Документ подписан простой электронной подписью
Информация о владельце:
ФИО: Локтионова Оксана Геннадьевна
Должность: проректор по учебной работе
Дата подписания: 16.06.2023 12:37:44
Уникальный программный ключ:
0b817ca911e6668abb13a5d426d39e5f1c11eabbf73e943df4a4851fda56d089

МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Юго-Западный государственный университет»
(ЮЗГУ)

Кафедра информационных систем и технологий

УТВЕРЖДАЮ
Проректор по учебной работе
_____________ О.Г. Локтионова
«30» 04 _____________ 2019 г.

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОГРАММИРОВАНИЕ

Методические указания к выполнению лабораторных работ для студентов направления подготовки «Математическое обеспечение и администрирование информационных систем»

Курск 2019

УДК 004.42

Составитель Д.О. Бобынцев

Рецензент: к.т.н., Ю.А. Халин

Объектно-ориентированный анализ и программирование: методические указания к выполнению лабораторных работ/ Юго-Зап. гос. ун-т; сост.: Д.О. Бобынцев. Курск, 2019. 60с.: табл. 8. Библиогр.: с. 60.

Содержит методические указания к выполнению лабораторных работ по дисциплине «Объектно-ориентированный анализ и программирование». Указывается порядок выполнения работ, контрольные вопросы. Предназначено для студентов направления подготовки «Математическое обеспечение и администрирование информационных систем».

Текст печатается в авторской редакции

Подписано в печать 30.04.19. Формат 60х84 1/16.
Усл.печ. л. 3,5. Уч.-изд. л. 3,16. Тираж 100 экз. Заказ. 464 Бесплатно.
Юго-Западный государственный университет.
305040, г. Курск, ул. 50 лет Октября, 94.

Содержание

1. Структуры и объединения.
2. Классы и объекты.
3. Наследование и виртуальные функции.
4. Полиморфизм. Перегрузка операций.
5. Библиотека шаблонов STL.

Структуры и объединения

1 Цель работы

Целью лабораторной работы является получение практических навыков в программировании структур, объединений и перечислений.

2 Темы для предварительной проработки

- арифметические операции;
- порядок выполнения операций;
- стандартные математические функции;
- работа с оператором цикла for;
- работа с оператором условия if;
- одномерные массивы;
- двумерные массивы.

3 Теоретический материал

3.1 Структуры в C++

Объявление структуры

Структуры языка C++ представляют поименованную совокупность компонентов, называемых полями, или элементами структуры. Элементом структуры может быть:

- переменная любого допустимого типа;
- битовое поле;
- функция;

Объявление структуры имеет следующее формальное описание:

```
struct [имя_структуры] {
   тип_элемента_структуры  имя_ элемента1;
   тип_элемента_структуры  имя_ элемента2;
    ...
   тип_элемента_структуры  имя_ элементаN;
} [список_объявляемых_переменных];
```

Объявление структуры с битовыми полями имеет следующее формальное описание:

```
struct [имя_структуры] {
```

```
тип_элемента_структуры
   [имя_ элемента1] : число_бит;
тип_элемента_структуры
   [имя_ элемента2] : число_бит;
 ...
тип_элемента_структуры
   [имя_ элементаN] : число_бит;
} [список_объявляемых_переменных];
```

Возможно неполное объявление структуры, имеющее следующее формальное описание:

struct имя_структуры;

При отсутствии имени объявляемой структуры создается анонимная структура. При создании анонимной структуры обычно указывается список объявляемых переменных.

Список объявляемых переменных типа данной структуры может содержать:

- имена переменных;
- имена массивов;
- указатели;

Например: struct sA {char a[2], int i;} struA, struB[10], *struC;

Для использования указателя на структуру ему необходимо присвоить адрес переменной типа структуры.

Размер структуры с битовыми полями всегда кратен байту. Битовые поля можно определять для целочисленных переменных типа int, unsigned int, char и unsigned char. Одна структура одновременно может содержать и переменные, и битовые поля. Если для битового поля не задано имя элемента, то доступ к такому полю не разрешен, но количество указанных бит в структуре размещается.

Типом элемента структуры может быть:

- другой структурный тип (допускаются вложенные структуры);
- указатель на данный структурный тип;
- неполно объявленный структурный тип;
- любой другой базовый или производный тип, не ссылающийся рекурсивно на объявляемый структурный тип.

Например:

structsA {chara[2], sA* this_struct;}; // Корректное объявление структуры
struct sB; // Неполное объявление структуры
struct sA {char a[2], sB* this_struct;}; // Корректное объявление структуры
struct sA {char a[2], sA this_struct;}; // Ошибочноеобъявление

Структура не может содержать в качестве вложенной структуры саму себя, но она может содержать элемент, являющийся указателем на объявляемую структуру.

Например:

```
struct structA {
   struct structA *pA; int iA; } sA;
// pA указатель на структуру
```

При одновременном объявлении структурного типа, объявлении переменной данного типа и ее инициализации список значений указывается в фигурных скобках в последовательности, соответствующей последовательности определения элементов структуры.

Например:

```
struct POINT   // Объявление структурного типа POINT
{
  int x;      // Объявление элементов x и y
  int y;
} p_screen = { 50, 100 };     // Эквивалентно записи p_screen.x =
50; и p_screen.y = 100;
```

Выделение памяти

При создании переменной типа структуры:

- память под все элементы структуры выделяется последовательно для каждого элемента;

- для битовых полей память выделяется, начиная с младших разрядов;
- память, выделяемая под битовые поля, кратна байту;
- общая выделяемая память может быть больше, чем сумма размеров полей структуры;

Рассмотрим пример выделения памяти под структуру:

```
struct structA {
        char cA;
        char sA[2];
        floatfA;
};
```

При создании переменной структурного типа:

```
structAs1;
```

будет выделено 7 байтов. Элементы структуры будут размещены в памяти в следующем порядке:

char cA	char sA[2]		floatfA			
1	2	3	4	5	6	7

Рассмотрим пример выделения памяти под структуру:

```
struct structB {
        int i1:2;
        int i2:3;
        int :6;
unsigned int i3:4;
};
```

При создании переменной структурного типа:

```
structB s2;
```

будет выделено 2 байта. Элементы структуры будут размещены в памяти в следующем порядке:

Поля		i3				Не доступны						i2			i1	
Биты	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Для целочисленных значений, предусматривающих наличие знакового разряда (например, int), старший левый бит из общего

числа битов, выделяемых под данное битовое поле, интерпретируется как знак. Например, битовое значение 11 для поля i1 будет восприниматься как -1, а значение 11 для поля i3 - как 3.

