в классе ios в виде перечисления enum.

```
public:
enum io_state{
goodbit, //нет ошибки 0X00
eofbit,   //конец файла 0X01
failbit,   //последняя операция не выполнилась 0X02
badbit,   //попытка использования недопустимой операции 0X04
hardfail  //фатальная ошибка 0X08
};
```

Флаги, определяющие результат последней операции с объектом ios, содержатся в переменной state. Получить значение этой переменной можно с помощью функции int rdstate().

Кроме того, проверить состояние потока можно следующими функциями:

```
int bad();          1, если badbit или hardfail
int eof();             1, если eofbit
int fail();         1, если failbit, badbit или hardfail
int good();         1, если goodbit
```

Если операция >> используется для новых типов данных, то при её перегрузке необходимо предусмотреть соответствующие проверки.

Файловый ввод-вывод.

Потоки для работы с файлами создаются как объекты следующих классов:

ofstream – запись в файл;

ifstream – чтение из файла;

fstream – чтение/запись.

Для создания потоков имеются следующие конструкторы:

0 fstream();

создает поток, не присоединяя его ни к какому файлу.

0 fstream(const char* name,int mode,int p=filebuf::openprot);

создает поток, присоединяет его к файлу с именем name, предварительно открыв файл, устанавливает для него режим mode и уровень защиты p. Если файл не существует, то он создается. Для mode=ios::out, если файл существует, то его размер будет усечен до нуля.

Флаги режима определены в классе ios и имеют следующие значения:

in – для чтения

out – для записи

ate – индекс потока помещен в конец файла. Чтение больше не допустимо, выводные данные записываются в конец файла;

app – поток открыт для добавления данных в конец. Независимо от seekp() данные будут записываться в конец;

trunc – усечение существующего потока до нуля;

nocreate-команда открытия потока будет завершена неудачно, если файл не существует;

noreplace -команда открытия потока будет завершена неудачно, если файл существует;

binary-поток открывается для двоичного обмена.

Если при создании потока он не присоединен к файлу, то присоединить существующий поток к файлу можно функцией

void open(const char* name,int mode,int p=filebuf::openprot);

Функция

void fstreambase::close();

сбрасывает буфер потока, отсоединяет поток от файла и закрывает файл. Эту функцию необходимо явно вызвать при изменении режима работы с потоком. Автоматически она вызывается только при завершении программы.

Таким образом, создать поток и связать его с файлом можно тремя способами:

1. Создается объект filebuf

filebuf fbuf;

Объект filebuf связывается с устройством (файлом)

fbuf.open(“имя”,ios::in);

Создается поток и связывается с filebuf

istream stream(&fbuf);

2. Создается объект fstream (ifstream, ofstream)

fstream stream;

Открывается файл, который связывается через filebuf с потоком

stream.open(“имя”,ios::in);

3. Создается объект fstream, одновременно открывается файл, который связывается с потоком

fstream stream(“имя”,ios::in);

5.4 Порядок выполнения работы.

1. Определить пользовательский тип данных (класс). Определить и реализовать в нем конструкторы, деструктор, операции присваивания, ввода и вывода для стандартных потоков.

2. Написать программу № 1 для создания объектов пользовательского класса (ввод исходной информации с клавиатуры с использованием перегруженной операции “>>) и сохранения их в потоке (файле). Предусмотреть в программе вывод сообщения о количестве сохраненных объектов и о длине полученного файла в байтах.

3. Выполнить тестирование программы.

4. Реализовать для вывода в поток свой манипулятор с параметрами.

5. Написать программу № 2 для чтения объектов из потока, сохранения их в массиве и просмотра массива. Для просмотра объектов использовать перегруженную для cout операцию << и свой манипулятор. Предусмотреть в программе вывод сообщения о количестве прочитанных объектов и байтов.

6. Выполнить программу для чтения из файла сохраненных предыдущей программой объектов и их просмотра.

7. Написать программу № 3 для добавления объектов в поток.

8. Выполнить программу, добавив в поток несколько объектов и просмотреть полученный файл.

9. Написать программу № 4 для удаления объектов из файла.

10. Выполнить программу, удалив из потока несколько объектов и просмотреть полученный файл.

11. Написать программу № 5 для корректировки (т.е. замены) записей в файле.

12. Выполнить программу и просмотреть полученный файл.

5.5 Методические указания.

1 В качестве пользовательского типа данных взять класс из лабораторной работы № 1. Поля класса типа char* заменить на char[целое].

2 В совокупности программы должны использовать все классы потоков: **istream, ostream, fstream, ifstream, ofstream.**

3 Также в программах следует показать все три способа создания потока и открытия файла (см. выше).

4 Необходимо продемонстрировать чтение из файла и запись в файл как с помощью функций **read/write**, так и с помощью перегруженных операций >> и <<.

5 Пользовательский манипулятор создается с не менее чем с двумя параметрами.

6 Определение пользовательского класса сохранить в h-файле.

7 Определение компонентных функций пользовательского класса сохранить в cpp-файле.

8 Реализацию манипулятора сохранить в h-файле.

В качестве параметров манипулятора можно использовать:

а) ширину поля вывода;

б) точность вывода вещественных чисел;

в) символ-заполнитель;

г) способ выравнивания (к левой или правой границе)

и т.д.

9. В поток записать не менее 5 объектов.

10. После записи объектов в файл и перед чтением их из файла определить количество записанных объектов и вывести эту информацию.

Определить количество записанных в файл объектов можно следующим образом:

а) стать на конец файла – функции seekp(),seekg();

б) определить размер файла в байтах – функции tellp(),tellg();

в) определить количество записанных объектов - размер файла поделить на размер объекта.

11. Необходимо тестировать ошибки при работе с файлом. Для этого следует использовать перегруженные операции operator!(),operator void*() и функции bad(),good().

12 Поскольку в файле может храниться любое, заранее не известное, количество объектов, для их сохранения в программе № 2 при чтении из файла использовать динамический массив.

13. Следует определить функцию **find()**, которая принимает значение ключевого поля объекта и возвращает смещение этого объекта от начала файла. Вызывать эту функцию перед удалением/изменением объекта в файле.

14 Для изменения и удаления объекта написать функции **del()** и **repl()**, которым передается ссылка на поток, смещение от начала файла изменяемой или удаляемой записи (результат вызова функции **find**), новое значение изменяемой записи.

5.6 Содержание отчета.

1. Титульный лист.
2. Постановка задачи.
3. Определение пользовательского класса.
4. Реализация манипулятора.
5. Реализация функций **find(),del()** и **repl().**
6. Пояснения к программам. Для каждой программы указывается, какие потоковые классы в ней используются, как создаются объекты потоковых классов, как открываются файлы, каким образом выполняется ввод и вывод данных.

6 Стандартная библиотека шаблонов

6.1 Цель.

Освоить технологию обобщенного программирования с использованием библиотеки стандартных шаблонов (STL) языка C++.

6.2 Основное содержание работы.

Написать три программы с использованием STL. Первая и вторая программы должны демонстрировать работу с контейнерами STL, третья – использование алгоритмов STL.

6.3 Основные теоретические сведения.

Стандартная библиотека шаблонов (STL).

STL обеспечивает общецелевые, стандартные классы и функции, которые реализуют наиболее популярные и широко используемые алгоритмы и структуры данных.

STL строится на основе шаблонов классов, и поэтому входящие в неё алгоритмы и структуры применимы почти ко всем типам данных.

Состав STL.

Ядро библиотеки образуют три элемента: **контейнеры**, **алгоритмы** и **итераторы**.

Контейнеры (containers) – это объекты, предназначенные для хранения других элементов. Например, вектор, линейный список, множество.

Ассоциативные контейнеры (associative containers) позволяют с помощью ключей получить быстрый доступ к хранящимся в них значениям.

В каждом классе-контейнере определен набор функций для работы с ними. Например, список содержит функции для вставки, удаления и слияния элементов.

Алгоритмы (algorithms) выполняют операции над содержимым контейнера. Существуют алгоритмы для инициализации, сортировки, поиска, замены содержимого контейнеров. Многие алгоритмы предназначены для работы с последовательностью (sequence), которая представляет собой линейный список элементов внутри контейнера.

Итераторы (iterators) – это объекты, которые по отношению к контейнеру играют роль указателей. Они позволяют получить доступ к содержимому контейнера примерно так же, как указатели используются для доступа к элементам массива.

С итераторами можно работать так же, как с указателями. К ним можно применить операции *, инкремента, декремента. Типом итератора объявляется тип iterator, который определен в различных контейнерах.

Существует пять типов итераторов:

1. **Итераторы ввода** (input_iterator) поддерживают операции равенства, разыменования и инкремента.

==, !=, *i, ++i, i++, *i++

Специальным случаем итератора ввода является istream_iterator.

2. Итераторы вывода (output_iterator) поддерживают операции разыменования, допустимые только с левой стороны присваивания, и инкремента.

++i, i++, *i=t, *i++=t

Специальным случаем итератора вывода является ostream_iterator.

3. Однонаправленные итераторы (forward_iterator) поддерживают все операции итераторов ввода/вывода и, кроме того, позволяют без ограничения применять присваивание.

==, !=, =, *i, ++i, i++, *i++

4. Двунаправленные итераторы (biderectional_iterator) обладают всеми свойствами forward-итераторов, а также имеют дополнительную операцию декремента (--i, i--, *i--), что позволяет им проходить контейнер в обоих направлениях.

5. Итераторы произвольного доступа (random_access_iterator) обладают всеми свойствами biderectional-итераторов, а также поддерживают операции сравнения и адресной арифметики, то есть непосредственный доступ по индексу.

i+=n, i+n, i-=n, i-n, i1-i2, i[n], i1<i2, i1<=i2, i1>i2, i1>=i2

В STL также поддерживаются **обратные итераторы** (reverse iterators). Обратными итераторами могут быть либо двунаправленные итераторы, либо итераторы произвольного доступа, но проходящие последовательность в обратном направлении.

Вдобавок к контейнерам, алгоритмам и итераторам в STL поддерживается ещё несколько стандартных компонентов. Главными среди них являются **распределители памяти**, **предикаты** и **функции сравнения**.

У каждого контейнера имеется определенный для него распределитель памяти (**allocator**), который управляет процессом выделения памяти для контейнера.

По умолчанию распределителем памяти является объект класса **allocator**. Можно определить собственный распределитель.

В некоторых алгоритмах и контейнерах используется функция особого типа, называемая **предикатом**. Предикат может быть унарным и бинарным. Возвращаемое значение: истина либо ложь. Точные условия получения того или иного значения определяются программистом. Тип унарных предикатов **UnPred**, бинарных – **BinPred**. Тип аргументов соответствует типу хранящихся в контейнере объектов.

Определен специальный тип бинарного предиката для сравнения двух элементов. Он называется **функцией сравнения** (comparison function). Функция возвращает истину, если первый элемент меньше второго. Типом функции является тип **Comp**.

Особую роль в STL играют объекты-функции.

Объекты-функции – это экземпляры класса, в котором определена операция «круглые скобки» (). В ряде случаев удобно заменить функцию на

объект-функцию. Когда объект-функция используется в качестве функции, то для ее вызова используется operator ().

Пример 1.

```
class less{
public:
bool operator()(int x,int y)
{return x<y;}
};
```

3. Классы-контейнеры.

В STL определены два типа контейнеров: последовательности и ассоциативные.

Ключевая идея для стандартных контейнеров заключается в том, что когда это представляется разумным, они должны быть логически взаимозаменяемыми. Пользователь может выбирать между ними, основываясь на соображениях эффективности и потребности в специализированных операциях. Например, если часто требуется поиск по ключу, можно воспользоваться **map** (ассоциативным массивом). С другой стороны, если преобладают операции, характерные для списков, можно воспользоваться контейнером **list**. Если добавление и удаление элементов часто производится в концы контейнера, следует подумать об использовании очереди **queue**, очереди с двумя концами **deque**, стека **stack**. По умолчанию пользователь должен использовать **vector**; он реализован, чтобы хорошо работать для самого широкого диапазона задач.

