Министерство образования Российской Федерации
Пермский государственный технический университет
Кафедра автоматизированных систем управления

А.М. Ноткин

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ С++

Утверждено Редакционно-издательским советом университета в качестве учебного пособия

Пермь 2001

ОГЛАВЛЕНИЕ

Предисловие…….4

Лабораторная работа № 1. **Классы и объекты в C++**……...……...5

Лабораторная работа № 2. **Наследование и виртуальные функции**……...……..16

Лабораторная работа № 3. **Иерархия объектов и группа. Итераторы**..……..…...……..25

Лабораторная работа № 4. **Обработка событий**…….……34

Лабораторная работа № 5. **Перегрузка операций**…….…….46

Лабораторная работа № 6. **Шаблоны функций и классов**……...…59

Лабораторная работа № 7. **Потоковые классы**……...……...67

Лабораторная работа № 8. **Стандартная библиотека шаблонов**…79

Список литературы……...…….…..92

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 восьми лабораторных работ. Первые четыре работы связаны с базовыми понятиями C++, такими как объекты и классы, наследование, полиморфизм и виртуальные функции, обработка событий. Последние четыре работы посвящены профессиональному программированию на C++ и охватывают разделы профессионального программирования, такие как перегрузка операций, шаблоны, потоковые классы и стандартная библиотека шаблонов.

Данное пособие дополняет конспект лекций того же автора “Алгоритмические языки и технология программирования. Часть 3: Объектно-ориентированное программирование на C++”.

Лабораторные работы № 1 – № 4 выполняются в среде Turbo C++ 3.0; № 5 – № 8 – в среде Borland C++ 5.02.

Лабораторная работа № 1

КЛАССЫ И ОБЪЕКТЫ В C++

Цель. Получить практические навыки реализации классов на C++.

Основное содержание работы.

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

Краткие теоретические сведения.

Класс.

Класс – фундаментальное понятие C++, он лежит в основе многих свойств C++. Класс предоставляет механизм для создания объектов. В классе отражены важнейшие концепции объектно-ориентированного программирования: инкапсуляция, наследование, полиморфизм.

С точки зрения синтаксиса, класс в C++ – это структурированный тип, образованный на основе уже существующих типов.

В этом смысле класс является расширением понятия структуры. В простейшем случае класс можно определить с помощью конструкции:

тип_класса имя_класса{список_членов_класса};

где

тип_класса – одно из служебных слов **class, struct, union**;

имя_класса – идентификатор;

список_членов_класса – определения и описания типизированных данных и принадлежащих классу функций.

Функции – это методы класса, определяющие операции над объектом.

Данные – это поля объекта, образующие его структуру. Значения полей определяет состояние объекта.

Примеры.

```
struct date                    // дата
{int month,day,year;           // поля: месяц, день, год
  void set(int,int,int);       // метод – установить дату
  void get(int*,int*,int*);    // метод – получить дату
  void next();                 // метод – установить следующую дату
  void print();                // метод – вывести дату
};
struct class complex           // комплексное число
{double re,im;
  double real(){return(re);}
```

```
  double imag(){return(im);}
  void set(double x,double y){re = x; im = y;}
  void print(){cout<<“re = “<<re; cout<<“im = “<<im;}
};
```

Для описания объекта класса (экземпляра класса) используется конструкция

имя_класса имя_объекта;

```
date today,my_birthday;
date *point = &today;              // указатель на объект типа date
date clim[30];                     // массив объектов
date &name = my_birthday;          // ссылка на объект
```

В определяемые объекты входят данные, соответствующие членам – данным класса. Функции – члены класса позволяют обрабатывать данные конкретных объектов класса. Обращаться к данным объекта и вызывать функции для объекта можно двумя способами. Первый с помощью “квалифицированных” имен:

имя_объекта. имя_данного
имя_объекта. имя_функции

Например:

```
complex x1,x2;
x1.re = 1.24;
x1.im = 2.3;
x2.set(5.1,1.7);
x1.print();
```

Второй способ доступа использует указатель на объект

указатель_на_объект–>имя_компонента

```
complex *point = &x1;  //  или  point = new complex;
point –>re = 1.24;
point –>im = 2.3;
point –>print();
```

Доступность компонентов класса.

В рассмотренных ранее примерах классов компоненты классов являются общедоступными. В любом месте программы, где “видно” опреде-

ление 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_парамет ров){операторы_тела_конструкто ра}

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.

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 помеща-

ется между списком параметров и телом конструктора. Каждый инициализатор 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;
```

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.

Методические 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
```

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.

Приложение. Варианты заданий.