Доступ к элементам структуры

Элементы структуры могут иметь модификаторы доступа: public, private и protected. По умолчанию все элементы структуры объявляются как общедоступные (public). Забегая вперед, следует сказать, что все члены класса по умолчанию объявляются как защищенные (private).

Для обращения к отдельным элементам структуры используются операторы: . и ->.

Доступ к элементам структуры может иметь следующее формальное описание:

```
переменная_структурного_типа.элемент_структуры=значение
;
имя_структурного_типа
*указатель_структуры=&переменная_структурного_типа;
указатель_структуры->элемент_структуры=значение;
```

Например:

```
struct structA {
  char c1;
  char s1[4];
  float f1;} aS1,
// aS1 - переменная структурного типа
  *prtaS1=&aS1;
    // prtaS1 - указатель на структуру aS1
structstructB {
struct structA aS2;
    // Вложеннаяструктура
} bS1,*prtbS1=&bS1;
aS1.c1= 'E';
    // Доступ к элементу c1 структуры aS1
prtaS1->c1= 'E';
    // Доступ к элементу c1 через
```

```
   // указатель prtaS1
(*prtaS1).c1= 'E';
   // Доступ к элементу c1
(prtbS1->aS2).c1='E';
   // Доступ к элементу вложенной структуры
```

Доступ к элементу массива структурного типа имеет следующий формальный синтаксис:

имя_массива[индекс_элемента_массива].элемент_структуры

При использовании указателей на массив структур следует сначала присвоить указателю адрес первого элемента массива, а затем реализовывать доступ к элементам массива, изменяя этот указатель адреса.

Например:

```
struct structA {
   int i; char c;} sA[4], *psA;
psA=&sA[0];
…
cout<<psA->i;
// Доступ к первому элементу массива
  // структур
  // Переход ко второму элементу массива
psA++;
  // Эквивалентно записи:  psA=&sA[1];
cout<<psA->i;
```

Передача структур в качестве параметров

Переменные структурного типа и элементы структуры можно передавать в функции в качестве параметров.

Передача параметров может выполняться:

- по ссылке или указателю;
- по значению.

При передаче параметра по указателю передается только указатель на структуру, при передаче по значению в стек копируется все содержание структуры.

Например:

```
struct structA {
   int i; char c;} sA, *psA=&sA;
void F1(struct structA sA);
// Передача параметров по значению
void F2(struct structA *psA);
   // Передача параметров по указателю
void F3(struct structA &sA);
   // Передача параметров по ссылке
…
voidF2(structstructA *psA) {
   psA->i =10; }
   // Доступ к элементу структуры
```

При большой вложенности вызовов и использовании большого числа структур или их значительных размерах вызов по значению может привести к переполнению стека.

Функция может возвращать значение структурного типа или типа указателя на структуру.

Например:

```
struct structA { int i; char c;};
struct structA Function3(void);
   // Функция возвращает значение
   // структурного типа
struct structA *Function4(void);
   // Функция возвращает указатель
   // на структуру
```

3.2 Объединения

Объединение позволяет размещать в одном месте памяти данные, доступ к которым реализуется через переменные разных типов.

Использование объединений значительно экономит память, выделяемую под объекты.

При создании переменной типа объединение память под все элементы объединения выделяется исходя из размера наибольшего его элемента. В каждый отдельный момент времени объединение используется для доступа только к одному элементу данных, входящих в объединение.

Так, компилятор C++ выделит 4 байта под следующее объединение:

```
union unionA {
   char ch1;
   float f1;
} a1;
```

<table>
<tr><td></td><td colspan="4">Элементы объединения</td></tr>
<tr><td rowspan="4">Количество занимаемых байтов</td><td colspan="4">char ch1</td></tr>
<tr><td>1</td><td colspan="3"></td></tr>
<tr><td colspan="4">float f1</td></tr>
<tr><td>1</td><td>2</td><td>3</td><td>4</td></tr>
</table>

Объединения, как и структуры, могут содержать битовые поля.

Инициализировать объединение при его объявлении можно только заданием значения первого элемента объединения.

Например:

```
union unionA {
   char ch1;
  floatf1;} a1={ 'M' };
```

Доступ к элементам объединения, аналогично доступу к элементам структур, выполняется с помощью операторов . и ->.

Например:

```
union TypeNum
{
  int    i;
  long  l;
  float  f;
};
  union TypeNum vNum = { 1 };
// Инициализация первого элемента объединения i = 1
cout<< vNum.i;
  vNum.f = 4.13;
cout<< vNum.f;
```

Элементы объединения не могут иметь модификаторов доступа и всегда реализуются как общедоступные (public).

3.3 Перечисления

Перечисление, или перечислимый тип определяет множество, состоящее из значений, указанных через запятую в фигурных скобках.

Перечисление задает для каждого мнемонического названия в указываемом множестве свой индекс.

Перечисление может иметь следующее формальное описание:

enum имя_типа {список_значений} список_объявляемых_переменных;

enum имя_типа список_объявляемых_переменных;

enum (список_элемент=значение);

Перечислимый тип описывает множество, состоящее из элементов-констант, иногда называемых нумераторами или именованными константами.

Значение каждого нумератора определяется как значение типа int. По умолчанию первый нумератор определяется значением 0, второй - значением 1 и т.д. Для инициализации значений нумератора не с 0, а с другого целочисленного значения, следует присвоить это значение первому элементу списка значений перечислимого типа.

Например:

```
// Создание перечисления
enum eDay{sn, mn, ts, wd, th, fr, st} day1;
   // переменная day1 будет принимать
   // значения в диапазоне от 0 до 6
day1=st;
   // day1 - переменная перечислимого типа
int i1=sn;
   // i1 будетравно 0
day1= eDay(0);
   // eDay(0) равно значению sn
enum(color1=255);
   // Объявление перечисления, определяющего
   // именованную целую константу color1
int icolor=color1;
enum eDay2{sn=1, mn, ts, wd, th, fr, st} day2;
```

// переменная day2 будет принимать
// значения в диапазоне от 1 до 7

Для перечислимого типа существует понятие диапазона значений, определяемого как диапазон целочисленных значений, которые может принимать переменная данного перечислимого типа.

Для перечислимого типа можно создавать указатели.

