1 ВВЕДЕНИЕ

Лабораторные работы выполняются в среде Microsoft Visual Studio как консольные приложения по следующим темам:

- Разработка программ с использованием методологии объектно-ориентированного программирования. Реализация классов и объектов на C++.
- Разработка программ с использованием наследования и включения классов. Полиморфизм, реализованный с помощью виртуальных функций;
- Программирование с использованием шаблонов функций и классов. Реализация перегрузки стандартных операций в различных шаблонных классах.

По каждой лабораторной работе оформляется отчет, который должен содержать:

- цель лабораторной работы;
- вариант задания на выполнение лабораторной работы;
- таблицу описания классов;
- схемы алгоритмов всех функций программы;
- листинги файлов программы.

После выполнения лабораторной работы студент должен защитить ее, пояснив процесс обработки данных, схемы алгоритмов и тексты программы, а также ответив на ряд контрольных вопросов.

2 ЛАБОРАТОРНАЯ РАБОТА № 7

Классы и объекты в C++

2.1 Цель лабораторной работы

Целью лабораторной работы является получение практических навыков реализации классов на C++.

2.2 Задание на выполнение лабораторной работы

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

2.3 Порядок выполнения

1. Определить пользовательский класс в соответствии с вариантом задания.
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 статические и динамические объекты и массивы объектов пользовательского класса и каждый вызов конструктора и деструктора сопровождается выдачей соответствующего сообщения (какой объект, какой конструктор или деструктор вызван).
8. Показать в программе использование указателя на объект и указателя на компоненту-функцию.

2.4 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, компонентные 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, для какого объекта, и какой конструктор или деструктор вызываются.

2.5 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);
voidShow(); };
```

Более профессионально определение поля ***name*** типа указатель: *char* name*. Однако в этом случае реализация компонентов-функций усложняется динамическим выделением памяти для поля ***name*** и необходимостью при определении деструктора включить операторы для освобождения этой памяти при уничтожении объектов класса.

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 статических и динамических.

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 с демонстрационной программы.
Для предотвращения многократного включения файла-заголовка следует использовать директивы препроцессора

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

2.6 Теоретические сведения

Объектно-ориентированный подход к программированию.

ООП представляет собой новую идеологию разработки программы. Традиционное процедурное программирование предполагает, что в некотором месте программы будут описаны данные, а затем разрабатываются функции, последовательность выполнения которых, определяет суть и способ обработки данных. Разделение данных и функций – отличительная черта процедурного программирования.

Суть объектно-ориентированного подхода состоит в том, что объединяются некоторые совокупности данных и функций, связанных с этими данными в субстанции, называемые *классами*.

Класс

Класс - фундаментальное понятие 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 – это структурированный тип, состоящий из фиксированного набора возможно разнотипных данных и совокупности функций для обработки этих данных.

Определение класса

ключ_класса имя_класса *// заголовок класса*
{ список компонентов класса}; *// тело класса*

ключ_класса – одно из слов *class , struct , union;*
имя_класса - идентификатор

Список компонентов (или членов класса - member) – это в простейшем случае определения и описания типизированных данных и принадлежащих классу функций.

Определения разнотипных данных в списке отделяется **‘;’** , если данные однотипные, их идентификаторы можно перечислить через запятую, аналогично как в структурах.

Членами класса в общем случае могут быть данные, функции, классы, перечисления, битовые поля, дружественные функции, дружественные классы и имена типов.

Функции – члены класса (member functions) , называют методами класса или компонентными функциями.

Данные класса (data members), называют компонентными данными или элементами данных класса.

Класс – это тип, который служит для определения переменных класса - ***объектов (экземпляров) класса.***

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  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;          //объявление двух объектов
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 “уточненных” имен:
имя_объекта. имя_данного;
имя_объекта. имя_функции(фактические параметры);
Например:
complex x1,x2;
x1.re = 1.24;
x1.im = 2.3;
x2.set(5.1,1.7);
x1.print();
И с использованием указателя на объект
указатель_на_объект–>имя_компонентного данного
указатель_на_объект–> имя_функции(фактические параметры);
*complex *point = &x1; // или point = new complex;*
point –>re = 1.24;
point –>im = 2.3;
point –>print();
Второй способ с помощью “квалифицированных” имен:
<имя объекта> . <имя класса> :: <имя компонента>
<имя объекта> . <имя класса> :: <вызов компонентной функции >
указатель_на_объект–><имя класса> :: <имя компонента>
указатель_на_объект–><имя класса> :: <вызов функции >

Объявление функции вне и внутри тела класса

При объявлении функции вне класса, внутри тела класса помещается только прототип или описание функции

Все экземпляры (объекты) класса будут использовать один код функции, и компилятор будет генерировать код для вызова функции при каждом обращении к функции.

При полном объявлении функции внутри класса функция по умолчанию считается *подставляемой*, т.е. при каждом вызове этих функций их код “встраивается“ в точку вызова. Это, так называемые, ***inline***-функции.

Вместо команд передачи управления единственному экземпляру тела функции компилятор в каждое место вызова функции помещает команды кода операторов тела функции.

Это может увеличить скорость выполнения программы, но увеличивает также размер кода программы. Если метод (функция) большой по объему, не следует настраивать компилятор на генерацию встроенного кода и следует объявить функцию вне класса, используя квалифицированное имя функции.

Например.
*class Men { char * name;*