Описания членов - данных пользовательских классов

1. СТУДЕНТ
имя – char*
курс – int
пол – int(bool)

4. ИЗДЕЛИЕ
имя – char*
шифр – char*
количество – int

7. АДРЕС
имя – char*
улица – char*
номер дома – int

10. ЦЕХ
имя – char*
начальник – char*
количество
работающих – int

13. СТРАНА
имя – char*
форма
правления – char*
площать – float

2. СЛУЖАЩИЙ
имя – char*
возраст – int
рабочий стаж – int

5. БИБЛИОТЕКА
имя – char*
автор – char*
стоимость – float

8. ТОВАР
имя – char*
количество – int
стоимость – float

11. ПЕРСОНА
имя – char*
возраст – int
пол – int(bool)

14. ЖИВОТНОЕ
имя – char*
класс – char*
средний вес – int

3. КАДРЫ
имя – char*
номер цеха – int
разряд – int

6. ЭКЗАМЕН
имя студента – char*
дата – int
оценка – int

9. КВИТАНЦИЯ
номер – int
дата – int
сумма – float

12. АВТОМОБИЛЬ
марка – char*

мощность – int тип – char*
стоимось – float

15. КОРАБЛЬ
имя – char*
водоизмещение – int

Лабораторная работа № 2
НАСЛЕДОВАНИЕ И ВИРТУАЛЬНЫЕ ФУНКЦИИ

Цель. Получить практические навыки создания иерархии классов и использования статических компонентов класса.

Основное содержание работы.

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

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

Но 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.h>
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.*

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).

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:

static person* begin; // указатель на начало списка

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.

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.

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.

<u>Приложение.Варианты заданий.</u>

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.

Лабораторная работа № 3
ИЕРАРХИЯ ОБЪЕКТОВ И ГРУППА. ИТЕРАТОРЫ

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

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. Примерами таких окон являются объекты классов, порожденных от абстрактного класса *TGroup(TDeskTop, TWindow, TDialog*) в иерархии классов библиотеки **Turbo Vision**, и объекты классов, порожденных от *TWindowObject* в иерархии классов библиотеки **OWL.**

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 класса *TCollection* библиотеки **Turbo Vision** и 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.Такой способ используется при реализации групп в **Turbo Vision.**

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. Эти средства реализованы в последних системах Borland C++ (версий 4.0 и выше).

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.h>
#include<typeinfo.h>
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

Порядок выполнения работы.

1. Дополнить иерархию классов лабораторной работы № 2 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.

Методические указания.

1. Класс-группа должен соответствовать иерархии классов лабораторной работы № 2, т.е. объекты этих классов должны входить в группу. Например, для варианта 1 может быть предложена следующая иерархия классов:

Tobject (абстр. класс)

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 связанного 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) ->Get-
Name());
}
```

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

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 № 2).
6. Реализация для одного не абстрактного класса-группы всех методов.
7. Реализация итератора.
8. Реализация передаваемой итератору функции.
9. Листинг демонстрационной программы.

Приложение.Варианты запросов.

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).

Лабораторная работа № 4
ОБРАБОТКА СОБЫТИЙ

Цель. Получить практические навыки разработки объектно-ориентированной программы, управляемой событиями.

Основное содержание работы.

Написать интерактивную программу, выполняющую команды, вводимые пользователем с клавиатуры.

Краткие теоретические сведения.

Объектно-ориентированная программа как программа, управляемая событиями.

При использовании ООП все объекты являются в некотором смысле обособленными друг от друга, и возникают определенные трудности в передаче информации от объекта к объекту. В ООП для передачи информации между объектами используется механизм обработки событий.

События лучше всего представить себе как пакеты информации, которыми обмениваются объекты и которые создаются объектно-ориентированной средой в ответ на те или иные действия пользователя. Нажатие на клавишу или манипуляция мышью порождают событие, которое передается по цепочке объектов, пока не найдется объект, знающий, как обрабатывать это событие. Для того чтобы событие могло передаваться от объекта к объекту, все объекты программы должны быть объединены в группу. Отсюда следует, что прикладная программа должна

быть объектом-группой, в которую должны быть включены все объекты, используемые в программе.

Таким образом, объектно-ориентированная программа – это программа, управляемая событиями. События сами по себе не производят никаких действий в программе, но в ответ на событие могут создаваться новые объекты, модифицироваться или уничтожаться существующие, что и приводит к изменению состояния программы. Иными словами все действия по обработке данных реализуются объектами, а события лишь управляют их работой.

Принцип независимости обработки от процесса создания объектов приводит к появлению двух параллельных процессов в рамках одной программы: процесса создания объектов и процесса обработки данных.

Это означает, что действия по созданию, например, интерактивных элементов программы (окон, меню и пр.) можно осуществлять, не заботясь о действиях пользователя, которые будут связаны с ними.

И наоборот, мы можем разрабатывать части программы, ответственные за обработку действий пользователя, не связывая эти части с созданием нужных интерактивных элементов.

Событие.

Событие с точки зрения языка C++ – это объект, отдельные поля которого характеризуют те или иные свойства передаваемой информации, например:

```
struct TEvent
{int what
union{
        MouseEventType  mouse;
        KeyDownEvent  keyDown;
      MessageEvent  message;
       };
```

Объект *TEvent* состоит из двух частей. Первая (*what*) задает тип события, определяющий источник данного события. Вторая задает информацию, передаваемую с событием. Для разных типов событий содержание информации различно. Поле *what* может принимать следующие значения:

evNothing – это пустое событие, которое означает, что ничего делать не надо. Полю *what* присваивается значение *evNothing*, когда событие обработано каким-либо объектом.

evMouse – событие от мыши.

Событие от мыши может иметь, например, такую структуру:

```
struct MouseEventType
  {int buttons;
   int doubleClick;
   TPoint where;
  };
```

где *buttons* указывает нажатую клавишу;

doubleClick указывает был ли двойной щелчок;

where указывает координаты мыши.

evKeyDown – событие от клавиатуры.

Событие от клавиатуры может иметь, например, такую структуру:

```
struct KeyDownEvent
 {union{int keyCode;
        union{char charCode;
              char scanCode;
               };
        };
};
```

evMessage – событие-сообщение от объекта.

Для события от объекта (*evMessage*) задаются два параметра :

command – код команды, которую необходимо выполнить при появлении данного события;

infoPtr – передоваемая с событием (сообщение) информация.

```
struct MessageEvent
{int command;
 void  infoPtr;
};
```

Методы обработки событий.

Следующие методы необходимы для организации обработки событий (названия произвольны).

GetEvent – формирование события;

Execute реализует главный цикл обработки событий. Он постоянно получает событие путем вызова GetEvent и обрабатывает их с помощью HandleEvent. Этот цикл завершается, когда поступит событие «конец».

HandleEvent – обработчик событий. Обрабатывает каждое событие нужным для него образом. Если объект должен обрабатывать определенное событие (сообщение), то его метод HandleEvent должен распознавать это событие и реагировать на него должным образом. Событие может распознаваться, например, по коду команды (поле command).

ClearEvent очищает событие, когда оно обработано, чтобы оно не обрабатывалось далее.

Обработчик событий (метод HandleEvent).

Получив событие (структуру типа TEvent), обработчик событий для класса TDerivedClass обрабатывает его по следующей схеме:

```
void TDerivedClass::HandleEvent(TEvent& event)
  { //Вызов обработчика событий базового класса
    TBaseClass::handleEvent( event );
      if( event.what == evCommand ) // Если обработчик событий базового
класса                                  //  событие не обработал
    {
    switch( event.message.command )
       {
       case cmCommand1:
          // Обработка команды cmCommand1
          // Очистка события
          ClearEvent( event );
           break;
    case cmCommand2:
        // Обработка команды cmCommand2
        ClearEvent( event );
         break;
      …
    case cmCommandN:
        // Обработка команды cmCommandN
        ClearEvent( event );
         break;
    default: // событие  не обработано
     break;
     }
  };
}
```

Обработчик событий группы вначале обрабатывает команды группы, а затем, если событие не обработано, передает его своим элементам, вызывая их обработчики событий.

```
void TGroup::HandleEvent(TEvent& event)
  { if( event.what == evCommand )
     {switch( event.message.command )
     // обработка событий объекта-группы
```

```
      default: // событие  не  группой  обработано
     //получить доступ к первому элементу группы
     while((event.what != evNothing)!!( /* просмотрены не все элементы */)
     {
     //вызвать HandleEvent текущего элемента
     //перейти к следующему элементу группы
     }
      break;
   }
}
```

Метод ClearEvent-очистка события.

ClearEvent очищает событие, присваивая полю event.What значение evNothing.

Главный цикл обработки событий (метод Execute)

Главный цикл обработки событий реализуется в методе Execute главной группы-объекта “прикладная программа” по следующей схеме:

```
int TMyApp::Execute()
{do{endState=0;

     GetEvent(event); //получить событие
     HandleEvent(event); //обработать событие
      if(event.what!=evNothing) //событие осталось не обработано
      EventError(event);
       }
  while(!Valid());
return endState;
}
```

Метод HandleEvent программы обрабатывает событие “конец работы”, вызывая метод EndExec. EndExec изменяет значение private – переменной EndState. Значение этой переменной проверяет метод–функция Valid, возвращающая значение true, если “конец работы”. Такой несколько сложный способ завершения работы программы связан с тем, что в активном состоянии могут находиться несколько элементов группы. Тогда метод Valid группы, вызывая методы Valid своих подэлементов, возвратит true, если все они возвратят true. Это гарантирует, что программа завершит свою работу, когда завершат работу все ее элементы.

Если событие осталось не обработанным, то вызывается метод EventError, которая в простейшем случае может просто выдать сообщение.

Пример обработки событий.

Рассмотрим простейший калькулятор, воспринимающий команды в командной строке. Здесь приводится упрощенный вариант. Вариант, по схеме которого следует выполнить лабораторную работу, приведен в **приложении.**

Формат команды:

знак параметр

Знаки +, –, *, /, =, ?, q

Параметр – целое число

Константы-команды:

```
const  int evNothing = 0;
const  int evMessage = 100;
const  int cmSet = 1;     //занести число
const  int cmGet = 2;    //посмотреть значение
const  int cmAdd = 3;   //добавить
и т.д.
const  int cmQuit = 101; //выход
```

Класс-событие

```
struct TEvent
{int what
union{
        int evNothing;
        union{int command;
                int a;}
       }
}
```

Объект-калькулятор, работающий с целыми числами.

```
class TInt{
  int EndState;
  public
  int x;
  Int(int x1);
  virtual ~Int();
  virtual void GetEvent (TEvent &event);
  virtual int Exicute();
  virtual void HandleEvent (TEvent& event);
  virtual void ClearEvent (TEvent& event);
 int Valid();
 void EndExec();
 int GetX();
 void SetX (int newX);
```

```
 void AddY (int Y);
…
};
```

Рассмотрим возможную реализацию основных методов.

```
void TInit::GetEvent(TEvent &event)
{char* OpInt = “+-*/=?q”; //строка содержит коды операций
char s[20];
char code;
cout<<‘>’;
cin>>s; code = s[1];
if(Test(char code,char*OpInt) // Функции Test проверяет, входит ли символ                                  // code в строку OpInt

{event.what = evMessage;
        swith(code)
       {case ‘+’: event.command=cmAdd;
         break;
         …
         case‘q’: event.command = cmQuit;
         break;
         }
  //выделить второй параметр, перевести его в тип int и присвоить полю A
   };
 else event.what= evNothing
};

int TMyApp::Execute()
{do{endState=0;
     GetEvent(event); //получить событие
     HandleEvent(event); //обработать событие
      if(event.what!=evNothing) //событие осталось не обработано
 while(!Valid());
 return endState;
}

void TInt::HandleEvent(TEvent& event)
   {
    if( event.what == evMessage)
```

```
        {
        switch( event.message.command )
           {
           case cmAdd:AddY(event.A);
                 ClearEvent( event );
                 break;
                 …
           case cmQuit:EndExec();
        ClearEvent( event );
                  break;
           };
         };
    }

int TInt::Valid();
{ if (EndState == 0) return 0;
  else return 1;
}

void TInt::ClearEvent(TEvent& event)
{
  Event. what:= evNothing;
}

void TInt::EndExec()
{
  EndState= 1;
}

void TInt::AddY(int Y)
{
 x+=Y;
и т.д.
void main()
{
TInt  MyApp;
MyApp.Execute();
}
```

Порядок выполнения работы.

1. Разобрать пример, представленный в приложении. Ответить на следующие вопросы:

а) какова здесь иерархия классов?

б) какова здесь иерархия объектов?

в) как КАЛЬКУЛЯТОРУ передаются аргументы операции? где они хранятся? Каким образом получают к ним доступ устройства СЛОЖЕНИЯ, ВЫЧИТАНИЯ и т.д. ?

г) как обрабатываются события группой?

д) каковы все маршруты события TEvent?

е) как выполняются HandleEvent всех классов?

2. Выбрать группу объектов, которые будут обрабатывать события (это не могут быть объекты, приведенные в приложении).

3.Для выбранной группы объектов определить перечень операций, которые должны выполняться по командам пользователя.

4.Определить вид командной строки <код_операции><параметры>. Решить вопросы:

как кодируются операции? какие передаются параметры?

5. Определить иерархию объектов. Если необходимо, добавить новые объекты (группы объектов).

6. Определить иерархию классов. Если необходимо, добавить новые классы.

7. Определить, какой объект в программе играет роль приложения. В случае необходимости добавить в иерархию классов класс ***TApp.*** Решить, в каком классе будет метод ***Execute***, организующий главный цикл обработки событий.

8. Определить и реализовать необходимые для обработки событий методы.

9. Написать основную функцию (main).

Методические указания.

1. В качестве группы, для которой организуется обработка событий, выбрать группу из лабораторной работы № 3.

2. Количество различных обрабатываемых команд должно быть не менее 5.

3. Определение классов поместить в файл *.h. Определение функций-членов класса поместить в файл *.cpp.

4. Для констант, связанных с командами, использовать мнемонические имена *cmXXXX.*

Содержание отчета.

1. Титульный лист.

2. Постановка задачи.

3. Схема иерархии классов.

4. Схема иерархии объектов.

5. Описание маршрута, который проходит событие *TEvent* от формирования до очистки.

6. Определения классов.

7. Реализация методов обработки событий *GetEvent, Exicute, EndExec, Valid*.

8. Реализация всех методов (для всех классов) *HandleEvent*.

9. Листинг функции main().

Приложение.

Объект КАЛЬКУЛЯТОР выполняет сложение, вычитание, умножение, деление вещественных чисел.

Иерархия объектов.

В примере приведены определения основных классов и типов и реализации только некоторых компонентных функций.