4. Задания для выполнения

№ ВАРИАНТА	ЗАДАНИЕ
Вариант 1	Создайте структуры, содержащую информацию о студентах группы. Включите в нее: фамилию, имя, отчество, возраст и основной изучаемый иностранный язык нескольких студентов. Найдите самого молодого студента.
Вариант 2	Создайте структуры, содержащую информацию о студентах группы. Включите в нее: фамилию, имя, отчество, возраст и основной изучаемый иностранный язык нескольких студентов. Найдите самого старшего студента.
Вариант 3	Создайте структуры, содержащую информацию об адресах студентов группы. Включите в нее: фамилию, город, улицу, дом и квартиру проживания. Упорядочите студентов в порядке возрастания номера дома.
Вариант 4	Создайте структуры, содержащую информацию об адресах студентов группы. Включите в нее: фамилию, город, улицу, дом и квартиру проживания. Упорядочите студентов в порядке убывания номеров квартиры.
Вариант 5	Создайте структуры, содержащую информацию о студентах группы. Включите в нее: фамилию, имя, отчество, возраст и основной изучаемый иностранный язык нескольких студентов. Упорядочите студентов по алфавиту имен.
Вариант 6	Создайте структуру, содержащую информацию о

	студентах группы. В структуру включите: фамилию, имя, отчество, номер зачетной книжки. Упорядочите студентов по номерам зачетной книжки.
Вариант 7	Создайте структуру, содержащую информацию о студентах группы. В структуру включите: фамилию, имя, отчество, номер зачетной книжки, средний балл по итогам семестра. Определите студента с максимальным средним баллом.
Вариант 8	Создайте структуру, содержащую информацию о студентах группы. В структуру включите: фамилию, имя, отчество, номер зачетной книжки, средний бал по итогам семестра. Определите студента с минимальным средним баллом.
Вариант 9	Создайте структуру, содержащую информацию о студентах группы. В структуру включите: фамилию, имя, отчество, номер зачетной книжки, средний бал по итогам семестра. Упорядочите студентов в порядке возрастания среднего балла.
Вариант 10	Создайте структуру, содержащую информацию о студентах группы. В структуру включите: фамилию, имя, отчество, номер зачетной книжки, средний бал по итогам семестра. Упорядочите студентов в порядке убывания среднего балла.

Контрольные вопросы:

1. Что такое структура?
2. Что такое объединение?
3. Что такое битовое поле?
4. Чем отличается структура от объединения?
5. Какой модификатор доступа имеют компоненты структуры по умолчанию?
6. Чем отличается передача структуры в качестве параметра функции по указателю от передачи по значению?
7. Какое значение по умолчанию имеет первый нумератор перечислимого типа?

Классы и объекты

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

<u>Основное содержание работы.</u>

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

Краткие теоретические сведения

Класс

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

С точки зрения синтаксиса, класс в С++ – 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 – установитьдату
void get(int*,int*,int*); // метод – получитьдату
void next(); // метод – установить следующую дату
```

```
void print(); // метод – вывести дату};
struct class complex // комплексноечисло
{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” 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 поумолчанию
public:
double real(){return re;}
double imag(){return im;}
void set(double x,double y){re = x; im = y;}};
```

Конструктор

Недостатком рассмотренных ранее классов является отсутствие автоматической инициализации создаваемых объектов. Для каждого вновь создаваемого объекта необходимо было вызвать функцию типа set (как для класса complex) либо явным образом присваивать значения данным объекта. Однако для инициализации объектов класса в его определение можно явно

включить специальную компонентную функцию, называемую **конструктором**. Формат определения конструктора следующий:

имя_класса(список_форм_параметров){операторы_тела_конструктора}

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

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:

1. Для конструктора не определяется тип возвращаемого значения. Даже тип void не допустим.
2. Указатель на конструктор не может быть определен, и соответственно нельзя получить адрес конструктора.
3. Конструкторы не наследуются.
4. Конструкторы не могут быть описаны с ключевыми словами virtual, static, const, mutuable, valatile.

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

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

имя_класса имя_объекта (фактические_параметры);

имя_класса (фактические_параметры);

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

complex ss (5.9,0.15);

Вторая форма вызова приводит к созданию объекта без имени:

complex ss = complex (5.9,0.15);

Существуют два способа инициализации данных объекта с помощью конструктора. Ранее мы рассматривали первый способ, а именно, передача значений параметров в тело конструктора. Второй способ предусматривает применение списка инициализаторов данного класса. Этот список помещается между списком параметров и телом конструктора. Каждый 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’;}
// создаетобъектпозаданнойстроке
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, вклассе 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 А 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.

Примеры

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&); // конструкторкопирования
~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.

<u>Примеры.</u>

а) массив студентов размещается в статической памяти

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

ит.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,

1. cpp-файл с реализацией класса,
2. 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.

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.

Контрольные вопросы

1. Для чего служит конструктор?
2. Может ли в классе быть несколько конструкторов?
3. Чем должны отличаться различные конструкторы одного и того же класса?
4. Для чего служит деструктор класса?
5. Имеет ли деструктор параметры?
6. В каком случае в тело деструктора включается оператор delete?
7. Какие сообщения и в какой последовательности будут выведены на монитор в следующем примере?

```
class Alpha {

public:

int x, y;

Alpha(){cout<<"Constructor #1"<<endl;}

Alpha(int _m){cout<<"Constructor #2"<<endl; x=y=_m; }

~Alpha(){cout<<"Destructor "<<endl;}
```

```
};
void main()
{
Alpha a1;
a1.x=1;
Alpha *a2;
a2=new Alpha;
Alpha a3[2];
Alpha a4(4);
Alpha a5=a1;
Alpha a6(a1);
cout<<a5.x<<endl<<a6.x<<endl;
}
```

8. Определен следующий класс.