```
          ...
      void setName ( char*);    //-функция- член класса
        ... };
void   Men:: setName  (char*n)
{ name = n ;}
```

Доступность компонентов класса

В рассмотренных ранее примерах классов (date и complex) компоненты классов являются общедоступными (в классах с ключом *struct* или *union* компоненты общедоступны по умолчанию). В любом месте программы, где "видно" определение класса, можно получить доступ к компонентам объекта класса. Тем самым не выполняется *основной принцип абстракции данных – инкапсуляция (сокрытие) данных внутри объекта.* Для изменения видимости компонент в определении класса можно использовать спецификаторы доступа: ***public, private, protected***.

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

имя_объекта.имя_члена_класса
ссылка_на_объект.имя_члена_класса
указатель_на_объект->имя_члена_класса

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

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

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

Пример.

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

Конструктор

Недостатком рассмотренных ранее классов является отсутствие автоматической инициализации создаваемых объектов. Для инициализации объектов класса в его определение можно явно включить специальную компонентную функцию, называемую ***конструктором***.

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

Формат определения конструктора следующий:

имя_класса (список_формальных_параметров)
{ операторы_тела_конструктора }

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

Конструктор инициализирует данные - члены класса. Конструктор явно или неявно вызывается при определении (или размещении в памяти с помощью операции *new*) каждого объекта класса.

Основное назначение конструктора - превращение участка памяти в объект класса, т.е. инициализация его полей данных

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, mutable, volatile.***

- 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:

- Первый способ - передача значений параметров в тело конструктора.
- Второй способ предусматривает применение списка инициализаторов данного класса.
  Этот список помещается между списком параметров и телом конструктора, отделенный от последнего разделителем ‘ : ‘ (двоеточие). Каждый инициализатор списка относится к конкретному компоненту и имеет вид: *имя_данного (выражение)*. Инициализаторы полей отделяются друг от друга запятыми.

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 <cstring>
#include <iostream>
using namespace std;
class string
{  char *ch; // указатель на текстовую строку
  int len;      // длина текстовой строки
  public:
  // конструкторы
  // создается объект – пустая строка
  string(int N = 80): len(0){ch = new char[N+1]; ch[0] = ‘\0’;}
  // создается 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 в любом классе ***T*** 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 (конструктор копирования по умолчанию). Побитовое копирование не во всех случаях является адекватным. Именно для таких случаев и необходимо определить собственный конструктор копирования.

Обязательное его определение в тех случаях, когда класс содержит поля, являющиеся указателями на динамические участки памяти.

Например, в классе *string*:

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

Массивы объектов статических и динамических

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

Определение статического объекта и массива статических экземпляров класса с явным присваиванием значений:

<имя класса>   <имя объекта> (параметры конструктора);

<имя класса>   <имя массива> [размер массива] =
{ <имя класса>(параметры конструктора для 0-го экземпляра), ...,   <имя класса> (параметры конструктора для последнего экземпляра) };

Определение динамического объекта и массива динамических экземпляров класса с явным присваиванием значений:

*<имя класса> * <имя указателя на объект> = new <имя класса>*
(параметры конструктора);

*<имя класса> * <имя массива указателей на объекты> [разм. массива] = {*
new <имя класса> (параметры конструктора для 0-го экземпляра), ...,
new<имя класса>(параметры конструктора для последнего экземпляра) };

Пример.