Идея обращения с различными видами контейнеров и, в общем случае, со всеми видами источников информации – унифицированным способом ведет к понятию **обобщенного программирования**. Для поддержки этой идеи STL содержит множество обобщенных алгоритмов. Такие алгоритмы избавляют программиста от необходимости знать подробности отдельных контейнеров.

В STL определены следующие классы-контейнеры (в угловых скобках указаны заголовочные файлы, где определены эти классы):

bitset множество битов <bitset.h>
vector динамический массив <vector.h>
list линейный список <list.h>
deque двусторонняя очередь <deque.h>
stack стек <stack.h>
queue очередь <queue.h>
priority_queue очередь с приоритетом <queue.h>
map ассоциативный список для хранения пар ключ / значение, где с каждым ключом связано одно значение <map.h>
multimap с каждым ключом связано два или более значений <map.h>
set множество <set.h>

multiset множество, в котором каждый элемент не обязательно уникален <set.h>

Обзор операций

Типы

value_type тип элемента
allocator_type тип распределителя памяти
size_type тип индексов, счетчика элементов и т.д.
iterator ведет себя как value_type*
reverse_iterator просматривает контейнер в обратном порядке
reference ведет себя как value_type&
key_type тип ключа (только для ассоциативных контейнеров)
key_compare тип критерия сравнения (только для ассоциативных контейнеров)
mapped_type тип отображенного значения

Итераторы

begin() указывает на первый элемент
end() указывает на элемент, следующий за последним
rbegin() указывает на первый элемент в обратной последовательности
rend() указывает на элемент, следующий за последним в обратной последовательности

Доступ к элементам

front() ссылка на первый элемент
back() ссылка на последний элемент
operator[](i) доступ по индексу без проверки
at(i) доступ по индексу с проверкой

Включение элементов

insert(p,x**)** добавление x перед элементом, на который указывает p
insert(p,n,x**)** добавление n копий x перед p
insert(p,first,last**)** добавление элементов из [first:last] перед p
push_back(x**)** добавление x в конец
push_front(x**)** добавление нового первого элемента (только для списков и очередей с двумя концами)

Удаление элементов

pop_back() удаление последнего элемента
pop_front() удаление первого элемента (только для списков и очередей с двумя концами)
erase(p**)** удаление элемента в позиции p
erase(first,last**)** удаление элементов из [first:last]
clear() удаление всех элементов

Другие операции

size() число элементов
empty() контейнер пуст?
capacity() память, выделенная под вектор (только для векторов)
reserve(n**)** выделяет память для контейнера под n элементов
resize(n**)** изменяет размер контейнера (только для векторов, списков и очередей с двумя концами)
swap(x**)** обмен местами двух контейнеров
==, **!=**, **<** операции сравнения

Операции присваивания

operator=(x**)** контейнеру присваиваются элементы контейнера x
assign(n,x**)** присваивание контейнеру n копий элементов x (не для ассоциативных контейнеров)
assign(first,last**)** присваивание элементов из диапазона [first:last]

Ассоциативные операции

operator[](k**)** доступ к элементу с ключом k
find(k**)** находит элемент с ключом k
lower_bound(k**)** находит первый элемент с ключом k
upper_bound(k**)** находит первый элемент с ключом, большим k
equal_range(k**)** находит lower_bound (нижнюю границу) и upper_bound (верхнюю границу) элементов с ключом k

Контейнера *vector*-вектор.

Вектор vector в STL определен как динамический массив с доступом к его элементам по индексу.

template<class T,class Allocator=allocator<T>>class std::vector{…};

где *T* – тип предназначенных для хранения данных.

Allocator задает распределитель памяти, который по умолчанию является стандартным.

В классе vector определены следующие конструкторы:

explicit **vector**(const Allocator& a=Allocator());

explicit **vector**(size_type число, const T&значение= T(), const Allocator&a= =Allocator());

vector(const vector<T,Allocator>&объект);

template<class InIter>**vector**(InIter начало, InIter конец, const Allocator&a= =Allocator());

Первая форма представляет собой конструктор пустого вектора.

Во второй форме конструктора вектора число элементов – это число, а каждый элемент равен значению значение. Параметр значение может быть значением по умолчанию.

Третья форма конструктора вектор – это конструктор копирования.

Четвертая форма – это конструктор вектора, содержащего диапазон элементов, заданный итераторами начало и конец.

Пример 2.

vector<int> a;

```
vector<double> x(5);
vector<char> c(5,'*');
vector<int> b(a); //b=a
```

Для любого объекта, который будет храниться в векторе, должен быть определен конструктор по умолчанию. Кроме того, для объекта должны быть определены операторы < и ==.

Для класса вектор определены следующие операторы сравнения:
==, <, <=, !=, >, >=.

Кроме этого, для класса vector определяется оператор индекса [].

Новые элементы могут включаться с помощью функций
insert(), push_back(), resize(), assign().

Существующие элементы могут удаляться с помощью функций
erase(), pop_back(), resize(), clear().

Доступ к отдельным элементам осуществляется с помощью итераторов
begin(), end(), rbegin(), rend(),

Манипулирование контейнером, сортировка, поиск в нем и тому подобное возможно с помощью глобальных функций файла – заголовка <algorithm.h>.

Пример 3.

```
#include<iostream.h>
#include<vector.h>
using namespace std;
void main()
{vector<int> v;
int i;
for(i=0;i<10;i++)v.push_back(i);
cout<<"size="<<v.size()<<"\n";
for(i=0;i<10;i++)cout<<v[i]<<" ";
cout<<endl;
for(i=0;i<10;i++)v[i]=v[i]+v[i];
for(i=0;i<v.size();i++)cout<<v[i]<<" ";
cout<<endl;
}
```

Пример 4. Доступ к вектору через итератор.

```
#include<iostream.h>
#include<vector.h>
using namespace std;
void main()
{vector<int> v;
int i;
for(i=0;i<10;i++)v.push_back(i);
cout<<"size="<<v.size()<<"\n";
vector<int>::iterator p=v.begin();
while(p!=v.end())
{cout<<*p<<" ";p++;}
```

```
}
```

Пример 5. Вставка и удаление элементов.

```
#include<iostream.h>
#include<vector.h>
using namespace std;
void main()
{vector<int> v(5,1);
int i;
//вывод
for(i=0;i<5;i++)cout<<v[i]<<“ ”;
cout<<endl;
vector<int>::iterator p=v.begin();
p+=2;
//вставить 10 элементов со значением 9
v.insert(p,10,9);
//вывод
p=v.begin();
while(p!=v.end())
{cout<<*p<<” “;p++;}
//удалить вставленные элементы
p=v.begin();
p+=2;
v.erase(p,p+10);
//вывод
p=v.begin();
while(p!=v.end())
{cout<<*p<<” “;p++;}
}
```

Пример 6. Вектор содержит объекты пользовательского класса.

```
#include<iostream.h>
#include<vector.h>
#include”student.h”
using namespace std;
void main()
{vector<STUDENT> v(3);
int i;
v[0]=STUDENT(“Иванов”,45.9);
v[1]=STUDENT(“Петров”,30.4);
v[0]=STUDENT(“Сидоров”,55.6);
//вывод
for(i=0;i<3;i++)cout<<v[i]<<“ ”;
cout<<endl;
}
```

Ассоциативные контейнеры (массивы).

Ассоциативный массив содержит пары значений. Зная одно значение, называемое **ключом** (key), мы можем получить доступ к другому, называемому **отображенным значением** (mapped value).

Ассоциативный массив можно представить как массив, для которого индекс не обязательно должен иметь целочисленный тип:

V& operator[](const K&) возвращает ссылку на V, соответствующий K.

Ассоциативные контейнеры – это обобщение понятия ассоциативного массива.

Ассоциативный контейнер **map** – это последовательность пар (ключ, значение), которая обеспечивает быстрое получение значения по ключу. Контейнер map предоставляет двунаправленные итераторы.

Ассоциативный контейнер map требует, чтобы для типов ключа существовала операция “<”. Он хранит свои элементы отсортированными по ключу так, что перебор происходит по порядку.

Спецификация шаблона для класса map:

template<class Key,classT,class Comp=less<Key>,class Allocator=allocator<pair> >

class std::map

В классе map определены следующие конструкторы:

explicit **map**(const Comp& c=Comp(),const Allocator& a=Allocator());

map(const map<Key,T,Comp,Allocator>& ob);

template<class InIter> **map**(InIter first,InIter last,const Comp& c=Comp(),const Allocator& a=Allocator());

Первая форма представляет собой конструктор пустого ассоциативного контейнера, вторая – конструктор копии, третья – конструктор ассоциативного контейнера, содержащего диапазон элементов.

Определена операция присваивания:

map& **operator=**(const map&);

Определены следующие операции: ==, <, <=, !=, >, >=.

В map хранятся пары ключ/значение в виде объектов типа **pair**.

Создавать пары ключ/значение можно не только с помощью конструкторов класса pair, но и с помощью функции **make_pair**, которая создает объекты типа pair, используя типы данных в качестве параметров.

Типичная операция для ассоциативного контейнера – это ассоциативный поиск при помощи операции индексации ([]).

mapped_type& **operator[]**(const key_type& K);

Множества **set** можно рассматривать как ассоциативные массивы, в которых значения не играют роли, так что мы отслеживаем только ключи.

template<classT,class Cmp=less<T>,class Allocator=allocator<T>>class std::set{…};

Множество, как и ассоциативный массив, требует, чтобы для типа T существовала операция “меньше” (<). Оно хранит свои элементы отсортированными, так что перебор происходит по порядку.

Алгоритмы.

Каждый алгоритм выражается шаблоном функции или набором шаблонов функций. Таким образом, алгоритм может работать с очень разными контейнерами, содержащими значения разнообразных типов. Алгоритмы, которые возвращают итератор, как правило, для сообщения о неудаче используют конец входной последовательности. Алгоритмы не выполняют проверки диапазона на их входе и выходе. Когда алгоритм возвращает итератор, это будет итератор того же типа, что и был на входе. Алгоритмы в STL реализуют большинство распространенных универсальных операций с контейнерами, такие как просмотр, сортировка, поиск, вставка и удаление элементов.

Алгоритмы определены в заголовочном файле <algorithm.h>.

Ниже приведены имена некоторых наиболее часто используемых функций-алгоритмов STL.

I. Немодифицирующие операции.

for_earch() выполняет операции для каждого элемента последовательности
find() находит первое вхождение значения в последовательность
find_if() находит первое соответствие предикату в последовательности
count() подсчитывает количество вхождений значения в последовательность
count_if() подсчитывает количество выполнений предиката в последовательности
search() находит первое вхождение последовательности как подпоследовательности
search_n() находит n-е вхождение значения в последовательность

II. Модифицирующие операции.

copy() копирует последовательность, начиная с первого элемента
swap() меняет местами два элемента
replace() заменяет элементы с указанным значением
replace_if() заменяет элементы при выполнении предиката
replace_copy() копирует последовательность, заменяя элементы с указанным значением
replace_copy_if() копирует последовательность, заменяя элементы при выполнении предиката
fill() заменяет все элементы данным значением
remove() удаляет элементы с данным значением
remove_if() удаляет элементы при выполнении предиката

remove_copy() копирует последовательность, удаляя элементы с указанным значением
remove_copy_if() копирует последовательность, удаляя элементы при выполнении предиката
reverse() меняет порядок следования элементов на обратный
random_shuffle() перемещает элементы согласно случайному равномерному распределению (“тасует” последовательность)
transform() выполняет заданную операцию над каждым элементом последовательности
unique() удаляет равные соседнии элементы
unique_copy() копирует последовательность, удаляя равные соседние элементы

III. Сортировка.

sort() сортирует последовательность с хорошей средней эффективностью
partial_sort() сортирует часть последовательности
stable_sort() сортирует последовательность, сохраняя порядок следования равных элементов
lower_bound() находит первое вхождение значения в отсортированной последовательности
upper_bound() находит первый элемент, больший чем заданное значение
binary_search() определяет, есть ли данный элемент в отсортированной последовательности
merge() сливает две отсортированные последовательности

IV. Работа с множествами.

includes() проверка на вхождение
set_union() объединение множеств
set_intersection() пересечение множеств
set_difference() разность множеств

V. Минимумы и максимумы.

min() меньшее из двух
max() большее из двух
min_element() наименьшее значение в последовательности
max_element() наибольшее значение в последовательности

VII. Перестановки.

next_permutation() следующая перестановка в лексикографическом порядке
pred_permutation() предыдущая перестановка в лексикографическом порядке

6.4 Порядок выполнения работы.