```
class Alhpa { public: int abc; };
```

Запишите обращения к компоненте abc с использованием точки и стрелки.

9. Что такое агрегация классов и как она изображается на диаграммах?

10. Чем отличается строгая агрегация от нестрогой?

Наследование и виртуальные функции

Цель работы: ознакомление с отношением наследования, виртуальными функциями, абстрактными классами.

Основные сведения

Базовый и производные классы

Наследование – механизм языка, позволяющий написать новый класс на основе уже существующего (базового, родительского) класса. Формат определения производного класса следующий:

тип_класса имя_производного_класса : список[
модификатор_доступа имя_базового_класса]
{определение_производного_класса};

В объекте производного класса наряду с собственными полями создаются поля, имя и тип которых определены в базовом классе, а сам объект производного класса получает доступ к методам базового. В этом, собственно, и заключается реализация механизма наследования.

Использование наследования позволяет строить иерархии классов.

При создании объектов производного класса сначала автоматически вызываются конструкторы базовых классов согласно списку базовых классов в объявлении производного класса, а затем конструктор производного класса. Объекты разрушаются в порядке, обратном их созданию, т.е. вначале вызывается деструктор производного класса, а затем базового.

Один класс может быть базовым для нескольких производных, производный класс может быть, в свою очередь, базовым для какого-либо класса и т.д. Если производный класс имеет несколько базовых, то такое наследование называется множественным.

Если доступ к собственным компонентам производных классов определяется обычным образом, то на доступ к наследуемым компонентам влияет, во-первых, атрибут доступа в базовом классе и, во–вторых, модификатор доступа (public / private), указанный перед именем базового класса в конструкции определения производного класса. Общепринятое правило - поля, базового класса определяются как защищенные (protected).

На диаграммах отношение наследования изображается сплошной линией, начинающейся у производного класса и заканчивающейся стрелкой-пустым треугольником у базового класса.

При создании иерархии классов, включающей базовый и производные классы, следует придерживаться принципа "это есть" (is a), выделяя общие черты классов и инкапсулируя их в базовом. Положим, например, что нам надо создать классы Car (автомобиль) и Lorry (грузовик). Что общего у этих классов и чем они разнятся? На самом деле у них много общих черт и много различий, но для упрощения примем следующие формулировки:

- грузовик есть средство передвижения, имеющее цену и год выпуска, а также характеризующееся грузоподъемностью;
- автомобиль есть средство передвижения, имеющее цену и год выпуска, а также характеризующееся скоростью.

Создадим базовый класс Vehicle. В нем определим 2 целых поля для цены и года и методы для установки и возвращения значений этих полей. В производных классах Car (автомобиль) и Lorry (грузовик) определим по одному полю и соответствующие методы.

Листинг 2.1

```
//Базовый класс
classVehicle{
protected:
intprice;//цена
intyear; //год выпуска
public:
//Устанавливаем цену и год выпуска
void setVehicle(int _price, int _year)
{price=_price; year=_year;}
int getPrice()const{return price;}
int getYear()const{return year;}
};
};
//Производныйкласс
class Lorry:public Vehicle{
intcapacity; // грузоподъемность
```

```
public:
//Устанавливаем и возвращаем грузоподъемность
void setCapacity(int _capacity)
{capacity=_capacity;}
int getCapacity()const {return capacity;}
};
// Производныйкласс
class Car: public Vehicle{
intspeed; // скорость
public:
//устанавливаем и возвращаем скорость
void setSpeed(int _speed){speed=_speed;}
int getSpeed()const {return capacity;}
};
```

Собственно, все. Теперь в main можно создавать одиночные объекты Lorry и Car, массивы и задавать их параметры, например, так:

```
Lorry lo;
lo. setVehicle(10000, 2012);
lo. setCapacity(5000);
Car car1[4];
car1[0]. setVehicle(15000, 2011); car1[0]. setSpeed(200);
car1[1]. setVehicle(....); car1[1]. setSpeed(...);
Lorry* lor=new Lorry[3];
. . .
```

Предположим теперь, что нам надо добавить к программе автобусы. Все просто: автобус есть средство передвижения для перевозки пассажиров – добавляем класс Bus с полем, например, "количество пассажиров". При этом уже имеющиеся классы не трогаем; их можно даже не перекомпилировать. Таким образом, расширение программы происходит гораздо проще, чем без использования наследования. Это обстоятельство является одной из главных причин применения наследования.

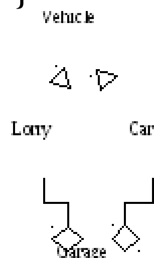
Положим, что в при использовании разработанных классов мы всегда будем иметь дело с массивами их объектов, которые надо организовать и обрабатывать. Чтобы не заставлять клиента (в данном случае main) заниматься этим, введем класс Garage,

включив в него разработанные нами типы, сформировав массивы и определив методы доступа к ним.

Вот как он может выглядеть (продолжаем программу):

```
class Garage{
int n, m; // Количество грузовиков и автомобилей в гараже
// Указатели на соответствующие классы
Lorry *lor;
Car *car;
public:
//Конструкторы
Garage(int _n, int _m)// конструктор для создания обоих видов транспорта
{ n=_n; lor=new Lorry[n];
m=_m; car=new Car[m];
}
// Ввод данных для автомобилей
void setCars()
{ int _price, _year, _speed;
for(int i=0; i<m; i++)
{ cout<<"Input values price, year, speed for "<<i+1<< " Car" <<endl;
cin>>_price>>_year>>_speed;
car[i].setVehicle(_price,_year);
car[i].setSpeed(_speed);
}
//Вводданныхдлягрузовиков
void setLorries()
{ int _price, _year, _capacity;
for(int i=0; i<n; i++)
{ cout<<"Price, year, capacity for "<<i+1<< " Lorry"<<endl;
cin>>_price>>_year>>_capacity;
lor[i].setVehicle(_price,_year);
lor[i].setCapacity(_capacity);
}
}
//Выводпараметровгрузовиков
void getLorries()
```

```
{ cout<<"Lorries"<<endl;
for(int i=0; i<n; i++)
cout<< lor[i].getPrice()<<" "<<lor[i].getYear()<<" "
<<lor[i].getCapacity()<<endl;
}
//Вывод для авто аналогичен
// Деструктор
~Garage(){ delete[] car;
delete[] lor;
}
};
//Теперь наш клиент приобретает совсем простой вид
int _tmain(int argc, _TCHAR* argv[])
{
Garage g1(2,3); //Объект - массив из 2-х грузовиков и 3-х авто
g1. setLorries(); // Вводданных
g1.setCars();
return 0;
}
```



В классе Garage можно инкапсулировать все требуемые методы работы с массивами объектов: сортировку, ввод-вывод в файл и т.п.

Упрощение клиентской части программы за счет введения некоторого дополнительного класса является содержанием паттерна Фасад (Façade)[2].

Разобранный выше пример иллюстрирует обычный подход к использованию наследования – в каждом производном классе определяются свои собственные методы, работающие с объектами производных классов наряду с методами базового.

Другой подход предполагает использование в иерархии классов виртуальных функций с целью получения общего интерфейса иерархии.

Виртуальной называется функция, определенная с атрибутом virtual в базовом классе и имеющая в каждом производном классе ту же самую сигнатуру - тип, имя и список параметров. При этом реализация функции (ее тело) в конкретном производном классе может отличаться от ее реализации в базовом и других производных классах.