```
class TShema;
class TObject //абстрактый класс - стоит во главе иерархии классов
{protected:
TShema* owner;
public:
TObject();
~TObject();
virtual void HandleEvent(TEvent&);
virtual void ClearEvent(TEvent&);
};
class TShema::public TObject // абстрактная группа
{protected:
TItem* last;
public:
TShema();
~TShema();
virtual void Insert(TObject*);
virtual void HandleEvent(TEvent&);
};
```

```
class TDevice: public TShema //   абстрактное                устройство управления
{protected:
int EndState;
public:
virtual void GetEvent(TEvent&);
virtual void Execute();
virtual int Valid();
virtual void EndExec();
};
class TRec: public TObject/ устройство для хранения данных-регистр
{protected:
float x;
public:
TReg();
~TReg();
float GetX();
void SetX(float&);
};
class TCalc : public TDevice //калькулятор
{pritected:
TReg* sum; // указатель на сумматор
TReg* reg; // указатель на регистр
public:
TCalc();
void HamdleEvent(TEvent&);
void GetEvent(TEvent&);
void Execute();
float GetSum(); // получить значение сумматора
void PutSum(float); //занести  число в сумматор
voit Help();
};
class TAdd : public TObject // схема сложения
{public:
void HandleEvent(TEvent&);
void Add();
};
TObject::TObject()
{owner=0;}
TShema::TShema()
{last=0;}
```

```
TCalc::TCalc()
{TObject* r;
sum=new TReg;
reg=new TReg;
r=new TAdd;
Insert (sum);
// и так далее для всех схем
};
TCalc::HandleEvent(TEvent& event)
{if(event.what==evMessage)
switch(event.command)
{cmQuit:
EndExec();
ClearEvent(event);
break;
cmGet:
cout<<GetSum()<<endl;
ClearEvent(event);
break;
cmSet:
PutSum(event.A);
ClearEvent(event);
break;
default:
TSheme::HandleEvent(event);
} }
TSheme::HandleEvent(TEvent&event)
{TItem* r;
if(event.what==evMassage)
{r=last;
while(event.what!=evNothing&&r!=0)
 {r->HandleEvent(event);
r=r->next;}
} }
TAdd::HandleEvent(TEvent&event)
{if(event.what==evMessage)
switch(event.command)
{cmAdd:
//занести в регистр число
(owner->reg)->SetX(event.A);
//вызвать метод сложения
Add();
```