Написать и отладить три программы. Первая программа демонстрирует использование контейнерных классов для хранения встроенных типов данных.

Вторая программа демонстрирует использование контейнерных классов для хранения пользовательских типов данных.

Третья программа демонстрирует использование алгоритмов STL.

В программе № 1 выполнить следующее:

1. Создать объект-контейнер в соответствии с вариантом задания и заполнить его данными, тип которых определяется вариантом задания.
2. Просмотреть контейнер.
3. Изменить контейнер, удалив из него одни элементы и заменив другие.
4. Просмотреть контейнер, используя для доступа к его элементам итераторы.
5. Создать второй контейнер этого же класса и заполнить его данными того же типа, что и первый контейнер.
6. Изменить первый контейнер, удалив из него n элементов после заданного и добавив затем в него все элементы из второго контейнера.
7. Просмотреть первый и второй контейнеры.

В программе № 2 выполнить то же самое, но для данных пользовательского типа.

В программе № 3 выполнить следующее:

1. Создать контейнер, содержащий объекты пользовательского типа. Тип контейнера выбирается в соответствии с вариантом задания.
2. Отсортировать его по убыванию элементов.
3. Просмотреть контейнер.
4. Используя подходящий алгоритм, найти в контейнере элемент, удовлетворяющий заданному условию.
5. Переместить элементы, удовлетворяющие заданному условию в другой (предварительно пустой) контейнер. Тип второго контейнера определяется вариантом задания.
6. Просмотреть второй контейнер.
7. Отсортировать первый и второй контейнеры по возрастанию элементов.
8. Просмотреть их.
9. Получить третий контейнер путем слияния первых двух.
10. Просмотреть третий контейнер.
11. .Подсчитать, сколько элементов, удовлетворяющих заданному условию, содержит третий контейнер.
12. Определить, есть ли в третьем контейнере элемент, удовлетворяющий заданному условию.

6.5 Методические указания.

2. В качестве пользовательского типа данных использовать пользовательский класс лабораторной работы № 6.

3. При создании контейнеров в программе № 2 объекты загружать из потока, для чего использовать программы записи и чтения потока из лабораторной работы № 6.

4. Для вставки и удаления элементов контейнера в программе № 2 использовать соответствующие операции, определенные в классе контейнера.

5. Для создания второго контейнера в программе № 3 можно использовать либо алгоритм **remove_copy_if**, либо определить свой алгоритм **copy_if**, которого нет в STL.

6. Для поиска элемента в коллекции можно использовать алгоритм **find_if**, либо **for_each**, либо **binary_search**, если контейнер отсортирован.

7. Для сравнения элементов при сортировке по возрастанию используется операция <, которая должна быть перегружена в пользовательском классе. Для сортировки по убыванию следует написать функцию **comp** и использовать вторую версию алгоритма **sort.**

8. Условия поиска и замены элементов выбираются самостоятельно и для них пишется функция-предикат.

9. Для ввода-вывода объектов пользовательского класса следует перегрузить операции “>>” и “<<”.

10. Некоторые алгоритмы могут не поддерживать используемые в вашей программе контейнеры. Например, алгоритм **sort** не поддерживает контейнеры, которые не имеют итераторов произвольного доступа. В этом случае следует написать свой алгоритм. Например, для стека алгоритм сортировки может выполняться следующим образом: переписать стек в вектор, отсортировать вектор, переписать вектор в стек.

10.При перемещении элементов ассоциативного контейнера в неассоциативный перемещаются только данные (ключи не перемещаются). И наоборот, при перемещении элементов неассоциативного контейнера в ассоциативный должен быть сформирован ключ.

6.6 Содержание отчета.

1. Титульный лист.

2. Постановка задач.

3. Определение пользовательского класса.

4. Определения используемых в программах компонентных функций для работы с контейнером, включая конструкторы.

5. Объяснение этих функций.

6. Объяснение используемых в программах алгоритмов STL.

7. Определения и объяснения, используемых предикатов и функций сравнения.

6.7 Варианты заданий.

№ п/п	Первый контейнер	Второй контейнер	Встроенный тип данных
1	vector	list	int
2	list	deque	long
3	deque	stack	float
4	stack	queue	double
5	queue	vector	char
6	vector	stack	string
7	map	list	long
8	multimap	deque	float
9	set	stack	int
10	multiset	queue	char
11	vector	map	double
12	list	set	int
13	deque	multiset	long
14	stack	vector	float
15	queue	map	int
16	priority_queue	stack	char
17	map	queue	char
18	multimap	list	int
19	set	map	char
20	multiset	vector	int

7 Графика

7.1 Цель.

Развить практические навыки работы с графикой в условиях алфавитно – цифрового экрана для программ C++ Linux.

7.2 Основное содержание работы.

Сформировать объектную систему отображения заданных графических объектов средствами псевдографики и ncurses в изменяемом окне экрана с использованием цвета и других эффектов отображения знаков.

7.3 Краткие теоретические сведения

Используется две технологии вывода информации на экран в алфавитно – цифровом режиме: псевдографика и ncurses.

Псевдографика. В данном режиме формируется массив символов по числу символов на экране. Объект: график или рисунок, переносится в массив, и указанный массив выводится на печать.

Ncurses. Используется специальная библиотека.

Для того, чтобы предоставить интерфейс «укажи и щелкни» пользователям текстовых терминалов, была разработана библиотека curses (ее название происходит от ее важнейшей функции – управления курсором, а

вовсе не от проклятья, которая она накладывает на программистов). Изначально библиотека curses создавалась для BSD UNIX. В Linux используется открытый (на условиях MIT License) клон curses – библиотека ncurses (new curses).

Основными концепциями пользовательского интерфейса программы, использующей ncurses, являются экран (screen), окно (window) и под - окно (sub-window). Экраном называется все пространство, на котором ncurses может выводить данные. С точки зрения ncurses, экран – это матрица ячеек, в которые можно выводить символы. Если монитор работает в текстовом режиме, экран ncurses совпадает с экраном монитора. Если терминал эмулируется графической программой, экраном является рабочая область окна этой программы. Окном ncurses называется прямоугольная часть экрана, для которой определены особые параметры вывода. В частности, размеры окна влияют на перенос и прокрутку строк, выводимых в этом окне. В каком-то смысле окно можно назвать «экраном в экране». На уровне интерфейса программирования окна представлены структурами данных, по этой причине мы будем часто говорить об окне как о структуре.

В процессе инициализации ncurses автоматически создается окно stdscr, размеры которого совпадают с размерами экрана. Кроме структуры stdscr по умолчанию создается еще одна структура – curscr. Операции вывода данных ncurses модифицируют содержимое структуры stdscr, однако, на экране всегда отображается содержимое окна curscr. Иначе говоря, данные, которые выводит программа в окно stdscr (или в другое окно), не отображаются на экране монитора автоматически. Для того чтобы сделать результаты вывода видимыми, вы должны вызывать специальные функции обновления экрана (refresh() или wrefresh()). Эти функции сравнивают содержимое окон stdscr и curscr и на основе различий между ними вносят изменения в структуру curscr, а затем обновляют экран. Благодаря наличию окна curscr, ncurses-программе не требуется «помнить» весь свой предыдущий вывод и перерисовывать его всякий раз, когда в этом возникает необходимость. Этим программы ncurses отличаются от графических программ

Хотя ваша программа может пользоваться для вывода данных исключительно окном stdscr, ваша задача по проектированию интерфейса существенно упростится, если вы будете создавать собственные окна, расположенные «внутри» stdscr. Программа, использующая ncurses, может работать с несколькими окнами одновременно, выполняя вывод в каждое из них. Кроме окон (windows) программы ncurses могут создавать под – окна (subwindows), поведение которых несколько отличается от поведения стандартных окон.

Важнейшей особенностью ncurses является возможность указать произвольную позицию курсора для вывода (и ввода) данных. Позиция курсора отсчитывается от левого верхнего угла текущего окна. Ячейка в верхнем левом углу имеет координаты (0, 0). При работе с функциями ncurses важно помнить, что первой координатой является номер строки, (что

соответствует y, в терминах графического программирования), а второй координатой – номер столбца (что соответствует x в графическом режиме).

В случае ошибки функции ncurses обычно возвращают константу ERR. Если функция не должна возвращать какое-то информативное значение (как, например, функция getch()), в случае успешного выполнения она возвращает значение OK.

Прежде чем переходить к программированию ncurses, следует рассмотреть решение одной задачи, с которой в настоящее время сталкиваются все разработчики, использующие эту библиотеку. Речь идет об изменении размеров окна терминала (под размерами окна в данном случае понимается число строк и столбцов). Пользователи настоящих текстовых терминалов редко переключали их режимы, и готовы были мириться с последствиями своих действий. В наши дни, когда экраном терминала зачастую служит окно графической программы, пользователь вправе ожидать, что при изменении размеров окна работа консольной программы не нарушится, а ее интерфейс не развалится.

Когда размеры окна терминала меняются, выполняющаяся в нем программа получает сигнал SIGWINCH. Это одновременно и хорошо и плохо. Хорошо – потому, что терминал информирует программу об изменении своих размеров, плохо – потому, что сигналы имеют особенность вмешиваться в работу программы. Например, если вы напишете программу, использующую ncurses, и не позаботитесь об обработке сигнала SIGWINCH, при изменении размеров окна терминала ваша программа может неожиданно завершиться, оставив терминал в неканоническом состоянии. Рассмотрим, как обрабатывается сигнал SIG_WINCH в программе cursed.

```
void sig_winch(int signo)
{
        struct winsize size;
        ioctl(fileno(stdout), TIOCGWINSZ, (char *) &size);
        resizeterm(size.ws_row, size.ws_col);
}
```

Функция sig_winch() представляет собой обработчик сигнала SIGWINCH.

Следует отметить, что изменение размеров окна программы, работающей в текстовом режиме, представляет собой довольно нетривиальную задачу и стандартного рецепта, описывающего, что должна делать программа, когда размеры окна изменились, не существует. Разработчики ncurses постарались упростить решение этой задачи, введя функцию resizeterm(). Функцию resizeterm() следует вызывать сразу после изменения размеров окна терминала. Аргументами функции resizeterm() должны быть новые размеры экрана, заданные в строках и столбцах. Функция resizeterm() старается сохранить внешний вид и порядок работы приложения в изменившемся окне терминала, но это ей удается не всегда, с чем мы столкнемся ниже. Необходимые для resizeterm() значения размеров окна мы получаем с помощью специального вызова ioctl(). При этом первым

параметром функции ioctl() должен быть дескриптор файла устройства, представляющего терминал. Вторым параметром ioctl() является константа TIOCGWINSZ, а третьим параметром – адрес структуры winsize. Структура winsize определенная в файле <sys/ioctl.h>, включает в себя поля ws_row и ws_col, в которых возвращается число строк и столбцов окна терминала.