```
class demo{
int x;
public:
demo(){x=0;}
demo(int i){x=i;}};
int  main(){
```

```
class demo a[20]; //вызов конструктора без параметров(по умолчанию)
class demo b[2]={demo(10),demo(100)};//явное присваивание
return 0;}
```

Деструктор

Деструктор - это функция, которая автоматически выполняется, когда экземпляр класса уничтожается.

Деструктор не уничтожает объект, а вызывается либо при выходе объекта за пределы своей области видимости, либо при уничтожении динамического объекта операцией *delete* и выполняет запрограммированные в его теле действия.

Назначение – выполнение всех действий, сопровождающих удаление объекта. Наиболее важное - это освобождение всех ресурсов, включенных в объект при его создании или выполнении действий над объектом. Такими ресурсами могут быть участки памяти динамически выделенными для полей объекта, файлы, открытые при создании объекта и связанные с ним, установка видеосистемы в исходное состояние, написание каких-либо фраз и т.д.

Класс может иметь несколько конструкторов, но деструктор может быть только один.

Формат деструктора:

~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.
- Деструктор может быть вызван явно для некоторого объекта, при этом объект не уничтожается, а выполняются операторы тела деструктора.

Указатели на компоненты-функции

Можно определить указатель на компонентные функции. Вне класса можно следующим образом определить указатель на метод класса:

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

Пример.
*double(complex : :*ptcom)(); // Определение указателя*
ptcom = &complex : : real; // Настройка указателя на метод real
complex A(5.2,2.7);
// Теперь для объекта A можно вызвать его функцию real через указатель
*cout<<(A.*ptcom)();*
Можно определить также тип указателя на функцию
*typedef double(complex::*PF)();*
а затем определить и сам указатель с инициализацией
PF ptcom=&complex::real;

2.7 Контрольные вопросы

1. Объектно-ориентированный подход к программированию.
2. Понятие класса как структурированного типа. Определение класса
3. Внешнее и внутреннее определение компонентных функций. ***inline***-функции.
4. Статусы доступа к членам класса. Спецификаторы доступа.
5. Создание объектов (экземпляров класса) статических и динамических. Инициализация. Обращение к компонентным данным и функциям.
6. Компонентные данные и компонентные функции.
7. Конструкторы и деструкторы. Назначение, формат определения, свойства.
8. Конструктор с параметрами. Список инициализации. Создание объектов (экземпляров класса) и массивов объектов статических и динамических.
9. Конструктор с аргументами, задаваемыми по умолчанию.
10. Конструктор по умолчанию.
11. Конструктор копирования. Назначение. Копирование по умолчанию.
12. Указатели на компонентные функции.

2.8 Варианты задания

1. СТУДЕНТ имя – char* курс – int пол – int(bool)	2. СЛУЖАЩИЙ имя – char* возраст – int рабочий стаж – int	3. КАДРЫ имя – char* номер цеха – int разряд – int
4. ИЗДЕЛИЕ имя – char* шифр – char* количество – int	5. БИБЛИОТЕКА имя – char* автор – char* стоимость – float	6. ЭКЗАМЕН имя студента – char* дата – int оценка – int
7. АДРЕС имя – char* улица – char* номер дома – int	8. ТОВАР имя – char* количество – int стоимость – float	9. КВИТАНЦИЯ номер – int дата – int сумма – float
10. ЦЕХ имя – char* начальник – char* количество работающих – int	11. ПЕРСОНА имя – char* возраст – int пол – int(bool)	12. АВТОМОБИЛЬ марка – char* мощность – int стоимость – float
13. СТРАНА имя – char* форма правления – char* площадь – float	14. ЖИВОТНОЕ имя – char* класс – char* средний вес – int	15. КОРАБЛЬ имя – char* водоизмещение – int тип – char*
16. ГОРОД имя – char* численность население - int год создания - int	17. САМОЛЕТ ФИО конструктора – char* год создания – int грузоподъемность- float	18. ЭКСКУРСИЯ страна - char* продолжительность в часах – int стоимость -float
19.СЕКЦИЯ название – char* характер занятий – char* стоимость - float	20. МУЗЕЙ название - char* характер экспозиции – char* стоимость билета-float	21. КИНОТЕАТР название –char* количество мест – int стоимость билета - float
22. ПЕЧАТНОЕ ИЗДАНИЕ название – char* тип – char* стоимость -float	23. КВАРТИРА адрес– char* кол. комнат – int площадь в кв. м.- float	24.ХОЛОДИЛЬНЫЙ АГРЕГАТ название – char* емкость камеры - float стоимость - float

3 ЛАБОРАТОРНАЯ РАБОТА № 8

Наследование и виртуальные функции

3.1 Цель лабораторной работы

Целью лабораторной работы является получение практических навыков использования основных методов наследования классов, создания иерархии классов и использования статических компонентов класса.

2.3 Задание на выполнение лабораторной работы

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

3.3 Порядок выполнения

1. Определить иерархию классов (в соответствии с вариантом).
2. Определить в базовом классе статическую компоненту - указатель на начало связанного списка объектов и статические функции для просмотра списка и очистки списка. В базовом классе объявить чистую виртуальную функцию для просмотра объекта.
3. Реализовать производные классы в соответствии с иерархией классов.
4. Написать демонстрационную программу, в которой создаются объекты различных классов и помещаются в список, после чего список просматривается.
5. Сделать соответствующие методы просмотра объектов виртуальными.
6. Реализовать вариант, когда объект добавляется в список при создании, то есть в конструкторе (смотри пункт 6 следующего раздела).

3.4 Содержание отчета

1. Титульный лист: название дисциплины; номер и наименование работы; фамилия, имя, отчество студента; дата выполнения.
2. Постановка задачи. Следует дать конкретную постановку, то есть указать, какие классы должны быть реализованы, какие должны быть в них конструкторы, деструкторы, данные и методы.
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 и очищения списка.
8. Листинг демонстрационной программы.
9. Объяснение необходимости виртуальных функций. Следует показать, какие результаты будут в случае виртуальных и не виртуальных функций.

3.5 Методические указания

1. Для определения иерархии классов связать отношением наследования классы, приведенные 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
*static void clear(void); //*освобождение списка
5. Статическую компоненту инициализировать вне определения класса, в глобальной области.
6. Добавление объекта в список можно осуществлять следующими описанными ниже способами.
- Во-первых, можно предусмотреть метод класса, то есть объект сам добавляет себя в список. Например, *a.Add()* - объект *a* добавляет себя в список.
- Во-вторых, включение объекта в список можно выполнять при создании объекта, то есть поместить операторы включения создаваемого объекта в конструктор класса. В случае иерархии классов, включение объекта в список должен выполнять только конструктор базового класса. Программа должна продемонстрировать оба этих способа.
7. Список будет состоять из объектов производных классов. Список просматривать путем вызова виртуального метода ***show*** каждого объекта списка.
8. Статический метод просмотра списка вызывать не через объект, а через класс.
9. Если производные классы ресурсоемкие, то деструктор в базовом классе следует объявлять виртуальным. Чтобы не осталось не освобожденных ресурсов при уже удаленных объектах.
10. Определение классов, их реализацию, демонстрационную программу поместить в отдельные файлы.

3.6 Теоретические сведения

Статические члены класса

Такие компоненты должны быть определены в классе, как **статические** *(**static**)*. Статические данные классов не дублируются при создании объектов, то есть каждый статический компонент существует в единственном экземпляре для всех объектов класса. 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 статический компонент класса получает память и становится доступным. Обращаться к статическому данному класса можно обычным образом через имя объекта:
имя_объекта.имя_компонента

К статическим компонентам после их инициализации можно обращаться и тогда, когда объекты класса еще не определены. Доступ к статическим компонентам возможен не только через имя объекта, но и через имя класса:
имя_класса : : имя_компонента

Однако так можно извне обращаться только к открытым, *public* компонентам.

Для обращения к *private* статической компоненте извне можно с помощью открытых методов класса. К моменту обращения к статическим данным класса, объекты класса могут быть еще не определены. Без имени объекта обычный метод класса вызвать нельзя в соответствии с требованиями синтаксиса. Хотелось бы иметь возможность обойтись без имени конкретного объекта при обращении к статическим данным. Такую возможность дают ***статические компонентные функции***. Статические методы класса можно вызывать, используя квалифицированное имя функции:
имя_класса : : имя_статической_функции

Пример:

```
#include <iostream>
using namespace std;
class TPoint {
  double x,y;
  static int N;  // статический компонент: 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
int main(){
  TPoint A(1.0,2.0);
  TPoint B(4.0,5.0);
  TPoint C(7.0,8.0);
  cout<< \nОпределены ”<<TPoint : : count()<<“точки”;
return 0;}
```

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.

Включение и наследование классов

Семантика отношений между классами может быть реализована по схеме ***наследования*** и по схеме ***включения***.

Об отношении ***включения*** говорят, используя выражение "включает как часть" (***has a*** – владеет, содержит в себе как часть).

При ***наследовании*** базовый класс – представляет объекты общего вида, производный класс описывает более конкретные объекты, которые являются разновидностью (частным случаем объектов базового класса).

Об отношении наследования можно сказать, используя выражение "является частным случаем" ***(is a)***.

Например, самолет является частным случаем транспортного средства. Это отношение наследования классов.

Самолет имеет крылья, мотор – здесь отношение включения.

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.

Производный класс получает в наследство поля данных и методы базового класса. При этом наследуемые компоненты не перемещаются в производный класс, а остаются в базовом классе. В каждый объект производного класса входит безымянный объект базового класса со всеми своими полями и методами.

Из наследуемых компонентов базового класса для объектов производного класса доступны компоненты со статусом *public и protected*.

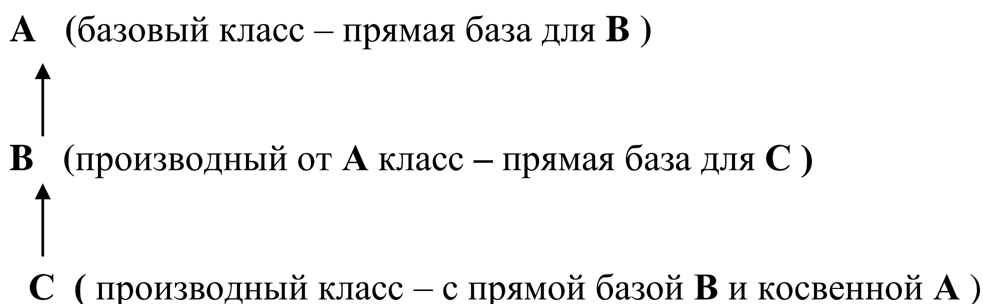
Любой производный класс может в свою очередь быть базовым для других классов и таким образом формируется структура, называемая ***иерархией классов***, определяющая для каждого класса приложения родственные связи (“родитель - потомок”) его с другими классами приложения.

Класс является ***прямым базовым классом***, если он входит в список базовых классов при определении производного класса.

А если сам базовый класс является производным от некоторого родителя, причем этот родитель не входит в список базовых классов, то этот родитель является ***непрямым (косвенным) базовым классом.***

Иерархию производных классов принято отображать в виде ***направленного ациклического графа (НАГ)***, где стрелкой изображают связь “ производный от”.

Производные классы располагаются ниже базовых. В том же порядке они должны располагаться в программе и так их объявления рассматривает компилятор.



На практике часто возникает необходимость создать производный класс, наследующий возможности нескольких классов.

Наличие в определении производного класса несколько прямых базовых классов называют ***множественным наследованием.***

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 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 реализовывать 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.* 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.*

3.7 Контрольные вопросы

1. Применение статического элемента класса в связанных списках объектов класса.

2. Указатель ***this*** . Применение указателя ***this*** в связанных списках объектов класса.
3. Различие между копированием и присваиванием. Блокировка копирования и присваивания.
4. Преобразование типов в классах пользователя, явные и неявные.
5. Отношения включения классов и наследования классов.
6. Наследование. Суть метода. Определение производного класса. Влияния формата определения производного классов и спецификаторов доступа на доступ наследуемых элементов.
7. Наследование. Передача параметров конструктора в базовый класс. Конструкторы с инициализацией по умолчанию в иерархии классов.
8. Множественное наследование. Порядок вызовов конструкторов и деструкторов базовых классов при множественном наследовании.
9. Множественное наследование. Прямое и косвенное наследование.
10. Иерархия производных классов в виде графа (НАГ).
11. Дублирование объектов базового класса, косвенно наследуемого при множественном наследовании
12. Виртуальные базовые классы. Примеры иерархии классов (НАГ), с участием виртуальных базовых классов**.**
13. Полиморфизм. Понятие виртуальной функции. Режимы раннего и позднего связывания. Полиморфные классы.
14. Замещение функций в производных классах. Виртуальные функции.
15. Пустая и чистая виртуальные функции. Абстрактный класс, назначение, свойства.
16. Преобразование типов указателей в иерархии классов. Работа с виртуальными функциями.

3.8 Варианты задания

Перечень классов:

1) студент, преподаватель, персона, зав. кафедрой;
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.

4 ЛАБОРАТОРНАЯ РАБОТА № 9