```
ClearEvent(event);
break;
} }
TAdd::Add() //в сумматор добавить содержимое регистра
{float a,b;
//получить значение сумматора
a=(owner->sum)->GetX();
//получить значение регистра
b=(owner->reg)->GetX();
//изменить значение сумматора
(owner->sum)->SetX(a+b);
}
```

Лабораторная работа № 5
ПЕРЕГРУЗКА ОПЕРАЦИЙ

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++.

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.

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 opera- tor=() 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.

Создание приложений в Borland C++ 5.02.

- Проекты и узлы.

Проект – это файл, содержащий все необходимое для построения конечного продукта: установленные параметры, информацию о целевой среде и о входных файлах. Файл проекта имеет расширение **ide**.

Термин "**узел**" применяется для обозначения различных объектов, находящихся в окне проекта. Каждый узел может зависеть от одного или большего количества узлов. Это означает, что до обработки такого узла, узлы, от которых он зависит, должны быть успешно обработаны. Самый верхний узел в иерархии узлов называется целью. Можно внести больше, чем одну целевую программу в файл **.ide**. Рекомендуется сохранять в одном проекте файлы, связанные по смыслу.

- Программа из одного модуля.

1. Если исходный файл существует, откройте его, выбрав меню **File|Open.** В противном случае – меню **File|New**, команда **Text Edit.** Введите исходный код в новое окно редактора и сохраните в файле, выбрав меню **File|Save.**

2. Активизируйте локальное меню редактора, нажав правую кнопку мыши в середине окна(или **Alt+F10**) и выберите опцию **Target Expert.** Появится окно диалога **Target Expert.**

3. Выберите для вашей программы подходящие параметры и нажмите **Ok**. Например, можно построить программу как стандартное приложение **DOC**, как приложение **EasyWin** для Windows 3.x (16-разрядное) или типа **Concole** для Win32. Рекомендуется строить **EasyWin**-программу.

4. В меню **Debug** (отладка) выберите **Run** (выполнить). Программа будет откомпилирована, отредактирована и выполнена.

5. Для ускорения повторной компиляция в случае обнаружения и исправления ошибок рекомендуется установить режим прекомпиляции и сохранения откомпилированного кода заголовочных файлов. Для этого в

меню **Options|Project|Compiler** выберите опцию **Precompiled Headers** и установите флажок **Generate and use.**

- Проекты и многомодульные программы.

Если для создания программы используется несколько исходных модулей, вы должны организовать проект.

Для организации проекта:

1. В меню **File|New** выберите **Project.** Появится окно **New Target.** Это расширенный вариант окна **Target Expert.**

2. В окне укажите имя проекта (**Project Path and Name**), имя и тип входного модуля (**Target Name**) и требуемые спецификации библиотек C++.

3. Выберите кнопку **Advanced**, чтобы уточнить проект. Появится окно **Advanced Options.**

4. Если программа не применяет управление ресурсами и не включает в себя файл **.def**, выключите селекторы **.rc** и **.def.**

5. Закройте окна **Advanced Options** и **New Target** (кнопкой **Ok**).

6. В появившемся окне **Project** с помощью правой кнопки мыши укажите узел, чтобы активизировать его локальное меню. Чтобы включит в проект новые модули, выберите в меню опцию **Add Node.** Появится окно **Add to Proect List**, которое позволит вам найти и указать те файлы, которые нужно включить в проект.

7. После того, как вы подключили новые узлы, с помощью правой кнопки мыши укажите на целевой узел. Теперь, чтобы построить программу, выберите опцию **Make Node.**

8. После того, как вы создали и сохранили проект, вы сможете в дальнейшем загружать его в **IDE** командой **Open Project** в меню **Project.** При этом загружаются все связанные с ним файлы.

- Построение нескольких целевых модулей.

1. В меню **Project** выберите **New Target.** Появится окно диалога **Add Target.**

2. Введите имя целевого модуля, установите тип цели (**Standard**) и нажмите **Ok.** Появится окно **New Target.**

3. Установите в окне **New Target** требуемые параметры.

4. Если необходимо вновь созданный целевой узел сделать подузлом другого узла, то с помощью левой кнопки мыши “перенесите” узел и “положите” его на узел, подузлом которого он будет.

5. Вызовите правой кнопкой мыши локальное меню целевого узла и выберите **Build Node**, чтобы построить программу.

Что такое EasyWin-программа?

Интегрированная среда разработки(**IDE**) BC++5.02 дает возможность создавать специальный вид программ, называемых EasyWin-программами, которые выполняются в простом окне, напоминающем окно Windows. Это окно содержит стандартные кнопки и полосы скроллера. Текст в окне выводится в кодировке Windows, что не создает проблем с кириллицей.