Перейдем теперь к функции main() программы cursed:

```
int main(int argc, char ** argv)
{
	initscr();
	signal(SIGWINCH, sig_winch);
	cbreak();
	noecho();
	curs_set(0);
	attron(A_BOLD);
	move(5, 15);
	printw("Hello, brave new curses world!\n");
	attroff(A_BOLD);
	attron(A_BLINK);
	move(7, 16);
	printw("Press any key to continue...");
	refresh();
	getch();
	endwin();
	exit(EXIT_SUCCESS);
}
```

Работа с ncurses начинается с вызова функции initscr(). Эта функции инициализирует структуры данных ncurses и переводит терминал в нужный

режим. По окончании работы с ncurses следует вызвать функцию endwin(), которая восстанавливает то состояние, в котором терминал находился до инициализации ncurses. После вызова initscr() мы устанавливаем обработчик сигнал SIGWINCH. Устанавливать обработчик SIGWINCH следует только после инициализации ncurses, поскольку в обработчике используется функция resizeterm(), предполагающая, что библиотека ncurses уже инициализирована.

Функция noecho() отключает отображение символов, вводимых с клавиатуры. Функция cur_set() управляет видимостью курсора. Если вызвать эту функцию с параметром 0, курсор станет невидимым, вызов же функции с ненулевым параметром снова «включает» курсор.

Функция attron() позволяет указать некоторые дополнительные атрибуты выводимого текста. Этой функции можно передать одну или несколько констант, обозначающих атрибуты (в последнем случае их следует объединить с помощью операции «|»). Например, атрибут A_UNDERLINE включает подчеркивание текста, атрибут A_REVERSE меняет местами цвет фона и текста, атрибут A_BLINK делает текст мигающим, атрибут A_DIM снижает яркость текста по сравнению с нормальной, атрибут A_BOLD делает

текст жирным в монохромном режиме и управляет яркостью цвета в цветном режиме работы монитора. Специальный атрибут COLOR_PAIR() применяется для установки цветов фона и текста.

Окно ncurses представляет собой матрицу ячеек для вывода символов. Помимо кода символа каждая ячейка содержит дополнительные атрибуты символа.

Режимы вывода

Для символов типа chtype можно устанавливать такие атрибуты, как мигание или цвет символа и фона. Для добавления символу атрибута мигания нужно включить флажок A_BLINK. Далается это так: chtype ch = 'w' | A_BLINK; Теперь при выводе этого символа он будет мигать, если конечно это позволяет сделать терминал. (A_DIM - пониженная яркость, A_BOLD - повышенная яркость, A_NORMAL - нормальное отображение, A_UNDERLINE - подчёркнутый, A_REVERSE - инверсный)

С включением цвета немного сложнее. Перед использованием цветов нужно проинициализировать палитру. Палитра это структура, в которой определённой цифре соответствует определённый цвет. В нашем случае одной цифре соответствуют сразу два цвета символов и фона.

```
...
chtype ch;
...
if (!has_colors())
{
        endwin();
        printf("Цвета не поддерживаются");
        exit(1);
}
start_color();

// 1 цвет в палитре - красные символы на чёрном фоне
init_pair(1, COLOR_RED, COLOR_BLACK);

// 2 цвета в палитре - зелёные символы на желтом фоне
init_pair(2, COLOR_GREEN, COLOR_YELLOW);


...
ch = 'w' | COLOR_PAIR(1); // символ с цветом 1 из палитры
```

Функция has_colors позволяет узнать можно ли использовать цвета. Функция start_color() должна вызываться до задания палитры. Функция init_pair() нужна чтобы задать какой цифре какой цвет будет соответствовать от 1 до COLOR_PAIRS-1 (0 зарезервирован для стандартного отображения). Для использования цвета в символе нужно включить флажок COLOR_PAIR(номер из палитры).

Список цветов:

COLOR_BLACK

COLOR_RED
COLOR_GREEN
COLOR_YELLOW
COLOR_BLUE
COLOR_MAGENTA
COLOR_CYAN
COLOR_WHITE

Следующие функции позволяют установить атрибуты вывода по умолчанию:

Включение атрибутов

int attron(int attrs)

включает атрибуты attrs. (Например attron(COLOR_PAIR(1)); устанавливает цвет 1 из палитры)

Установка атрибутов

int attrset(int attrs)

Заменяет текущие атрибуты атрибутами attrs (Например attrset(A_NORMAL); заменяет текущие атрибуты на A_NORMAL)

Установка атрибутов очистки

void bkgdset(chtype ch)

Устанавливает атрибуты с которыми очищается экран такими функциями как clear(). (Например bkgdset(COLOR_PAIR(1)); очистка будет осуществляться цветом 1 из палитры)

Сбросить атрибуты можно с помощью функции attroff().

Так же, как и в случае с attron(), функции attroff() можно передать несколько констант, обозначающих атрибуты, разделенных символом «|». Так же, как и установка атрибута, сброс атрибута влияет только на текст, напечатанный после сброса (текст, напечатанный ранее с установленным атрибутом, остается без изменений). В нашей программе мы сначала устанавливаем атрибут A_BOLD. Теперь, до тех пор, пока мы не сбросим этот атрибут, весь текст будет печататься жирным шрифтом.

Функция getch() предназначена для считывания символов из потока ввода терминала. Функция считывает по одному символу и может работать в двух режимах: блокирующем (режим по умолчанию) и неблокирующем. В блокирующем режиме функция приостанавливает выполнение программы до появления символа в потоке ввода, а в неблокирующем – возвращает значение сразу же, независимо от того, есть ли символ в потоке ввода или нет (если символа в потоке ввода нет, функция getch() в неблокирующем режиме возвращает значение ERR). В режиме cbreak() (о котором подробнее будет рассказано во второй части статьи) функция, считавшая символ, передает его программе, не дожидаясь, пока пользователь нажмет [Enter]. Таким образом, программа cursed завершается сразу же после нажатия на любую клавишу.

Интерфейс программирования ncurses не является частью glibc, а вынесен в отдельную библиотеку libncurses, поэтому во время сборки

программы эту библиотеку нужно подключать явным образом, например: gcc cursed.c –o cursed –lncurses

Окна

В текстовых интерфейсах, построенных на основе ncurses, окна играют такую же важную роль, что и в графических интерфейсах. Прежде чем переходить к созданию приложений, использующих окна, необходимо внести некоторые уточнения в описание интерфейса ncurses. Прежде всего, вы должны понимать, что при работе с ncurses вы всегда имеете дело с окнами. В рассмотренной выше программе cursed мы работали с окном stdscr. Выше уже отмечалось, что каждое окно ncurses описано структурой данных, однако при работе с stscr нам не приходилось иметь дело ни с какими специальными структурами. Объясняется это тем, что в программе cursed мы использовали функции (attron(), move(), printw(), attroff(), getch()), специально предназначенные для работы с окном stdscr. Поскольку эти функции работают исключительно с окном stdscr, передавать им структуру, описывающую окно, не требуется. Для работы с другими окнами нам придется использовать обобщенные варианты функций. Списки параметров обобщенных функций совпадают со списками параметров функций, предназначенных для работы с stdscr, за исключением того, что первым параметром обобщенной функции должен быть указатель на структуру WINDOW, определяющую окно, для которого вызывается функция. Например, для установки атрибутов текста в произвольном окне применяется функция wattron(). Первым параметром этой функции служит указатель на структуру WINDOW, а второй параметр wattron() полностью аналогичен параметру функции attron().

Как получить указатель на структуру WINDOW, соответствующую некоторому окну? Переменная stdscr, которую экспортирует библиотека ncurses, содержит указатель на структуру WINDOW, представляющую корневое окно stdscr. Эта переменная определена как

extern WINDOW * stdscr;

Из того, что stdscr является обычным окном ncursees, следует, что вместо функций, предназначенных специально для stdscr, мы можем использовать их обобщенные аналоги, указывая переменную stdscr в качестве идентификатора окна. Например, вызов attron(A_BOLD) эквивалентен вызову wattron(stdscr, A_BOLD).

Обобщенным вариантом функции attroff() является функция wattroff(), а обобщенным вариантом функции move() функция wmove(). Вызов move(5, 15); из программы cursed можно заменить вызовом wmove(stdscr, 5, 15);

Функции printw() соответствует обобщенная функция wprintw(). Функции getch() соответствует функция wgetch(), аргументом которой должен быть все тот же указатель на WINDOW. Отметим, что функциями getch()/wgetch() и printw()/wprintw() не исчерпывается многообразие функций ввода/вывода символов ncurses.

Смысл создания дополнительных окон заключается в том, что мы ограничиваем область вывода текста (и область применения различных

атрибутов) отдельными участками экрана. Создав в некоторой области экрана окно, мы можем быть уверены, что вывод данных, который мы выполняем в этом окне, никак не повлияет на остальной экран.

Библиотека ncurses предоставляет в наше распоряжение несколько функций, создающих новые окна. Самой простой и часто используемой является функция newwin(). У функции newwin() четыре параметра. Первые два параметра соответствуют количеству строк и столбцов в создаваемом окне, а вторые два указывают положение верхнего левого угла нового окна (строка и столбец) относительно окна stdscr.

Функция newwin() возвращает указатель на структуру WINDOW (или NULL в случае ошибки). После завершения работы с окном, выделенные ему ресурсы следует высвободить с помощью функции delwin(). Единственный параметр этой функции – указатель на структуру WINDOW, которую следует удалить.

Отметим, что удаление окна с помощью delwin() само по себе не влияет на содержимое экрана. Данные, выведенные в окно останутся на экране до тех пор, пока не будут перезаписаны другими данными.

Помимо функции newwin() нам будет полезно познакомиться еще с двумя функциями: subwin() и derwin(). Эти две функции предназначены для создания под-окон. Списки параметров у этих функций такие же, как и у newwin(), с той разницей, что первым параметром каждой функции является указатель на структуру WINDOW, соответствующую родительскому окну.

Последние два аргумента у subwin() и derwin() интерпретируются по-разному. У функции subwin() они задают положение верхнего левого угла окна относительно экрана, а у функции derwin() – относительно родительского окна. Чем же под-окно отличается от обычного окна? Окно и его под-окно разделяют массив, в котором хранятся символы и их атрибуты. Новое под- окно наследует все атрибуты своего родителя. Эти атрибуты затем могут быть изменены, что не повлияет на атрибуты родительского окна.

Займемся созданием окон в программе cursedwindows. Список заголовочных файлов и обработчик сигнала SIG_WINCH у программы cursedwindows такие же, как и у программы cursed, так что в листинге мы их пропустим и рассмотрим только функцию main().

```
int main(int argc, char ** argv)
{
        WINDOW * wnd;
        WINDOW * subwnd;
        initscr();
        signal(SIGWINCH, sig_winch);
        cbreak();
        curs_set(0);
        refresh();
        wnd = newwin(6, 18, 2, 4);
        box(wnd, '|', '-');
```

```
    subwnd = derwin(wnd, 4, 16, 1, 1);
    wprintw(subwnd, "Hello, brave new curses world!\n");
    wrefresh(wnd);
    delwin(subwnd);
    delwin(wnd);
    move(9, 0);
    printw("Press any key to continue...");
    refresh();
    getch();
    endwin();
    exit(EXIT_SUCCESS);
}
```

В функции main() мы инициализируем ncurses с помощью функции initscr() и устанавливаем обработчик SIGWINCH. Далее делаем курсор невидимым. После этого мы должны обновить экран с помощью refresh().