Шаблоны функций и классов

4.1 Цель лабораторной работы

Целью лабораторной работы является получение практических навыков создания абстрактных типов данных и перегрузки операций в языке С++, создания шаблонов абстрактных типов данных и использования их в программах на С++.

2.4 Задание на выполнение лабораторной работы

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

4.3 Порядок выполнения

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

4.4 Содержание отчета

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

Следует дать конкретную постановку, то есть указать шаблон, какого класса должен быть создан, какие должны быть в нем конструкторы, компонентные 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 перегружаемых операций, которые заданы в варианте задания.

6. То же самое для пользовательского класса.
7. Реализация перегруженных операций с обоснованием выбранного способа (функция - член класса, внешняя функция, внешняя дружественная функция).
8. Результаты тестирования. Следует указать для каких типов, и какие операции проверены.

4.5 Методические указания

1. Класс одномерный массив реализовать как динамический массив (вектор). Для этого определение класса должно иметь следующие поля:
- указатель на начало массива;
- текущий размер массива.
2. Для ввода и вывода определить в классе функции ***input*** и ***print***.
3. Чтобы 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 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 использовать 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. Тестирование должно быть выполнено для всех типов данных и для всех операций.

4.6 Теоретические сведения

Шаблоны классов, так же как и шаблоны функций, поддерживают в C++ парадигму ***обобщенного программирования***, то есть программирования с использованием типов в качестве параметров.

Механизм шаблонов в C++ допускает применение *абстрактного типа в качестве параметра* при определении класса или функции.

После того как шаблон класса определен, он может использоваться для определения конкретных классов.

Процесс генерации компилятором определения конкретного класса по шаблону класса и аргументам шаблона называется *инстанцированием шаблона (template instantiation).*

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 С++ параметризированная функция создается с помощью ключевого слова ***template***.

Для параметризации используется список формальных параметров шаблона, который следует после слова ***template***, заключенный в угловые скобки < >. Каждый параметр обозначается словом ***class*,** за которым следует имя параметра. Имя параметра – это название типа, его можно использовать для обозначения типов параметров функции, типа возвращаемого результата и для объявления типов локальных переменных функции.

Формат шаблона функции:

template<class тип_данных>
тип_воз_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 не параметризованных формальных параметров. Может быть не параметризованное 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.

Шаблоны функций предназначены для того чтобы отделить алгоритм обработки данных, независимый от конкретных типов данных, с которыми он работает, передавая тип в качестве параметра.

Шаблон класса

Шаблоны классов предоставляют аналогичную возможность, позволяя создавать параметризованные классы.

Параметризованный класс создает семейство родственных классов, которые можно применять к любому типу данных, передаваемому в качестве параметра.

Наиболее широкое применение шаблоны находят при создании *контейнерных классов.*

Контейнерным называется класс, который предназначен для хранения каким-либо образом организованных данных и работы с ними.

Преимущество использования шаблонов состоит в том, что как только алгоритм работы с данными определен и отлажен, он может применяться к любым типам данных без переписывания кода.

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 { . . . };

Отметим, что язык позволяет вместо ключевого слова class перед параметром шаблона использовать другое ключевое слово — ***typename***, то есть написать:

template < typename T> class point { /* ... */ };

После объявления *T* используется внутри шаблона точно так же, как имена других типов.

Параметры шаблонов

Параметрами шаблонов могут быть абстрактные типы или переменные встроенных типов, таких как **int**.

Первый вид параметров мы уже рассмотрели. При инстанцировании на их место подставляются аргументы либо встроенных типов, либо типов, определенных программистом.

Второй вид используется, когда для шаблона предусматривается его настройка некоторой константой. Например, можно создать шаблон для массивов, содержащих ***n*** элементов типа ***T***:

```
template <class T, int n> class Array { / * . . . * / };
```

Тогда, объявив объект:

```
Array<point, 20> ap;
```

мы создадим массив из 20 элементов типа point.

Приведем менее тривиальный пример использования параметров второго вида:

```
void f1() { cout << "I am f1 () . " << endl; }
void f2() { cout << "I am f2 ( ) . " << endl; }

template<void (*pf)()>
struct A { void Show() { pf();} };

int main() {
A<&fl> aa;
aa. Show();   // вывод: I am f1().
A<&f2> ab;
ab. Show();// вывод: I am f2().
return 0;
}
```

Здесь параметр шаблона имеет тип указателя на функцию. При инстанцировании класса в качестве аргумента подставляется адрес соответствующей функции. Естественно, у шаблона может быть несколько параметров.

Параметры шаблона могут иметь умалчиваемые значения:

```
template<class T=char, int size=64>
class arr  {
T data [size];
int length;
public:
arr():length(size){ }
T& operator [ ] (int i) {
if( i<0 | | i> size){ cout<<”Index error”; exit(1);}
return data[i];
}
```

```
};
int main () {
arr<double,5> rf;
arr<int>ri; // массив int , умалчание для size
arr< > rc; // умолчание для обоих параметров
for(int i=0; i<5; i++)
{   rf[i]=i;    ri=i*i;  rc='A'+i; }
for(int i=0; i<5; i++)
cout<<rf[i]<< "  "<< ri[i]<< " " <<rc[i]<< '\t';
return 0;
}
0 0 A     1 1 B     2 4 C     3 9 D     4 16 E
```

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

Определение встроенных методов внутри шаблона класса практически не отличается от записи в обычном классе. Но если определение метода выносится за пределы шаблона класса, то синтаксис его заголовка усложняется.

Если метод описывается вне шаблона, его заголовок должен иметь следующие элементы:

```
template <описание_параметров_шаблона>
возвр_тип  имя_класса <параметры_шаблона>::
имя_функции (список_параметров функции)
```

Пример:
```
template <class T> class point {
public:
point(T _x = 0, T _y = 0) : x(_x), y(_y) {}
void Show();
private:
T x, y;
};
template <class T> void point<T>::Show() {
cout << " (" << x << ", " <<y << ")" <<endl;
```

Использование шаблона класса

При включении шаблона класса в программу никакие классы на самом деле не генерируются до тех пор, пока не будет создан экземпляр шаблонного класса, в котором вместо абстрактного типа ***T*** указывается некоторый конкретный тип.

Такая подстановка приводит к ***актуализации***, или ***инстанцированию***, шаблона. Как и для обычного класса, экземпляр создается либо объявлением объекта, например:

point<int> anyPoint(13, -5);

либо объявлением указателя на актуализированный шаблонный тип с присваиванием ему адреса, возвращаемого операцией *new*, например:

point<double>* pOtherPoint = new point<double>(9.99. 3.33);

Встретив подобные объявления, компилятор генерирует код соответствующего класса.

Еще один пример.

```
// шаблон векторов
template<class T>           // T – параметр шаблона
class Vector       // Vector  - имя семейства классов
{T* data ;          //    данные класса
 int size ;           // размер пространства
 public:
 Vector(int);       // конструктор
 ~Vector () { delete [ ]data; }    //  деструктор
// перегрузка операции  “ [ ]”
 T& operator [ ] ( int i) { return data[i];}
friend ostream & operator << ( ostream& ,  Vector <T> );
};
template<class T>   // внешнее определение конструктора шаблона
Vector <T>:: Vector(int n)
{data = new T[n];
```

```
size =n;}
// определение перегрузки операции  <<
ostream & operator << ( ostream& out ,  Vector <T> X)
{ out<<endl;
for( int i=0;i<X.size; i++)
out<<X[i]<<"  "; return out;}
```

Теперь можно объявлять объекты конкретных классов, порожденных из шаблона:

имя параметризованного класса(шаблона)
< фактические параметры шаблона>
имя объекта (параметры конструктора);

т.е. имя конкретного класса будет:

имя параметризованного класса(шаблона)
< фактические параметры шаблона>