```
class Based {
/* ...*/
public:
virtual int fb(){return 3;}
/* ...*/
};
class Derived1:public Based {
/* ...*/
int fb() {return 5;}
/* ...*/
};
class Derived2:public Based {
int fb() {return 7;}
/* . . . */
};
```

Говорят, что если виртуальная функция вызывается из производного класса, то она перекрывает базовую версию.

Виртуальная функция класса может вызываться обычным образом - как член производного или базового класса через его объект. Но тогда никаких преимуществ по сравнению с обычной функцией ее применение не приносит.

Правильный вызов виртуальной функции заключается в последовательном применении следующих 2-х инструкций:

указатель_ на_базовый_ класс=указатель_на__объект_производного_ класса;

указатель_на базовый_класс ◊вызов_виртуальной_функции;

(Инструкции в тексте могут быть разделены. Заметьте, что в первой инструкции никакого преобразования типа указателей не нужно.)

Для нашего примера вызов функции fb из производного класса:

```
Based* ptr=new Derived1;
ptr->fb(); // возвращает 5
. . .
ptr=new Derived2;
ptr->fb();// возвращает 7
```

Виртуальная функция в базовом классе может быть равна 0 (чистая функция). В нашем случае

```
virtual int fb( )=0;
```

Класс с чистой виртуальной функцией называется абстрактным; объект такого класса создать нельзя, только указатель на объект. Мало того, если в каком-либо производном классе эта функция отсутствует, то от такого класса тоже нельзя создавать объекты.

При наличии виртуальных функций деструктор в базовом классе должен быть виртуальным.

Ниже приводится программа, аналогичная листингу 2.1, но с некоторыми упрощениями.

Листинг 2.2

```
class Vehicle{
protected:
int price;//цена
intyear; //год выпуска
public:
//Чистые виртуальные функции
virtual void setVehicle()=0;
virtual void getVehicle()=0;
//Виртуальныйдеструктор
virtual ~ Vehicle(){}
};
class Lorry: public Vehicle{
intcapacity;
public:
//Для упрощения делаем параметры любого грузовика одними и теми же
void setVehicle(){cout<<"setLorry"<<endl;price=40000; year=5;capacity=6;}
```

```
void getVehicle(){cout<<"Lorry"<<endl<<price<<" "<<year<<"
"<<capacity<<endl;}
~Lorry(){}
};
class Car:public Vehicle{
intspeed;
public:
//То же для авто
void setVehicle(){cout<<"setCar"<<endl; price=10000;
year=2;speed=3;}
void getVehicle(){cout<<"Car"<<endl<<price<<" "<<year<<"
"<<speed<<endl;}
~Car(){}
};
classGarage
{
public:
/*Для упрощения количество грузовиков и авто в гараже
одинаковы и заданы константой */
staticconstintn=2;
//Массивы указателей на производные классы
Lorry*lor[n];
Car*car[n];
//Указатель на базовый класс
Vehicle*vec;
public:
Garage()
{
//Инициализация указателей на производные классы
for (int i=0; i<n; i++)
{ car[i]=new Car;
lor[i]= new Lorry;}
}
// Ввод и вывод данных для машин с вызовом виртуальных
функций
void setVehicle(){
for(int i=0; i<n; i++)
```