Мы создаем новое окно с помощью функции newwin(). Наше окно насчитывает 6 строк и 18 столбцов и его верхний левый угол находится в ячейке (2, 4) окна stdscr. Указатель на структуру WINDOW, который возвращает функция newwin(), мы сохраняем в переменной wnd. Функция box(), которую мы вызываем далее, позволяет создать рамку вдоль границы окна. Аргументами этой функции должны быть идентификатор окна и символы, используемые, соответственно, для рисования вертикальной и горизонтальной границы. Теперь было бы логично вывести какой-нибудь текст в окно, обрамленное рамкой, но тут возникает одна сложность. Поскольку символы рамки сами находятся внутри окна, символы текста могут затереть их в процессе вывода. Мы решаем эту проблему с помощью создания под-окна subwnd внутри окна wnd и вывода текста в это под-окно. Поскольку окно subwnd по размерам меньше, чем окно wnd, символы рамки не будут стерты.

Теперь мы можем распечатать текст, что мы и делаем с помощью функции wprintw(), указав ей идентификатор окна subwnd. Для того чтобы символы, напечатанные в окне, стали видимыми, мы должны вызвать функцию wrefresh(). Мы вызываем эту функцию только для окна wnd, поскольку оно содержит символьный массив и своего под-окна subwnd.

Обратите внимание на то, что символы строки "Hello, brave new curses world!", которую мы печатаем в под-окне subwnd с помощью функции wprintw(), переносятся при достижении границы под-окна (рис. 2). После завершения работы с окнами мы можем удалить структуры wnd и subwnd с помощью функции delwin(). Весь вывод, выполненный в окне wnd, останется на экране (точнее в окне curscr) до тех пор, пока вы не перезапишете его другим выводом.

7.4 Порядок выполнения работы.

1. Определить иерархию классов рисования (в соответствии с вариантом).
2. Определить методы классов
3. Реализовать классы.

4. Сформировать в виде псевдокода технологию задания и обработки размеров окон и вида знаков.
5. Написать демонстрационную программу, в которой рисуются требуемые объекты в псевдографике и ncurses технологиях.

7.5 Методические указания.

Сформировать облик класса (свойства и методы), необходимые для получения графика. Допустим, нужно нарисовать окружность и линию.

Формируется класс

```
//файл  Drawing.h
#pragma once
class Drawing
{
   public: virtual void drawLine (
      double xl, double yl,
      double x2, double y2) = 0;
      virtual void drawCircle (
      double x, double y,
      double r) = 0;
};
```

с виртуальными методами drawLine и drawCircle.

Далее формируются потомки класса Drawing – класс V1Drawing, отвечающий за методы псевдографики

```
//Файл V1Drawing.h
#include " Drawing.h"
class V1Drawing : public Drawing
{
   public: void drawLine(
      double xl, double yl,
      double x2, double y2);
   public:
void drawCircle(
      double x, double y, double r);
};
```

и класс V2Drawing, отвечающий за методы ncurses

```
//Файл V2Drawing.h
include "Drawing.h"
class V2Drawing : public Drawing {
   public: void drawLine (
      double xl, double yl,
      double x2, double y2);
   public:
       void drawCircle(
         double x, double y, double r);
};
```

В основной программе задать возможность изменения окон и выбора метода рисования.
В графике обязательно используются эффекты изменения окна, цвета, мерцания и жирности.

7.6 Содержание отчета.

1. Титульный лист: название дисциплины; номер и наименование работы; фамилия, имя, отчество студента; дата выполнения.
2. Постановка задачи. Следует дать конкретную постановку, т.е. указать, какие классы должны быть реализованы, какие должны быть в них конструкторы, компоненты-функции и т.д.
3. Иерархия классов в виде графа.
4. Определение пользовательских классов с комментариями.
5. .Реализация конструкторов с параметрами и деструктора.
6. Реализация методов.
7. Описание используемой технологии обработки изменяемых размеров окон и характеристик знаков.
8. Листинг демонстрационной программы.
9. Объяснение необходимости виртуальных функций. Следует показать, какие результаты будут в случае виртуальных и не виртуальных функций.

7.7 Варианты заданий.

Выбираются два объекта из задач на графику по задачнику 2 семестра 1- го курса. Берется одна задача на рисование и одна на движение.

8 Шаблоны программирования

8.1 ***Цель.***

Получить практические навыки работы с шаблонами в программах C++.

8.2 Основное содержание работы.

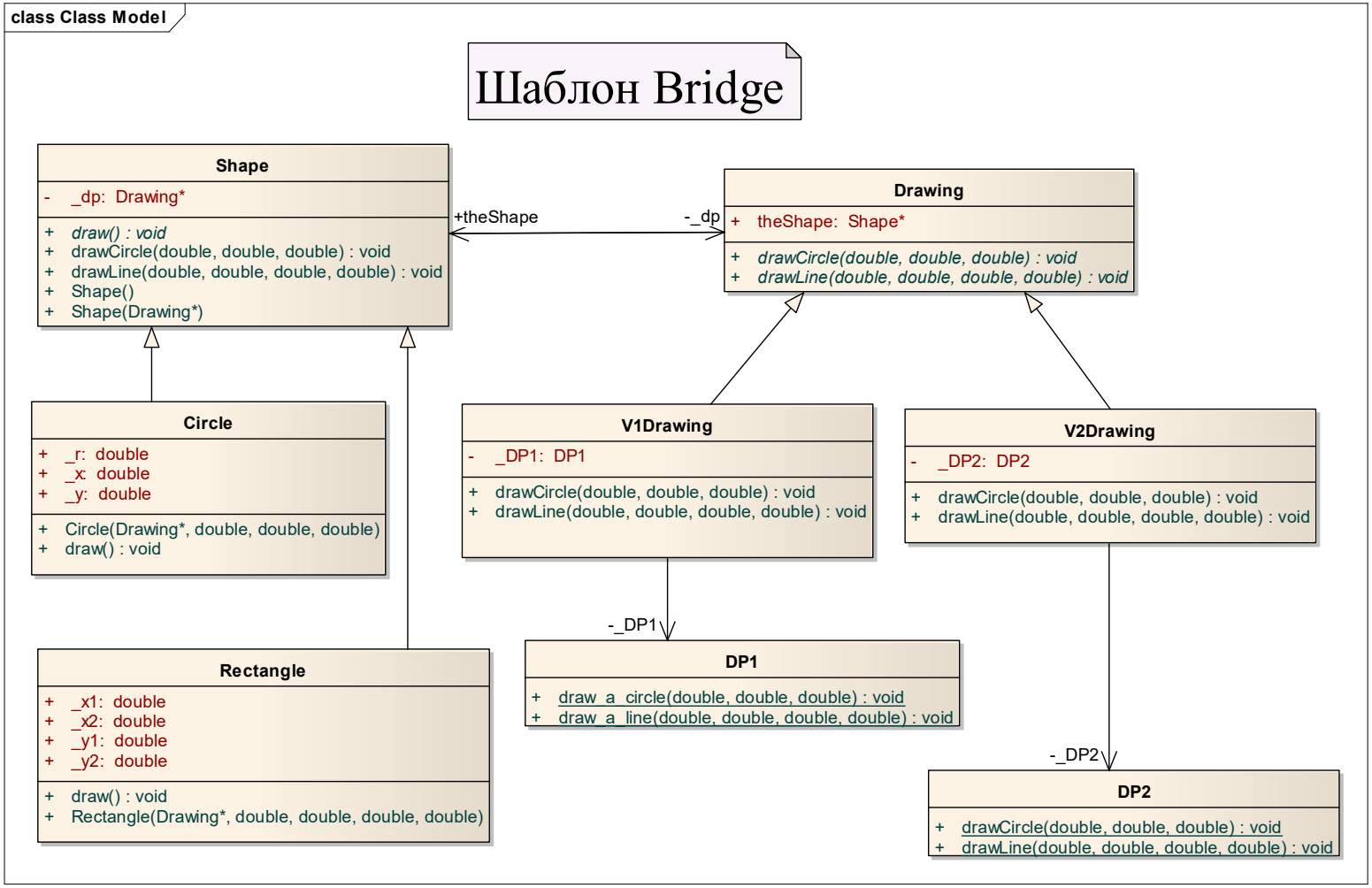
Сформировать систему построения графических объектов средствами псевдографики и ncurses с использованием шаблона Bridge.

8.3 Краткие теоретические сведения

Шаблон Bridge применяется в ситуации, когда некоторый комплекс объектов обрабатывается набором технологий. Количество объектов и количество технологий не связано и может увеличиваться.

Рассмотрим методику использования шаблона Bridge для рисования в условиях алфавитно – цифрового экрана. Предполагается, что имеется два объекта рисования и два метода рисования.

Диаграмма шаблона имеет вид:



UML диаграмма шаблона Bridge

Класс Shape является общим для объектов рисования Circle и Rectangle.

Класс Drawing содержит методы рисования, сгруппированные к классах V1Drawing и V2Drawing. Для совместимости с возможными прежними разработками добавлены классы DP1 и DP2.

8.4 Порядок выполнения работы

1. Определить иерархию классов объектов рисования (в соответствии с вариантом).
2. Определить методы классов с учетом иерархии
3. Реализовать классы.
4. Написать демонстрационную программу мультипликации, в которой рисуются требуемые объекты в псевдографике и ncurses технологиях.

8.5 Методические указания.

Реализовать требуемые действия, рассматривая приведенные тексты как псевдокод.

```
//Функция Main
#include <iostream.h>

#include "Shape.h"
#include "Drawing.h"
#include "V1Drawing.h"
#include "V2Drawing.h"
#include "Rectangle.h"
#include "Circle.h"
//using namespace std;
int main ()
{
   Shape *s1;
   Shape *s2;
   Drawing *dp1, *dp2;

   dp1 = new V1Drawing;
   s1 = new Rectangle(dp1, 1.0, 1.0, 2.0, 2.0);

   dp2 = new V2Drawing;
   s2= new Circle(dp2, 2, 2, 4);

   s1->draw() ;
   s2->draw();

   float a;
   cin >>  a;
   delete s1; delete s2;
   delete dp1; delete dp2;
   return 0;
```

```
}

// Shape.h: interface for the Shape class.
#pragma once
#include "Drawing.h"
class Shape
{
public:
    //Чистая  виртуальна  функция.
   // Параметры не указаны, поскольку их количество  и тип могут меняться
   virtual void draw() = 0;
   Shape();
   Shape(Drawing *dp);
   void drawLine(double xl, double yl,
      double x2, double y2);
   void drawCircle(double x, double y,
      double r);
private: Drawing *_dp;
};

// Drowing.h: interface for the Drowing class.

#pragma once
class Drawing
{
   public: virtual void drawLine (
      double xl, double yl,
      double x2, double y2) = 0;
      virtual void drawCircle (
      double x, double y,
      double r) = 0;
};

// Rectangle.h: interface for the Rectangle class.

#include "Shape.h"

#include "Drawing.h"
class Rectangle : public Shape{
   public:
         Rectangle::Rectangle(Drawing *dp,
         double x1, double y1,
         double x2, double y2);
         double _x1; double _y1;
         double _x2; double _y2;
```

```
        void draw();
};
// Circle.h: interface for the Circle class.

#include "Shape.h"

#include "Drawing.h"
class Circle : public Shape{
   public: Circle::Circle(
      Drawing *dp,
      double x, double y, double r);
   public: double _x; double _y; double _r;
   void draw();
};

// V1Drawing.h: interface for the V1Drawing class.

#include "Drawing.h"

#include "DP1.h"
class V1Drawing : public Drawing{
   private: DP1 _DP1;
   public: void drawLine(
      double xl, double yl,
      double x2, double y2);
   public:
void drawCircle(
      double x, double y, double r);
};

// V2Drawing.h: interface for the V2Drawing class.

#include "Drawing.h"

#include "DP2.h"
class V2Drawing : public Drawing {
   private: DP2 _DP2;
   public: void drawLine (
      double xl, double yl,
      double x2, double y2);
   public:
 void drawCircle(
        double x, double y, double r);
};
```

```
// DP1.h: interface for the DP1 class.

class DP1 {
public:
   static void draw_a_line (
      double xl, double yl,
      double x2, double y2);
   static void draw_a_circle (
      double x, double y, double r);
};
// DP2.h: interface for the DP2 class.

class DP2 {
   public:
      static void drawLine (
         double xl, double x2,
         double yl, double y2);
      static void drawCircle (
         double x, double y, double r);
};
# Makefile for Bridge project

bridge: Main.o Circle.o DP1.o DP2.o Drawing.o Rectangle.o Shape.o
V1Drawing.o V2Drawing.o
   g++ -o bridge Main.o Circle.o DP1.o DP2.o Drawing.o Rectangle.o Shape.o
V1Drawing.o V2Drawing.o
Main.o: Main.cpp
   g++ -c Main.cpp
Circle.o: Circle.cpp
   g++ -c Circle.cpp
DP1.o: DP1.cpp
   g++ -c DP1.cpp
DP2.o: DP2.cpp
   g++ -c DP2.cpp
Drawing.o: Drawing.cpp
   g++ -c Drawing.cpp
Rectangle.o: Rectangle.cpp
   g++ -c Rectangle.cpp
Shape.o: Shape.cpp
   g++ -c Shape.cpp
V1Drawing.o: V1Drawing.cpp
   g++ -c V1Drawing.cpp
V2Drawing.o: V2Drawing.cpp
   g++ -c V2Drawing.cpp
clean:
```

```
rm -f *.o bridge
```