```
Vector<char> A( 5) ;   // вектор A – массив из пяти значений типа
 //char,  здесь Vector <char> - имя класса

int main()
{clrscr();
// статическте объекты:
Vector<int> X(5);
Vector <char> C(5);
// динамический объект:
Vector<int>*p= new Vector<int> (10);
// заполнение элементов значениями и вывод элементов вектора,
// используя перегруженную операцию []
for(int j =0; j<10; j++)
{(*p)[j]= j; cout<< (*p)[j]<<" ";}
return ;}
```

Достоинства и недостатки шаблонов

Шаблоны представляют собой мощное и эффективное средство обращения с различными типами данных, которое можно назвать параметрическим полиморфизмом, а также обеспечивают безопасное использование типов.

Однако следует иметь в виду, что программа, использующая шаблоны, содержит полный код для каждого порожденного типа, что может увеличить размер исполняемого файла.

Перегрузка операций

Возможность использовать знаки стандартных операций для записи выражений как для встроенных, так и для абстрактных типов данных. Примером абстрактного типа данных является класс в языке С++.

Перегрузка операций - это распространение действий стандартных операций на операнды, для которых эти операции не предполагались или предание стандартным операциям другое назначение.

Формат определения операции-функции:
Тип возвращаемого значения operator знак_ операции (спецификация формальных параметров)
{ тело операции-функции }

Механизм перегрузки операций во многом схож с механизмом определения функций, если принять что:
- конструкция ***operator <знак операции>*** есть имя некоторой функции;
- *список формальных параметров* – список операндов с указанием их типов, которые участвуют в операции;
- *тип возвращаемого результата* – это тип значения, которое возвращает операция;
- *тело операции-функции* – это алгоритм нового действия операции.

Большинство операций перегружаемы, однако, не все.
Операции, не допускающие перегрузки:
. **-** операция доступа к элементу класса
.* - операция доступа к указателю на элемент класса
?: **-** условная операция
: : - операция разрешения области видимости
sizeof **-** размер объекта
**-** препроцессорное преобразование строк
**-** препроцессорная конкатенация строк

Перегрузка унарных операций

- Любая унарная операция *A* может быть определена двумя способами: либо как компонентная функция без параметров, либо как глобальная (возможно дружественная) функция с одним параметром. В первом случае выражение *A Z* означает вызов *Z.operator A ()*, во втором - вызов *operator A(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 operator++(person& ob)
{++ob.age;}
void main()
{class person jon;
++jon;}
```

Перегрузка бинарных операций

- Любая бинарная операция *A* 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 A y*** означает вызов ***x.operatorA(y),*** во втором – вызов ***operator A(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

Рассмотрим перегрузку ряда операций для класса *Complex*:

```
class Complex {
 float re , im ;
 public:
 Complex ( float, float ) ; // конструктор для инициализации
 friend float real (Complex);          // функция –друг для получения re
 friend float image (Complex);    // функция –друг для получения im
Complex operator – ( ) ;// перегрузка одноместной операции ”-” , метод
                                  //класса
 Complex operator – ( Complex&)  ;      // перегрузка двуместной операции
                                                   // ”-” , метод класса
 friend Complex operator + ( Complex&, Complex&) ;   // перегрузка “+”-
                                                               //друг
 };
  Complex :: Complex (float r , float i )
    {re = r ; im = i ; }

  Complex Complex:: operator-( )
   { return Complex (-re, -im);  }
```

```
Complex  Complex :: operator-( Complex& z) // передача по ссылке , чтобы
{ return Complex (re – z.re, im- z.im);  }        // не копировать объект в стек

Complex  operator+( Complex &z1, Complex &z2)
{ return Complex  (z1.re + z2.re ,  z1.im+ z2.im ) ;  }

float  real (Complex z){ return z.re ; }
float  image (Complex z ){ return z.im ;}

void main ( )
{ Complex c1 (1, 2) ;             // создание   объекта c1
Complex c2 (3, 4) ;             // создание   объекта c2
Complex c3 = - c2 ;    // копирование в c3 результата вызова  c2.operator-( );
Complex c4=c2-c1; //копирование в c4 результата вызова  c2.operator-(c1 );
Complex c5=c2+c1// копирование в c5 результата вызова operator+(c2,c1 );
```

Видим в двух последних строках, что синтаксис правил использования функций - перегрузки операций, определенных как методы класса или как друзья класса, совершенно одинаков.

Однако определение операций-функций как дружественных создает преимущество в смысле симметрии и, главное, в тех случаях, когда для выполнения действий над операндами требуется преобразование типа операндов.

Дело в том, что компилятор автоматически выполняет преобразование типа для *аргументов* функций, но не для объекта, для которого вызывается функция-член.

Если *функция-оператор* (другое название *операции - функции*) реализуется как друг и получает оба аргумента как параметры функции, то компилятор выполняет автоматическое преобразование типов двух аргументов операции.

- В соответствии с семантикой бинарных операций ***=, [], ->*** *операции–функции* с названиями не могут быть внешними, а должны быть нестатическими методами того класса для которого они определены.

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()
{ adresbook persons;
   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 *operator=*.