```
{ vec=car[i]; //Автомобили
vec->setVehicle();
vec=lor[i]; //Грузовики
vec->setVehicle();
}
}
void getVehicle(){
for(int i=0; i<n; i++)
{ vec=car[i];
vec->getVehicle();
vec=lor[i];
vec->getVecicle();
}
}
~Garage()
{
for (int i=0; i<n;i++)
{delete car[i];
delete lor[i];
}
}
};
int _tmain(int argc, _TCHAR* argv[])
{ Garage g;
g.setVehicle();
g.getVehicle();
}
```

Что мы получили, используя виртуальные функции? Возможность доступа к производным классам с использованием одних и тех же функций, т.е. получили общий интерфейс иерархии. Естественно, что при добавлении третьего производного класса интерфейс должен остаться тем же.

Класс, содержащей виртуальный метод, называют полиморфным классом. В нашем случае это означает, что функции setVehicle, getVehicle в разных точках программы действуют по-разному.

Механизм виртуальных функций правильно работает только для указателей и ссылок на объекты. Полиморфизм проявляется только тогда, когда объект производного класса адресуется косвенно, через указатель или ссылку на базовый. Использование самого объекта базового класса не сохраняет идентификацию типа производного класса.

Заметим, что функции Garage::getVehicle, Garage::setVehicle находятся вне иерархии и не имеют никакого отношения к виртуальным. Их можно было назвать по-другому.

Существенное замечание. Наряду с общими для всех классов виртуальными функциями в производном классе могут быть определены собственные функции (и поля). Но помните, что к ним невозможно обратиться через указатель на базовый класс, ибо последний работает только с теми компонентами иерархии, которые определены в базовом классе.

Варианты задания

Следующие варианты предполагают необходимость создания фасадного класса, в который инкапсулированы массивы объектов в динамической памяти и методы ввода и вывода параметров, и другие методы согласно заданию.

Базовый класс иерархии может и не быть абстрактным, но должен содержать хотя бы одну виртуальную функцию. Если есть необходимость, то в производном классе можно объявлять дополнительные компоненты.

1. Создать базовый класс CList (линейный однонаправленный список) с полями: указатели на следующий элемент; информационная часть – целое число. В производных классах – CQueue (очередь) и CStack (стек) – должны быть определены методы вставки и удаления узла в соответствии с дисциплиной обслуживания соответствующего класса.

2. Описать класс CString (Строка) и производные CStringBit (Битовая_строка) и CStringHex (Шестнадцетиричная_строка). (Описание класса см. вариант 9 в 1-й работе.) Строки первого подкласса могут содержать только двоичные символы, второго только шестнадцетиричные. Разработать методы ввода данных с проверкой допустимых символов. Содержимое строк рассматривается как двоичное или

шестнадцетиричное число без знака. Разработать операции сравнения двух строк, сложения и вычитания. Фасадный класс - Text.

3. Создать класс ColoredPoint. На его основе создать производные классы Line и PolyLine (многоугольник). Все классы должны иметь методы установки и получения значений всех координат, а также изменения цвета и получения текущего цвета. Фасадный класс - Picture.

4. Создать класс Figure. На его основе реализовать классы Rectangle (прямоугольник), Circle (круг) и Trapeze (трапеция) с возможностью вычисления площади, центра тяжести и периметра. Фасадный класс - Picture.

5. Создать класс Number с виртуальными методами, реализующими арифметические операции. На его основе реализовать классы Integer и Real.Операции в классах не должны быть одинаковыми. Фасадный класс - Calculus.

6. Создать класс Body. На его основе реализовать классы Parallelepiped (прямоугольный параллелепипед), Cone (конус) и Ball (шар) с возможностью вычисления площади поверхности и объема. Фасадный класс - Series.

7. Создать класс Currency для работы с денежными суммами. Определить в нем методы перевода в рубли. На его основе реализовать классы Dollar, Euro и Pound (фунт стерлингов) с возможностью пересчета в центы и пенсы соответственно. Фасадный класс – Purse.

8. Создать класс Triangle (треугольник), задав в нем длину двух сторон, угол между ними, методы вычисления площади и периметра. На его основе создать классы, описывающие равносторонний, равнобедренный и прямоугольный треугольники со своими методами вычисления площади и периметра. Фасадный класс - Picture.

9. Создать класс Solution (решение) с методами вычисления корней уравнения и вывода на экран. На его основе реализовать классы Linear (линейное уравнение) и Square (квадратное уравнение). Фасадный класс - Series.

10. Создать класс Function (функция) с виртуальными методами вычисления значения функции y = f(x) в заданной точке

x и вывода результата на консоль. На его основе определить классы Ellipse, Hiperbola и Parabola, в которых реализуются соответствующие математические зависимости. В фасадном классе Series создаются массивы для хранения нескольких последовательных значений y.

11. Создать класс Triad (тройка) с виртуальными методами увеличения на 1. На его основе реализовать классы Date (дата) и Time (время). В фасадном классе Memories, создать массив пар (дата-время) объектов этих классов в динамической памяти. Предусмотреть возможность выборки самого раннего и самого позднего событий.

12. Описать класс Element (элемент логической схемы), задав в нем числовой идентификатор, количество входов, идентификаторы присоединенных к нему элементов (до 10) и двоичные значения на входах и выходе. На его основе реализовать классы AND и OR – двоичные вентили, которые могут иметь различное количество входов и один выход и реализуют логическое умножение и сложение соответственно. В фасадном классе Scheme создать массивы вентилей.

13. Описать класс Element (элемент логической схемы) задав в нем символьный идентификатор, количество входов, идентификаторы присоединенных к нему элементов (до 10) и двоичные значения на входах и выходе. На его основе реализовать классы AND_NOT и OR_NOT — двоичные вентили, которые могут иметь различное количество входов и один выход и реализуют логическое умножение с отрицанием и сложение с отрицанием соответственно. В фасадном классе Scheme создать массивы вентилей.

14. Описать класс Trigger (триггер), задав в нем идентификатор и двоичные значения на входах и выходах. На его основе реализовать классы RS и JK, представляющие собой триггеры соответствующего типа. В фасадном классе Register предусмотреть общий сброс и установку значений произвольного триггера по заданным значениям входов.

15. Создать класс Progression (прогрессия) с виртуальными методами вычисления заданного элемента и суммы прогрессии. На

его основе реализовать классы Linear (арифметическая) и Exponential (геометрическая). Фасадный класс - Series.

16. Создать класс Pair (пара значений) с виртуальными методами, реализующими арифметические операции сложения и вычитания. На его основе реализовать классы Fractional (дробное) и LongLong (длинное целое).В классе Fractional вещественное число представляется в виде двух целых, в которых хранятся целая и дробная часть числа соответственно. В классе LongLong длинное целое число хранится в двух целых полях в виде старшей и младшей части. Фасадный класс - Series.

17. Создать класс Integer (целое) с символьным идентификатором, виртуальными методами, реализующими арифметические операции, и методом вывода на экран. На его основе реализовать классы Decimal (десятичное) и Binary (двоичное). Число представить в виде массива цифр. В фасадном классе Series предусмотреть возможность вывода значений и идентификаторов всех объектов списка и вывода общей суммы всех десятичных значений.

18. Создать класс Sorting (сортировка) с массивом, задаваемым с помощью new, и виртуальными методами ввода элементов, сортировки и вывода на экран. На его основе реализовать классы Choice (метод выбора) и Quick (быстрая сортировка). Фасадный класс - Series.

19. Создать класс Pair (пара значений) с виртуальными методами, реализующими арифметические операции, и методом вывода на экран. На его основе реализовать классы Money (деньги) и Complex (комплексное число). В классе Money денежная сумма представляется в виде двух целых, в которых хранятся рубли и копейки соответственно. При выводе части числа снабжаются словами «руб.» и «коп.». В классе Complex предусмотреть при выводе символ мнимой части (i). Фасадный класс - Series.

20. Создать класс Worker с полями, описывающими должность, фамилию работника и его обязанности, а также фамилию его руководителя. На его основе реализовать классы Manager (руководитель проекта), Developer (разработчик) и Coder программист) с соответствующими методами. Фасадный класс Group, в котором предусмотрена возможность выборки по

фамилии с выводом всего дерева подчиненных. (Роль руководителя проекта заключается в полной ответственности за успешное планирование, выполнение и завершение проекта. Разработчик - поддержка существующего продукта, разработка новых функциональных возможностей и новых компонентов, выбор программных средств.)

Контрольные вопросы

1. В чём заключается наследование одного класса другому?
2. В чем разница в организации наследования полей и методов?
3. Определены 2 класса:

```
class Based{public: int x;};
class Derived :public Based{};
```

Какое значение выводится на консоль при применении следующего кода:

```
Based b1;
b1.x=3;
Derived d1;
d1.x=4;
cout<<b1.x;
```

3. Удачной ли является иерархия классов, при которой некоторый класс X является производным от большого количества классов с большим числом полей в каждом (A⇓B ⇓ C ⇓…⇓ X)? Какая существует альтернатива наследованию?
4. Каким образом производится управление доступом к унаследованным компонентам производных классов? Есть ли в представленном фрагменте программы ошибки и какое сообщение выдаст компилятор? Как исправить ошибку?