8.6 Содержание отчета

1. Титульный лист: название дисциплины; номер и наименование работы; фамилия, имя, отчество студента; дата выполнения.
2. Постановка задачи. Следует дать конкретную постановку, т.е. указать, какие классы должны быть реализованы, какие должны быть в них конструкторы, компоненты-функции и т.д.
3. Иерархия классов в виде графа.
4. Определение технологии наследования с доказательством оптимальности.
5. Определение конструкторов классов и пояснение структуры аргументов конструкторов.
6. Реализация методов.
7. Make оператор
8. Листинг демонстрационной программы.
9. Объяснение необходимости виртуальных функций. Следует показать, какие результаты будут в случае виртуальных и не виртуальных функций.

8.7 Варианты заданий

В качестве классов рисования выступают классы работы 7.

В качестве объектов рисования берутся два объекты мультипликации из пункта Эффект мультипликации раздела Графика за 1 курс.

9 Исключительные ситуации

9.1 ***Цель.***

Получить практические навыки создания обработки исключений в программах на языке C++.

9.2 Содержание работы.

Определить и обработать систему исключений для заданного множества объектов.

9.3 Краткие теоретические сведения

При выполнении операторов программы могут встретиться проблемы различного уровня и характера. Одни из них непреодолимы, например, деление на 0, другие могут потребовать вмешательства оператора. Например, вводимая фамилия содержит более 255 символов.

При невозможности продолжать выполнение программы, в системе возбуждается исключительная ситуация. Многие языки программирования, в том числе язык C++, предусматривают исключительные ситуации (exceptions), представляющие собой механизм для обработки ошибок.

Для обработки проблем типа слишком длинной фамилии, можно сгенерировать (throw) исключительную ситуацию.

При делении на 0 система сообщает о возникновении ошибки и прекращает работу. Может потребоваться дополнительная обработка исключительной ситуации – например, определение номера оператора, вызвавшего ошибку.

Говорят, что код, предназначенный для работы с исключительной ситуацией, перехватывает (catch), или обрабатывает (handle) ее.

Перехват исключительной ситуации. Для перехвата исключительной ситуации в языке C++ предусмотрены **блоки try-catch** (try-catch blocks). Оператор, который может породить исключительную ситуацию, следует поместить в блок *try*. За этим блоком должны следовать один или несколько блоков *catch*. В каждом блоке *catch* должен быть указан тип исключительной ситуации, для перехвата которой он предназначен. С блоком *try* могут быть связаны несколько блоков *catch,* даже если отдельный оператор может порождать исключительные ситуации нескольких типов. Кроме того, блок *try* может содержать много операторов, каждый из которых может генерировать исключительную ситуацию. Общая синтаксическая конструкция блока *try* приведена ниже.

```
try
{
        оператор(ы);
}
```

Синтаксис блока catch выглядит так.

```
catch (КлассИсключительнойСитуации: идентификатор)
{
        операторы
}.
```

Для каждого типа исключительной ситуации нужно применять отдельный блок catch, предназначенный для ее обработки

Когда операторы, помещенные в блок *try,* порождают исключительную ситуацию, оставшаяся часть блока *try* игнорируется, а управление передается операторам, размещенным в блоке *catch,* соответствующем типу возникшей исключительной ситуации. Затем выполняются операторы блока *catch.* Выполнение программы возобновляется, начиная с точки, следующей за последним блоком *catch*. Если для порожденной исключительной ситуации не подходит ни один блок *catch,* программа завершается аварийно.

Обратите внимание, если исключительная ситуация генерируется в середине блока *try,* вызываются деструкторы всех локальных объектов этого блока. Это гарантирует освобождение всех ресурсов, захваченных блоком, даже если он не будет выполнен до конца.

Приведем пример программы обработки исключительной ситуации.

```
#include <iostream>
 using namespace std;
int main(int argc, char* argv[])
{
   int a, b;
   cout << "Введите числа a и b " << endl;
   cin>>a>>b;
   try
   {
```

```
        a=a/b;
        a=a+1;
    }
    catch (...)
         {
                cout << "Program mistake" << endl;
         }
    cout << "Good Day C++  !" << endl;
    return 0;
}
```

В программе обрабатывается деление на 0 и переполнение.

Генерирование исключительных ситуаций. Когда внутри функции обнаруживается ошибка, исключительную ситуацию можно сгенерировать с помощью оператора следующего вида.

Srow КлассИсключительнойСитуации (стро*ковыйАргумент) ;*

Здесь обозначение КлассИсключительнойСитуации относится к типу исключительной ситуации, которую необходимо сгенерировать, а запись стро*ковыйАргумент* означает аргумент конструктора этого класса, описывающий возникающую ошибку. При выполнении оператора throw оставшийся код функции не выполняется, а исключительная ситуация передается обратно в точку, из которой была вызвана функция.

В стандартной библиотеке C++ можно найти класс исключительной ситуации, удовлетворяющий потребностям программы. Однако программист может определять свой собственный класс исключительных ситуаций. При этом в качестве базового обычно используется класс исключительных ситуаций *exception,* или один из производных от него классов. Это обеспечивает возможности стандартизированной работы с исключительными ситуациями. В частности, все исключительные ситуации, предусмотренные в стандартной библиотеке языка C++, содержат функцию-член *what,* возвращающую сообщение, описывающее возникшую исключительную ситуацию. Если при создании своего собственного класса исключительных ситуаций в качестве базового применяется класс *exception,* нужно использовать пространство имен *std.*

Чтобы указать, какая исключительная ситуация будет генерироваться функцией, включите раздел *throw* в заголовок функции, как показано ниже.

```
void myMethod (int x)
throw(BadArgException, MyException)
if (x == MAX)
   throw BadArgException("BadArgException: причина") ;
// Какой-то код
throw MyException("MyException: причина") ; } //
Конец функции myMethod
```

Включение раздела *throw* в спецификацию функции гарантирует, что данная функция сможет генерировать только указанные исключительные ситуации. Попытка возбудить любую другую исключительную ситуацию приведет к аварийному завершению работы программы.

Если оператор в блоке *try* генерирует исключительную ситуацию, оставшаяся часть блока игнорируется, и управление передается блоку *catch,* предназначенному для обработки исключительных ситуаций этого типа. Затем выполняются операторы блока *catch.* После завершения этого блока выполнение программы возобновляется с точки, следующей за последним оператором блока *catch.* Если возникшая исключительная ситуация не может быть обработана ни одним боком *catch,* выполнение программы завершается аварийно.

Обратите внимание, что если исключительная ситуация возбуждается внутри блока *try,* вызываются деструкторы всех локальных объектов этого блока. Это позволяет гарантировать, что все ресурсы, задействованные в этом блоке, будут освобождены, даже если блок будет выполнен не полностью. Компилятор выбирает подходящий блок catch, перебирая их один за другим в порядке, указанном в программе. Подходящим считается блок *catch,* аргумент которого совпадает с возникшей исключительной ситуацией. Таким образом, разделы *catch* должны быть упорядочены, так чтобы первыми оказались блоки, предназначенные для обработки более узких исключительных ситуаций, а разделы, ориентированные на более общие типы, должны размещаться за ними.

Рассмотрим пример.

```
string str = "Sarah";
try
{
    str.substr(99, 1);
    // Здесь размещаются другие операторы
} // Конец блока try
        catch (exception e)
{
    cout << "Перехвачено что-то другое" << endl;
} // Конец блока catch
catch (out_of__range e)
{
      cout << "Перехвачена исключительная ситуация out_of_range" << endl;
```

} / / Конец блока catch

При компиляции этого фрагмента программы на экране появляется следующее предупреждение.

```
TestExceptionExample.cpp(11) : warning C4286:
'class std::out_of_range' : is caught by base class
( 'class exception') online 8
Linking...
```

Чтобы скомпилировать этот код без предупреждений, нужно поменять местами два раздела *catch*.

Программа, приведенная ниже, демонстрирует, что произойдет, если исключительная ситуация будет сгенерирована, но не обработана. Программа кодирует строку, выполняя простую подстановку. Каждая буква исходной строки заменяется буквой, расположенной в алфавите на три позиции ниже. Достигнув конца алфавита, мы переходим на его начало. Например, буква 'a' заменяется буквой 'd', буква Ъ' — буквой 'e', а буква 'x' — буквой 'a'. Поток управления при возникновении исключительной ситуации в этой программе показан на рис.

```
#include <iostream>
#include <string>
using namespace std;
void encodeChar(int i, string&     str)
{
        int base;
        if (islower(str[i]))
                base = int('a');
        else
                base = int('A');
        char newChar = (int(str[i]) - base +3) % 26 + base;
        str.replace(i, 1, 1, newChar);
} // Конец функции encodeChar
void encodeString(int numChar, string^ str)
{
        for (int i = numChar-1; i>=0; i--)
                encodeChar(i, str);
} //Конец функции encodeString
int mainO
{
        string strl = "Sarah";
        encodeString(99, strl);
        return 0;
} // Конец оператора main
```