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

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

Блокировка копирования и присваивания:

Для того, чтобы исключить непреднамеренное дублирование объекта класса, надо в его закрытой части объявить конструктор копирования и перегрузку операции присваивания.

```
 private :
  A ( A& );
  A  operator = ( A &) ;
```

И тогда ни одно из приведенных в главной функции дублирований объектов скомпилировать не удастся.

Преобразование типов в классах пользователя

Иногда бывает необходимость преобразовать переменные некоторого класса в базовые типы.

Проблема решается с помощью специального ***оператора преобразования типа.***

Рассмотрим это преобразование для класса ***stroka*.**

Преобразование типа ***stroka*** в тип ***char*.***

Надо в описание класса включить компонентную функцию:

operator char* ( ) { return ch; } **(1)**

И тогда в предложении

stroka s1(“string”);

char s = s1;*

произойдет преобразование объекта s1 к типу ***char**** **,** правило (1)**,** и переменной **s** присвоится значение***“string”.***

Аналогично можно определить в классе преобразование:

operator int ( ) { return len; } **(2)**

И тогда в предложении:

int l = s1;

переменная класса (объект *s1*) преобразуется к целому значению, по правилу, описанному в операторе преобразования (2).

4.7 Контрольные вопросы

1. В чем смысл использования шаблонов?
2. Синтаксис/семантика шаблонов функций?
3. Синтаксис/семантика шаблонов классов?
4. Определить параметризованную функцию сортировки массива методом обмена.
5. Определить шаблон класса “вектор” - одномерный массив.
6. Что такое параметры шаблона функции?
7. Перечислите основные свойства параметров шаблона функции.
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?
17. Каковы синтаксис/семантика “операции-функции”?
18. Когда нужно перегружать операцию присваивания для определенного пользователем типа данных, например класса?
19. Можно ли изменить приоритет перегруженной операции?
20. Можно ли изменить количество операндов перегруженной операции?
22. Можно ли, используя дружественную функцию, перегрузить оператор присваивания?
23. Все ли операции языка C++ могут быть перегружены?
25. Все ли операции можно перегрузить с помощью глобальной дружественной функции?
26. В каких случаях операцию можно перегрузить только глобальной функцией?
27. В каких случаях глобальная операция-функция должна быть дружественной?

4.8 Варианты задания

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 - однонаправленный список list. Дополнительно перегрузить следующие операции:
+ - добавить элемент в начало (list+item);
-- - удалить элемент из начала (--list);
= = - проверка на равенство.
5. Класс - однонаправленный список list. Дополнительно перегрузить следующие операции:
+ - добавить элемент в начало (item+list);
-- - удалить элемент из начала (--list);
!= - проверка на неравенство.
6. Класс - однонаправленный список list. Дополнительно перегрузить следующие операции:

+ - добавить элемент в конец (list+item);
-- - удалить элемент из конца (типа list--);
!= - проверка на неравенство.
7. 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.
8. 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.
9. 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.
10. Класс - стек stack**.** Дополнительно перегрузить следующие операции:
+ - добавить элемент в стек;
-- - извлечь элемент из стека;
bool() - проверка, пустой ли стек.
11. Класс - очередь queue. Дополнительно перегрузить следующие операции:
+ - добавить элемент;
-- - извлечь элемент;
bool() - проверка, пустая ли очередь.
12. Класс - одномерный массив. Дополнительно перегрузить следующие операции:
+ - сложение массивов;

[] - доступ по индексу;
+ - сложить элемент с массивом.
13. Класс - одномерный массив. Дополнительно перегрузить следующие операции:
- - вычитание массивов;
[] - доступ по индексу;
- - вычесть из массива элемент.

5. СПИСОК ЛИТЕРАТУРЫ

1) Подбельский В.В. Стандартный Си++. - М.: Финансы и статистика , 2008.
2) Павловская Т. А., Щупак Ю. А. С++. Объектно-ориентированное программирование. Практикум. – СПб.: Питер, 2006.
3) Павловская Т. А. С/С+. Программирование на языке высокого уровня. – СПб.: Питер, 2003.

СОДЕРЖАНИЕ

1 Введение
2 Лабораторная работа № 7
Классы и объекты в С++.
2.1 Цель лабораторной работы
2.2 Задание на выполнение лабораторной работы
2.3 Порядок выполнения
2.4 Содержание отчета
2.5 Методические указания
2.6 Теоретические сведения
2.7 Контрольные вопросы
2.8 Варианты задания
3 Лабораторная работа № 8
Наследование и виртуальные функции.
3.1 Цель лабораторной работы
3.2 Задание на выполнение лабораторной работы
3.3 Порядок выполнения
3.4 Содержание отчета
3.5 Методические указания
3.6 Теоретические сведения
3.7 Контрольные вопросы
3.8 Варианты задания
4 Лабораторная работа № 9
Шаблоны функций и классов.
4.1 Цель лабораторной работы
4.2 Задание на выполнение лабораторной работы
4.3 Порядок выполнения работы
4.4 Содержание отчета
4.5 Методические указания
4.6 Теоретические сведения
4.7 Контрольные вопросы
4.8 Варианты заданий
5 СПИСОК ЛИТЕРАТУРЫ