- Порядок создания EasyWin-программы.

1. Загрузите IDE BC++5.02.
2. Выберите меню **File.**
3. Выберите команду **New/Project,** вызывающую окно **New Target.**
4. Введите маршрут и имя файла проекта **.ide** в поле ввода в верхней части окна (поле **Project Path and Name**) . Введенное имя будет повторено в поле ввода имени цели (поле **Target Name***)*, тем самым имя программы будет соответствовать имени проекта. Кнопка **Brows** служит для выбора каталога, содержащего файлы проекта.
5. Выберите **EasyWin[.exe]** в списке **Target Type**.
6. Щелкните на кнопке **Advanced,** чтобы вызвать диалоговое окно **Advanced Options**. Выберите в нем переключатель, помеченный как **.cpp Node**. Этот выбор заставит IDE вставлять **.cpp-**узлы. Это диалоговое окно дает также возможность добавлять или удалять модули **.rc** и **.def**. Так как реально они нужны только при создании Windows-приложений, а не Easy-Win, удостоверьтесь, что эти две кнопки не выбраны. Закройте окно (**Ok**).
7. Нажмите **Ok**, чтобы создать новый файл проекта.
8. IDE выведет окно проекта **Project**, в котором перечислены узлы различных программ. Когда вы создаете новый файл проекта, окно **Project** будет содержать только один узел.
9. Узлы программы EasyWin содержат только один файл – **.cpp** с текстом программы. Дважды щелкните на файле –**.cpp**, для того чтобы начать редактирование этого файла.
10. Введите исходный текст программы.
11. Откомпилируйте программу (**F9**).
12. Исправьте ошибки и повторите компиляцию.
13. Повторяйте пункт 12, пока компиляция не будет выполняться без ошибок.
14. Выполните программу (**Ctrl+F9**).

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.

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).

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”.

Приложение. Варианты заданий.

1. АТД – множество с элементами типа **char.** Дополнительно перегрузить следующие операции:

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

+ – объединение множеств;

== – проверка множеств на равенство.

2. АТД – множество с элементами типа **char.** Дополнительно перегрузить следующие операции:

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

* – пересечение множеств;

< – сравнение множеств.

3. АТД – множество с элементами типа **char.** Дополнительно перегрузить следующие операции:

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

> – проверка на подмножество;

!= – проверка множеств на неравенство.

4. АТД – множество с элементами типа **char.** Дополнительно перегрузить следующие операции:

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

* – пересечение множеств;

int()– мощность множества.

5. АТД – множество с элементами типа **char.** Дополнительно перегрузить следующие операции:

() – конструктор множества (в стиле конструктора Паскаля);

+ – объединение множеств;

<= – сравнение множеств .

6. АТД – множество с элементами типа **char.** Дополнительно перегрузить следующие операции:

> – проверка на принадлежность(char in set Паскаля);

* – пересечение множеств;

< – проверка на подмножество.

7. АТД – однонаправленный список с элементами типа **char.** Дополнительно перегрузить следующие операции:

\+ – объединить списки (list+list);

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

== – проверка на равенство.

8. АТД – 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. АТД – 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. АТД – 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. АДТ – 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. АДТ – 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. АДТ – стек. Дополнительно перегрузить следующие операции:

+ – добавить элемент в стек;

– – извлечь элемент из стека;

bool()– проверка, пустой ли стек.

14. АДТ – очередь. Дополнительно перегрузить следующие операции:

+ – добавить элемент;

– – извлечь элемент;

bool() – проверка, пустая ли очередь.

15. АДТ – 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. АТД – 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. АТД – 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. АТД – 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. АТД – 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. АТД – двухмерный массив (матрица) вещественных чисел. Дополнительно перегрузить следующие операции:

() – доступ по индексу;

* – умножение матриц;

* – умножение матрицы на число;

* – умножение числа на матрицу.

21. АТД – двухмерный массив (матрица) вещественных чисел. Дополнительно перегрузить следующие операции:

() – доступ по индексу;

- – разность матриц;

- – вычесть из матрицы число;

== – проверка матриц на равенство.

22. АТД – 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. АТД – двухмерный массив (матрица) вещественных чисел. Дополнительно перегрузить следующие операции:

() – доступ по индексу;

== – проверка матриц на равенство;

++ – транспонировать матрицу.

Лабораторная работа № 6
ШАБЛОНЫ ФУНКЦИЙ И КЛАССОВ

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

Основное содержание работы.

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

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.

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.

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?

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.

Лабораторная работа № 7

ПОТОКОВЫЕ КЛАССЫ

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

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.

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.

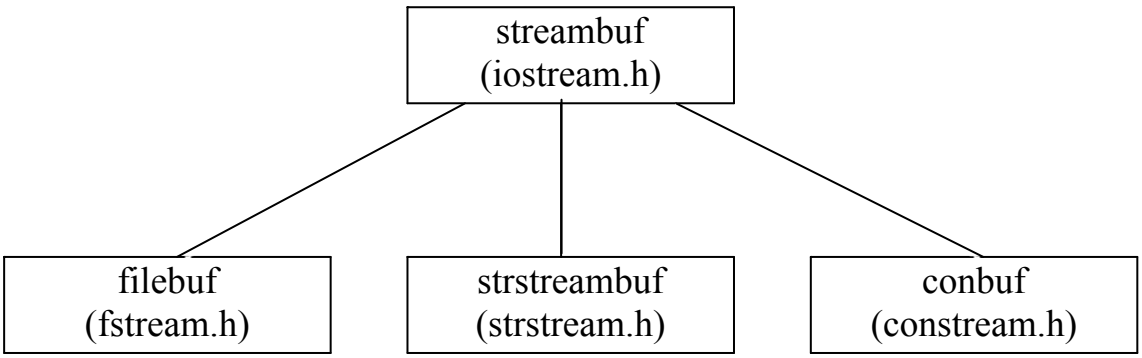
template <class charT, class traits = ios_traits <charT>> class basic_istream: virtual public basic_ios <charT, traits>;

Потоковые классы в C++.

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

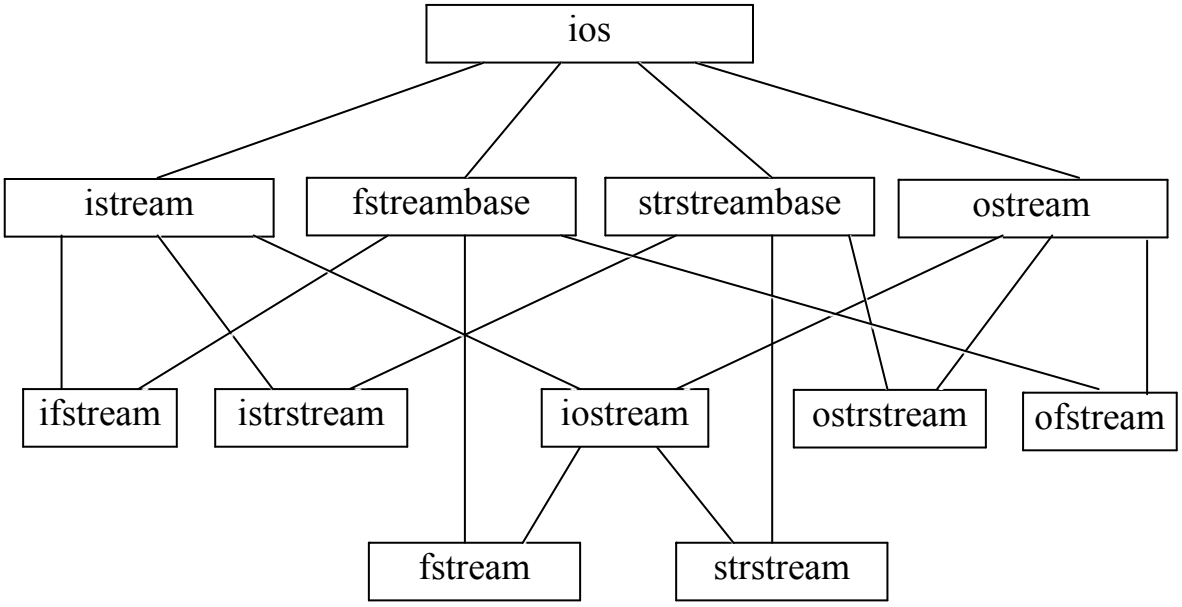
Класс streambuf обеспечивает организацию и взаимосвязь буферов ввода-вывода, размещаемых в памяти, с физическими устройствами ввода-вывода. Методы и данные класса streambuf программист явно обычно не использует. Этот класс нужен другим классам библиотеки ввода-вывода. Он доступен и программисту для создания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

- fstream();

создает поток, не присоединяя его ни к какому файлу.

- 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);

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.

Методические указания.

1. Программы создается как EasyWin-приложение в Borland C++5.02.

Проект должен содержать 5 целевых узлов (по числу программ).

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

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

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

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

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

7. Определение пользовательского класса сохранить в h-файле.

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

9. 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.

10. В поток записать не менее 5 объектов.

11. 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.

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

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

14. 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.

15. 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.

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.

Лабораторная работа № 8
СТАНДАРТНАЯ БИБЛИОТЕКА ШАБЛОНОВ

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

Основное содержание работы.

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

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 унар-

ных 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

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.

Методические указания.

1. Программы создается как EasyWin-приложение в Borland C++5.02. Проект должен содержать 3 целевых узла (по числу программ).

2. В качестве пользовательского типа данных использовать пользовательский класс лабораторной работы № 7.

3. При создании контейнеров в программе № 2 объекты загружать из потока, для чего использовать программы записи и чтения потока из лабораторной работы № 7.

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.

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.

Приложение. 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

Список литературы

Основная

1. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. Второе издание. – М.: Бином, 1998.

2. Паппас К., Мюррей У. Visual C++6: Руководство разработчика. Киев: BHV, 2000.

3. Подбельский В.В. Язык C++ – М.: Финансы и статистика, 1996.

4.Страуструп Б. Язык программирования C++. Третье издание, М.: Бином, 1999.

Дополнительная

1. Аммераль Л. STL для программистов на C++. – М., ДМК, 1999.

2. Грегори К. Использование Visual C++6. – М., Вильямс, 1999.

3. Киммел П. Borland C++5. – СПб.: BHV, 1997.

4. Крейг Арнуш. Borland C++: освой самостоятельно – М.: Бином, 1997.

5. Лейнекер Р. Энциклопедия Visual C++6. – Спб, Питер, 1999.

6. Луис Д. C и C++. Справочник. – М: Бином, 1997.

7. Пол Айра. Объектно-ориентированное программирование на C++. Второе издание. – М.: Бином, 1999.

8. Секунов Н.Ю. Самоучитель Visual C++6. – Спб, BHV, 1999.

9. Скляров В.А. Язык C++ и ООП. – Минск: Вышэйшая школа, 1997.

10. Фейсон Т. Объектно-ориентированное программирование на C++ 4.5. – Киев: Диалектика, 1996.

11. Шилдт Г. Самоучитель C++. Второе издание. – СПб.: BHV, 1998.

12. Шилдт Г. Теория и практика C++. – СПб.: BHV, 1996.

13. Элджер Дж. C++: библиотека программиста – СПб: Питер, 1999.