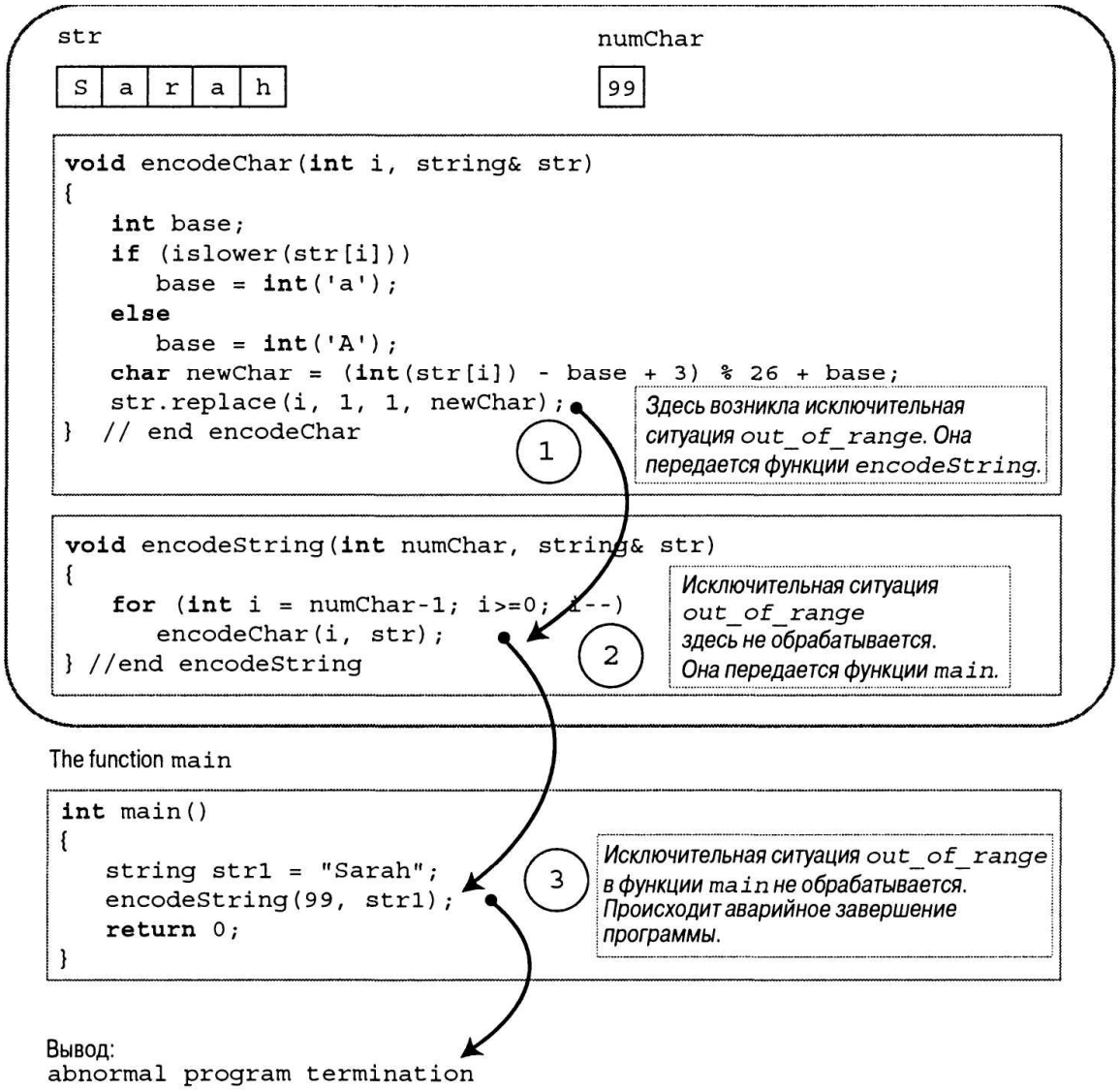


Рис.. Поток управления при необработанной исключительной ситуации

На самом деле метод *encodeChar* порождает исключительную ситуацию *out_of_range,* которая генерируется, когда происходит попытка доступа к 99-му символу строки str при вызове *str.replace (99,* I, 1, *newChar),* Поскольку эта исключительная ситуация в функции *encodeChar* не обрабатывается, выполнение функции прерывается, а исключительная ситуация возвращается в функцию *encodeString,* а именно: в точку вызова функции *encodeChar.*

Функция *encodeString* также не обрабатывает данную исключительную ситуацию, поэтому ее выполнение также прерывается, а исключительная ситуация передается в функцию *main.* Поскольку и там обработка исключительной ситуации не предусмотрена, происходит аварийное завершение программы.

9.4 Методические указания.

9.5 Варианты заданий.

В качестве системы объектов выбрать задание работы 8

10 Классы и объекты в С ++

Данную работу сделать по Буч

10.1 Цель.

Получить практические навыки реализации классов на С++.

10.2Основное содержание работы.

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

10.3Краткие теоретические сведения.

Доступность компонентов класса.

В рассмотренных ранее примерах классов компоненты классов являются общедоступными. В любом месте программы, где “видно” определение класса, можно получить доступ к компонентам объекта класса. Тем самым не выполняется основной принцип абстракции данных – инкапсуляция (сокрытие) данных внутри объекта. Для изменения видимости компонент в определении класса можно использовать спецификаторы доступа: **public, private, protected**.

Общедоступные (public) компоненты доступны в любой части программы. Они могут использоваться любой функцией как внутри данного класса, так и вне его. Доступ извне осуществляется через имя объекта:

имя_объекта.имя_члена_класса
ссылка_на_объект.имя_члена_класса
указатель_на_объект->имя_члена_класса

Собственные (private) компоненты локализованы в классе и не доступны извне. Они могут использоваться функциями – членами данного класса и функциями – “друзьями” того класса, в котором они описаны.

Защищенные (protected) компоненты доступны внутри класса и в производных классах.

Изменить статус доступа к компонентам класса можно и с помощью использования в определении класса ключевого слова **class**. В этом случае все компоненты класса по умолчанию являются собственными.

Пример.

```
class complex
{
 double re, im;                 // private по умолчанию
 public:
 double real(){return re;}
 double imag(){return im;}
 void set(double x,double y){re = x; im = y;}
};
```

Конструктор.

Недостатком рассмотренных ранее классов является отсутствие автоматической инициализации создаваемых объектов. Для каждого вновь создаваемого объекта необходимо было вызвать функцию типа set (как для класса complex) либо явным образом присваивать значения данным объекта. Однако для инициализации объектов класса в его определение можно явно включить специальную компонентную функцию, называемую **конструктором**. Формат определения конструктора следующий:

имя_класса(список_форм_параметров){операторы_тела_конструктора}

Имя этой компонентной функции по правилам языка C++ должно совпадать с именем класса. Такая функция автоматически вызывается при определении или размещении в памяти с помощью оператора new каждого объекта класса.

a. Пример.

```
complex(double re1 = 0.0,double im1 = 0.0){re = re1; im = im1;}
```

Конструктор выделяет память для объекта и инициализирует данные – члены класса.

Конструктор имеет ряд особенностей:

Для конструктора не определяется тип возвращаемого значения. Даже тип void не допустим.

Указатель на конструктор не может быть определен, и соответственно нельзя получить адрес конструктора.

Конструкторы не наследуются.

Конструкторы не могут быть описаны с ключевыми словами virtual, static, const, mutuable, valatile.

Конструктор всегда существует для любого класса, причем, если он не определен явно, он создается автоматически. По умолчанию создается конструктор без параметров и конструктор копирования. Если конструктор описан явно, то конструктор по умолчанию не создается. По умолчанию конструкторы создаются общедоступными (public).

Параметром конструктора не может быть его собственный класс, но может быть ссылка на него (T&). Без явного указания программиста конструктор всегда автоматически вызывается при определении (создании) объекта. В этом случае вызывается конструктор без параметров. Для явного вызова конструктора используются две формы:

имя_класса имя_объекта (фактические_параметры);

имя_класса (фактические_параметры);

Первая форма допускается только при не пустом списке фактических параметров. Она предусматривает вызов конструктора при определении нового объекта данного класса:

```
complex ss (5.9,0.15);
```

Вторая форма вызова приводит к созданию объекта без имени:

complex ss = complex (5.9,0.15);

Существуют два способа инициализации данных объекта с помощью конструктора. Ранее мы рассматривали первый способ, а именно, передача значений параметров в тело конструктора. Второй способ предусматривает применение списка инициализаторов данного класса. Этот список помещается между списком параметров и телом конструктора. Каждый иници-ализатор списка относится к конкретному компоненту и имеет вид:

имя_данного (выражение)

Примеры.

```
class CLASS_A
{
  int i; float e; char c;
public:
  CLASS_A(int ii,float ee,char cc) : i(8),e( i * ee + ii ),c(cc){}
. . .
};
```

Класс “символьная строка”.

```
#include <string.h>
#include <iostream.h>
class string
{
  char *ch; // указатель на текстовую строку
  int len;      // длина текстовой строки
public:
  // конструкторы
  // создает объект – пустая строка
  string(int N = 80): len(0){ch = new char[N+1]; ch[0] = '\0';}
  // создает объект по заданной строке
  string(const char *arch){len = strlen(arch);
      ch = new char[len+1];
      strcpy(ch,arch);}
  // компоненты-функции
  // возвращает ссылку на длину строки
  int& len_str(void){return len;}
  // возвращает указатель на строку
  char *str(void){return ch;}
. . .};
```

Здесь у класса string два конструктора – перегружаемые функции.

По умолчанию создается также конструктор копирования вида T::T(const T&), где T – имя класса. Конструктор копирования вызывается всякий раз, когда выполняется копирование объектов, принадлежащих классу. В частности он вызывается:

а) когда объект передается функции по значению;

б) при построении временного объекта как возвращаемого значения функции;

в) при использовании объекта для инициализации другого объекта.

Если класс не содержит явным образом определенного конструктора копирования, то при возникновении одной из этих трех ситуаций производится побитовое копирование объекта. Побитовое копирование не во всех случаях является адекватным. Именно для таких случаев и необходимо определить собственный конструктор копирования. Например, в классе string:

```
string(const string& st)
{len=strlen(st.len);
ch=new char[len+1];
strcpy(ch,st.ch);  }
```

Можно создавать массив объектов, однако при этом соответствующий класс должен иметь конструктор по умолчанию (без параметров).

Массив объектов может инициализироваться либо автоматически конструктором по умолчанию, либо явным присваиванием значений каждому элементу массива.

```
class demo{
int x;
public:
demo(){x=0;}
demo(int i){x=i;}
};
void main(){
class demo a[20]; //вызов конструктора без параметров(по умолчанию)
class demo b[2]={demo(10),demo(100)};//явное присваивание
```

Деструктор.

Динамическое выделение памяти для объекта создает необходимость освобождения этой памяти при уничтожении объекта. Например, если объект формируется как локальный внутри блока, то целесообразно, чтобы при выходе из блока, когда уже объект перестает существовать, выделенная для него память была возвращена. Желательно, чтобы освобождение памяти происходило автоматически. Такую возможность обеспечивает специальный компонент класса – **деструктор** класса. Его формат:

~имя_класса(){операторы_тела_деструктора}

Имя деструктора совпадает с именем его класса, но предваряется символом “~” (тильда).

Деструктор не имеет параметров и возвращаемого значения. Вызов деструктора выполняется не явно (автоматически), как только объект класса уничтожается.

Например, при выходе за область определения или при вызове оператора delete для указателя на объект.

string *p=new string “строка”);

delete p;

Если в классе деструктор не определен явно, то компилятор генерирует деструктор по умолчанию, который просто освобождает память, занятую данными объекта. В тех случаях, когда требуется выполнить освобождение и других объектов памяти, например область, на которую указывает ch в объекте string, необходимо определить деструктор явно: ~string(){delete []ch;}

Так же, как и для конструктора, не может быть определен указатель на деструктор.

Указатели на компоненты-функции.

Можно определить указатель на компоненты-функции.

тип_возвр_значения(имя_класса::*имя_указателя_на_функцию) (специф_параметров_функции);

Пример.

```
double(complex : :*ptcom)(); // Определение указателя
ptcom = &complex : : real; // Настройка указателя
// Теперь для объекта  A можно вызвать его функцию
complex A(5.2,2.7);
cout<<(A.*ptcom)();
```

Можно определить также тип указателя на функцию

typedef double&(complex::*PF)();

а затем определить и сам указатель

PF ptcom=&complex::real;

10.4 Порядок выполнения работы.

1. Определить пользовательский класс в соответствии с вариантом задания (смотри приложение).

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.

5. Определить указатель на компоненту-функцию.

6. Определить указатель на экземпляр класса.

7. Написать демонстрационную программу, в которой создаются и разрушаются объекты пользовательского класса и каждый вызов

конструктора и деструктора сопровождается выдачей соответствующего сообщения (какой объект какой конструктор или деструктор вызвал).

8. Показать в программе использование указателя на объект и указателя на компоненту-функцию.

10.5Методические указания.

1. Пример определения класса.

```
const int LNAME=25;
class STUDENT{
char name[LNAME];              // имя
int age;                       // возраст
float grade;                   // рейтинг
public:
STUDENT();                     // конструктор без параметров
STUDENT(char*,int,float);      // конструктор с параметрами
STUDENT(const STUDENT&); // конструктор копирования
~STUDENT();
char * GetName() ;
int GetAge() const;
float GetGrade() const;
void SetName(char*);
void SetAge(int);
void SetGrade(float);
void Set(char*,int,float);
void Show(); };
```

Более профессионально определение поля **name** типа указатель: char* name. Однако в этом случае реализация компонентов-функций усложняется.

2. Пример реализации конструктора с выдачей сообщения.

```
STUDENT::STUDENT(char*NAME,int AGE,float GRADE)
{
strcpy(name,NAME); age=AGE; grade=GRADE;
cout<< \nКонструктор с параметрами вызван для объекта  <<this<<endl;
}
```

3. Следует предусмотреть в программе все возможные способы вызова конструктора копирования. Напоминаем, что конструктор копирования вызывается:

а) при использовании объекта для инициализации другого объекта

Пример.

```
STUDENT a(“Иванов”,19,50), b=a;
```

б) когда объект передается функции по значению

Пример.

```
void View(STUDENT a){a.Show;}
```

в) при построении временного объекта как возвращаемого значения функции

Пример.

```
STUDENT NoName(STUDENT & student)
{STUDENT temp(student);
temp.SetName("NoName");
return temp;}

STUDENT c=NoName(a);
```

4. В программе необходимо предусмотреть размещение объектов как в статической, так и в динамической памяти, а также создание массивов объектов.

Примеры.

а) массив студентов размещается в статической памяти

```
STUDENT gruppa[3];
gruppa[0].Set("Иванов",19,50);
```

и т.д.

или

```
STUDENT gruppa[3]={STUDENT("Иванов",19,50),
                   STUDENT("Петрова",18,25.5),
                   STUDENT("Сидоров",18,45.5)};
```

б) массив студентов размещается в динамической памяти

```
STUDENT *p;
p=new STUDENT [3];
p-> Set("Иванов",19,50);
```

и т.д.

5. Пример использования указателя на компонентную функцию

```
void (STUDENT::*pf)();
pf=&STUDENT::Show;
(p[1].*pf)();
```

6. Программа использует три файла:

заголовочный h-файл с определением класса,
cpp-файл с реализацией класса,
cpp-файл демонстрационной программы.

Для предотвращения многократного включения файла-заголовка следует использовать директивы препроцессора

```
#ifndef STUDENTH
#define STUDENTH
// модуль STUDENT.H
...
#endif
```

10.6 Содержание отчета.

1. Титульный лист: название дисциплины; номер и наименование работы; фамилия, имя, отчество студента; дата выполнения.

2. Постановка задачи. Следует дать конкретную постановку, т.е. указать, какой класс должен быть реализован, какие должны быть в нем конструкторы, компоненты-функции и т.д.

3. Определение пользовательского класса с комментариями.

4. Реализация конструкторов и деструктора.

5. Фрагмент программы, показывающий использование указателя на объект и указателя на функцию с объяснением.

6. Листинг основной программы, в котором должно быть указано, в каком месте и какой конструктор или деструктор вызываются.

10.7Варианты заданий.

1)Типография. Оформление заказов на печать книжной продукции. Подготовка производства. Оформление электронного образа книги. Печать и сборка тома. Персонал, специалисты, места работы и зарплата.

2)Проектирование нефтяного оборудования. Хранение сведений о проектах и проектировщиках. Формирование проектных документов. Персонал, специалисты, места работы и зарплата.

3)Добыча нефти. Хранение карт бурения скважин. Сведения о скважинах и оборудовании. Данные о добыче и качестве нефти. Специалисты, места работы и зарплата.

4)Хранение нефти. Характеристики и состояние нефтяных танков. Сведения о поступлении и марке топлива. Сведения об отгрузке топлива. Данные о поставщиках, потребителях и расчетах за топливо. Специалисты, места работы и зарплата.

5)Бензозаправочная станция. Список поставщиков, расчеты с поставщиками. Получение и продажа топлива с каждой колонки. Персонал, график работы и расчеты с персоналом.

6)Проектирование зданий. Хранение сведений о проектах и проектировщиках. Формирование проектных документов. Персонал, специалисты, места работы и зарплата.

7)Строительство домов. Хранение сведений о деталях дома и о строительной технике. Поступление деталей, состояние строительства. Персонал, график работы и расчеты с персоналом.

8)Продажа квартир. Списки квартир, сведения о продавцах, покупателях и риэлтерах. Сложные структуры продаж. Договоры, оформление и хранение. Персонал, график работы и расчеты с персоналом.

9)Высшее учебное заведение. Сопровождение процесса обучения студентов. Списки студентов, сведения об оплате за обучение и стипендиях. Сессия, сведения об успеваемости. Предметы, преподаватели, расписание.

10)Школа. Сопровождение процесса обучения. Административная структура школы. Списки классов, сведения об учениках и родителях.

Расписание уроков. Сведения об успеваемости. Предметы, преподаватели, расписание. Учителя, график работы.

11)Строительство больниц. Хранение сведений о деталях дома и о строительной технике. Сведения о палатах, специальном оборудовании. Поступление деталей, состояние строительства. Персонал, график работы и расчеты с персоналом.

12)Больница. Отделения и палаты, врачи, специальности врачей. Графики дежурства и графики приема. Списки больных. Истории болезней.

13)Амбулатория. Кабинеты, врачи – специалисты. Запись больных на прием, амбулаторные карты. Лечение. Персонал, график работы.

14)Лаборатория больницы. Врачи, анализы, направления на анализы. Больные, результаты анализов.

15)Аптека. Списки лекарств. Поставщики и производители лекарств. Рецепты, заказы на изготовление, цены на лекарства и контроль продажи лекарств. Персонал, график работы и расчеты с персоналом.

16)Производство лекарств. Списки видов сырья и оборудования. Производители и поставщики сырья и оборудования. Заказы на поставки лекарств. Оборудование и сырье для производства лекарств. Контроль расхода сырья и учет производства лекарств.

17)Аптечный склад. Хранение лекарств. Списки видов лекарств и оборудования для хранения лекарств. Производители и поставщики лекарств. Оптовые покупатели лекарств. Заказы на поставки лекарств. Персонал, график работы.

18)Проектирование автомобилей. Списки: Номер проекта, марка автомобиля, тип кузова, тип шасси и т.д. Проектировщики, специальности проектировщиков по типу частей автомобиля. Списки чертежей, изготовители деталей, графики изготовления. Испытания и результаты испытаний по номеру проекта проектируемого автомобиля.

19)Производство автомобилей. Списки поставщиков запчастей и оборудования для сборки и изготовления автомобиля. Заказы на поставку автомобилей и соответствующие заказы на получение комплектующих. Конвейер и рабочие конвейера по специальностям. График работы, отработанные часы и расчеты с персоналом.

20)Стоянка автомобилей. Списки мест на стоянке и постоянных клиентов. Заказы на постановку автомобилей. Списки владельцев, поставивших машины на стоянку и расчеты с клиентами. График работы, отработанные часы и расчеты с персоналом.

21)Продажа автомобилей. Списки поставщиков и цены на автомобили. Заказы на покупку автомобилей и поставки от производителя. Графики поступления автомобилей от поставщиков и графики продаж. Торговля запасными частями. Списки запчастей и производителей, состояние склада, движение запчастей со склада покупателям и поступление от производителей.

22)Обучение езде на автомобиле. Сопровождение процесса обучения. Административная структура школы. Списки групп, сведения об учениках и

местах работы. Расписание теоретических и практических занятий. Сведения об успеваемости. Предметы, преподаватели, расписание. Инструкторы, график работы. Расписание экзаменов и сведения о сдаче экзаменов.

24)Ателье. Административная структура учреждения. Одежда, фасоны и структурные составляющие одежды. Фасоны. Ткани и аксессуары. Списки заказов и графики выполнения заказов. График работы, результаты работы и расчеты с персоналом.

25)Продажа компьютеров. Списки поставщиков и цены на компьютеры. Заказы на покупку компьютеров, структура заказанного компьютера и поставки с центральных складов. Графики поступления комплектующих от поставщиков и графики продаж. Торговля комплектующими. Списки комплектующих и производителей, состояние склада, движение комплектующих со склада покупателям и поступление от производителей.

26)Продажа хозяйственных товаров. Административная структура учреждения. Списки поставщиков и цены на товары. Списки ходовых и обязательных товаров. Графики поступления товаров от поставщиков и графики продаж. Торговля, состояние склада, движение товара со склада в отделы и поступление от производителей.

27)Универмаг. Административная структура учреждения. Списки поставщиков и цены на товары. Списки ходовых и обязательных товаров. Графики поступления товаров от поставщиков и графики продаж. Торговля. состояние склада, движение товара со склада в отделы и поступление от производителей.

28)Универсам Административная структура учреждения. Списки поставщиков и цены на товары. Списки ходовых и обязательных товаров. Графики поступления товаров от поставщиков и графики продаж. Торговля, состояние склада, движение товара со склада в отделы и поступление от производителей.

29)Телефонная станция. Административная структура учреждения. Перечень и цены услуг. Типы договоров с абонентами. Списки абонентов с атрибутами: номер телефона, наличие блокиратора, Ф.И.О., адрес, договор, счет и его состояние. Расчеты с абонентами за различные виды услуг, квитанции, формирование квитанций, учет оплаты.

30)Спорт, Олимпийские игры. Виды спорта, рекорды. Страна, команда, фамилии титулы и специальности спортсменов. Расписание соревнований – город, дата, время, вид спорта, фамилии спортсменов и сопровождающих лиц. Результаты соревнований. Журналисты и допуск на стадион.

31)Географический атлас. Континенты, регионы, страны, города, реки и проч. Население – численность, языки. Фауна, флора, полезные ископаемые, промышленность и сельское хозяйство.

2. Список литературы

Основная

1. 1. 1. А.М. Ноткин ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ С++ Пермский государственный технический университет Кафедра автоматизированных систем управления 2001г.
2. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на С++. Второе издание. – М.: Бином, 1998.
3. 2. Паппас К., Мюррей У. Visual C++6: Руководство разработчика. Киев: BHV, 2000.
4. 3. Подбельский В.В. Язык С++ – М.: Финансы и статистика, 1996.
5. 4.Страуструп Б. Язык программирования С++. Третье издание, М.: Бином, 1999.

Дополнительная

6. Аммераль Л. STL для программистов на С++. – М., ДМК, 1999.
7. Грегори К. Использование Visual C++6. – М., Вильямс, 1999.
8. Киммел П. Borland C++5. – СПб.: BHV, 1997.
9. Крейг Арнуш. Borland C++: освой самостоятельно – М.: Бином, 1997.
10. Лейнекер Р. Энциклопедия Visual C++6. – Спб, Питер, 1999.
11. Луис Д. С и С++. Справочник. – М: Бином, 1997.
12. Пол Айра. Объектно-ориентированное программирование на С++. Второе издание. – М.: Бином, 1999.
13. Секунов Н.Ю. Самоучитель Visual C++6. – Спб, BHV, 1999.
14. Скляров В.А. Язык С++ и ООП. – Минск: Вышэйшая школа, 1997.
15. Фейсон Т. Объектно-ориентированное программирование на С++ 4.5. – Киев: Диалектика, 1996.
16. Шилдт Г. Самоучитель С++. Второе издание. – СПб.: BHV, 1998.
17. Шилдт Г. Теория и практика С++. – СПб.: BHV, 1996.
18. Элджер Дж. С++: библиотека программиста – СПб: Питер, 1999.