```
class Base {
int x;
public:
int y;
void setX(int n){x=n;}
void showX() const{ cout<<x<<endl;}
};
class Derived: public Base {
```

```
public:
void setY(int n){y=n;}
void show_sum() const{ cout<<x+y<<endl; }
};
int _tmain(int argc, _TCHAR* argv[])
{
Derived d1;
d1.show_sum();
return 0;
}
```

5. Есть ли ошибки в нижеследующих объявлениях

```
// класс Shape абстрактный
Shape sh;
Shape *psh;
Shape *psh1=new Shape();
```

6. Что из себя представляет виртуальная функция и как она должна вызываться?

7. Как создать общий интерфейс иерархии классов?

Полиморфизм. Перегрузка операций.

Цель работы: освоить базовую концепцию объектно-ориентированного программирования – полиморфизм на примере перегрузки операций.

Теоретические сведения

Одним из базовых принципов объектно-ориентированного стиля программирования является полиморфизм. Это свойство, которое позволяет одно и то же имя использовать для решения двух или более назначений. Целью полиморфизма, применительно к ООП, является использование одного имени для задания общих для класса действий. Выполнение каждого конкретного действия будет определяться типом данных.

Одни и те же знаки операций (из списка уже существующих) могут быть определены и для объектов, введенных пользователем. Например, описав класс «вектор в трехмерном пространстве», не нужно придумывать имя функции для выполнения операции сложения над векторами, а можно переопределить смысл имеющейся операции «+» таким образом, что сложение двух векторов будет записываться в привычной форме: A + B.

Транслятор также может различать функции не только по именам, но и по типам аргументов, поэтому вполне допустимым является наличие, например, двух следующих функций:

int sqr (int x)
{return (x*x)}
float sqr (float x)
{return (x*x)}

Код какой конкретно функции будет использован при вызове, зависит от типа фактического аргумента. Перегруженные функции хорошо использовать при введении новых данных. Описав тип COMPLEX для работы с комплексными числами, можно составить функцию ***pow()*** для возведения в степень комплексного числа, а назвать ее так же, как и функцию, работающую с вещественными числами.

Формат перегрузки операторов:

[<класс>] operator <оператор>(<параметры>)
{ <выражения>;

return(<выражения>);
}

Пример. Переопределим некоторые операции ("+", "-", "^", "%", "*") для нахождения суммы, разности, скалярного произведения, векторного произведения двух заданных векторов, а также произведение каждого из них на скаляр.

```
#define g goto
#define o odin
#define d dva
#define t tri
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <conio.h>
struct St //структуравектор
{int s[4];};
St operator + (St a,St b) //перегрузкаоперации +
{static St sum; int i;
for (i=1;i<=3;i++) {sum.s[i] = a.s[i]+b.s[i];}
return sum;
}
St operator - (St a, St b) //перегрузкаоперации –
{static St sum; int i;
for (i=1;i<=3;i++){sum.s[i]=a.s[i]-b.s[i];}
return sum;
}
int operator ^ (St a,St b) //перегрузкаоперации ^
{int i, sum = 0;
for (i=1; i<=3; i++) {sum += a.s[i] * b.s[i];}
return sum;
}
St operator % (St a,St b) //перегрузкаоперации %
{static St sum; int i;
sum.s[1] = a.s[2] * b.s[3] - a.s[3] * b.s[2];
sum.s[2] = -(a.s[1] * b.s[3] - a.s[3] * b.s[1]);
sum.s[3] = a.s[1] * b.s[2] - a.s[2] * b.s[1];
return sum;
```

```
}
St operator * (St a,int b) //перегрузкаоперации *
{static St sum;int i;
for (i=1;i<=3;i++){sum.s[i]=a.s[i]*b;}
return sum;
}
void main() //основная программа
{ static St a,b,c;int j,res,con;char vibor;
clrscr();
randomize();
printf("ВВЕДИТЕКООРДИНАТЫВЕКТОРА a ");
scanf("%d %d %d",&a.s[1],&a.s[2],&a.s[3]);
printf("ВВЕДИТЕ КООРДИНАТЫ ВЕКТОРА b ");
scanf("%d %d %d",&b.s[1],&b.s[2],&b.s[3]);
ddd: printf("a=(%d,%d,%d)\n",a.s[1],a.s[2],a.s[3]);
printf("b=(%d,%d,%d)\n",b.s[1],b.s[2],b.s[3]);
printf("ОПЕРАЦИЯ");
printf("=>1: a+b; 2: a-b; 3:(a,b); 4:[axb]; 5:const*a; 6:const*b;
0:EXIT\n");
vibor=getch();
switch (vibor)
{ case '1' : {c=a+b;g o;}
case '2' : {c=a-b;g o;}
case '3' : {res=a^b;g d;}
case '4' : {c=a%b;g o;}
case '5' : { printf("ВВЕДИТЕconst ");
scanf("%d",&con);
c=a*con;g o;
}
case '6' : { printf("ВВЕДИТЕconst ");
scanf("%d",&con);
c = b*con;g o;}
case '0' : {g t;}
}
odin: printf("результатс = (%d,%d,%d) \n",c.s[1],c.s[2],c.s[3]);g
ddd;
dva: printf("результат %d\n",res);
```

```
g ddd;
tri:
}
```

Аргументы по умолчанию

При объявлении функции в языке C++ для нее можно задать аргументы по умолчанию. Например, объявление ***void fun(int,int=5);***

позволяет вызвать функцию **fun** двумя способами, например ***fun(10)*** и ***fun(10,20)***.

Когда у функции задан только один аргумент, второй аргумент получит значение, заданное по умолчанию. Поэтому вызов fun(10) назначит первому аргументу значение 10, а второму – 5. Второй вызов fun(10,20) назначит аргументам значения 10 и 20.

Пример:

```
#include <iostream.h>
void fun(int, char='%', float=1.25); // Объявлениефункции
void fun(int=1,char,float); // Переобъявлениефункции
void fun(char *); // Объявлениефункции
void main()
{ int ii=10; char cc='*'; float ff=22.33;
fun(ii); // Результат 10 % 1.25
fun(ii,cc); // Результат 10 * 1.25
fun(ii,cc,ff); // Результат 10 * 22.33
fun("HELLO\n"); // Результат HELLO
fun(); // Результат 1 % 1.25
}
void fun (int i,char c,float f) // Описаниефункции
{cout << i << '\t' << c << '\t' << f << '\t' << endl;}
void fun (char *s)
{cout<<s;}
```

Второе объявление

void fun(int=1,char,float);

добавляет в первую функцию значение первого аргумента, заданного по умолчанию. Для обоих объявлений достаточно одного описания. Новую функцию ***fun***, заданную в виде ***void***

fun(char *), можно отличить по числу аргументов, поэтому неоднозначности не возникает.

Если функция имеет аргумент, заданный по умолчанию, то после него могут идти только аргументы, тоже заданные по умолчанию. Например:

int f1(int a, char b = '#', float f); // Неверно

int f2(int a ,char b = '#', float f = 5.28); // Нетошибки

Рассмотрим пример 2:

```
#include <iostream.h>
class my_class
{ int x,y,z;
public :
my_class() {x=y=0;}
float add_mul(int a=0, int b=0, int c=0, char ch='+');
void display() {cout << x << '\t' << y << endl;}
};
float my_class::add_mul(int a,int b,int c,char ch)
{ if (ch=='+') return x=a+b+c;
if (ch=='*') return y=a*b*c;
cout<<"ERROR \n"; return NULL;
}
void main(void)
{ my_class A;
cout<<A.add_mul(5,10)<<endl; // Результат 15
cout<<A.add_mul(2,5,10)<<endl; // Результат 17
cout<<A.add_mul(2,5,10,'*')<<endl; // Результат 100
A.display(); // Результат 17 100
}
```

В задании требуется реализовать классы и операции (функции) для работы с объектами этих классов.

Задания:

1. Для строки символов реализовать операции:

а) сравнение строк (операция ==);

б) удаление из строки заданного символа (операция –).

Кроме того, членом класса сделать функцию с именем strset() для удаления из первой строки всех символов, встречающихся во второй строке.

2. Для строки символов реализовать операции:

а) проверка в строке наличия заданного символа (операция !);

б) перевод указанного символа строки в код ASCII(операция %).

Членом класса сделать функцию с именем strlen() для нахождения суммы кодов ASCII всех символов данной строки.

3. В британском формате дата задается как число/месяц/год. Реализовать с учетом високосных годов:

а) сложение даты и заданного количества дней (операция +);

б) вычитание из даты заданного количества дней (операция –).

Кроме того, членом класса сделать функцию с именем printf() для вывода конечной даты.

4. В британском формате дата задается как число/месяц/год. Реализовать с учетом високосных годов:

а) определение числа дней, прошедших между двумя датами (операция %);

б) нахождение порядкового номера даты в заданном году (операция /).

Членом класса сделать функцию с именем printf() для вывода конечной даты.

5. Ввести класс для работы с прямоугольной матрицей. Реализовать операции:

а) сложение двух матриц (операция +);

б) умножение двух матриц (операция *).

Членом класса сделать функцию printf() для вывода конечной матрицы и ее модуля.

6. Ввести класс для работы с прямоугольной матрицей. Реализовать операции:

а) проверка наличия заданного числа в заданной матрице (операция !);

б) вычитание двух матриц (операция –).

Членом класса сделать функцию с именем printf() для вывода конечной матрицы и ее модуля.

7. Время задается в формате час/минута/секунда. Реализовать следующие операции(учесть переход через 24 часа):

а) сложение времени и заданного количества секунд (операция +);

б) вычитание из времени заданного количества секунд (операция –).

Членом класса сделать функцию с именем printf() для вывода конечного времени.

8. Время задается в формате час/минута/секунда. Реализовать следующие операции (учесть переход через 24 часа):

а) вычитание из одного момента времени другого (операция –);

б) подсчет числа секунд между двумя моментами времени в пределах одних суток (операция %).

Членом класса сделать функцию printf() для вывода конечного времени.

9. Время задается в формате час/минута/секунда. Реализовать следующие операции (учесть переход через 24 часа):

а) сложение двух моментов времени(операция +);

б) определение времени дня по заданному количеству пройденных секунд(операция /).

Членом класса сделать функцию printf() для вывода конечного времени.

10. Ввести класс для работы с объектом «полином». Реализовать операции:

а) сложение двух полиномов (операция +);

б) умножение двух полиномов (операция *).

Членом класса сделать функцию printf() для вывода конечного полинома.

11. Ввести класс для работы с объектом «полином». Реализовать операции:

а) вычитание двух полиномов (операция –);

б) деление двух полиномов (операция /).

Членом класса сделать функцию printf() для вывода конечного полинома.

12. Ввести класс для работы с объектом «полином». Реализовать операции:

а) умножение полинома на число (операция &);

б) вычисление значения полинома в заданной точке X (операция |).

Членом класса сделать функцию printf() для вывода конечного полинома.

13. Ввести класс для работы с объектом «множество целых чисел». Реализовать следующие операции:

а) объединение двух множеств (операция +);

б) пересечение двух множеств (операция &).

Членом класса сделать функцию printf() для вывода конечного множества.

14. Ввести класс для работы с объектом «множество целых чисел». Реализовать следующие операции:

а) разность двух множеств (операция –);

б) добавление элемента во множество (операция +).

Членом класса сделать функцию printf() для вывода конечного множества.

15. Ввести класс для работы с объектом «множество целых чисел». Реализовать следующие операции:

а) удаление элемента из множества (операция –);

б) проверка наличия заданного элемента в заданном множестве (операция /). Членом класса сделать функцию printf() для вывода конечного множества.

16. Ввести класс работы с объектом «рациональная дробь» вида m/n. Реализовать:

а) сложение дробей (операция +);

б) умножение двух дробей (операция *).

Членом класса сделать функцию printf() для вывода конечной дроби в виде m/n.

17. Ввести класс работы с объектом «рациональная дробь» вида m/n. Реализовать:

а) вычитание дробей (операция –);

б) деление двух дробей (операция /).

Членом класса сделать функцию с именем abs() для приведениея дроби к несократимому виду.

18. Ввести класс работы с объектом «рациональная дробь» вида m/n. Реализовать:

а) сравнение двух дробей (операция ||);

б) возведение в целую положительную степень (операция ^).

Членом класса сделать функцию с именем printf() для вывода конечной дроби в несократимом виде m/n.

19. Комплексное число задается своей вещественной и мнимой частями (например, 5+3i задается парой 5,3). Реализовать:

а) сложение чисел (операция +);

б) произведение двух чисел (операция *).

Членом класса сделать функцию printf() для вывода конечного числа в виде U+iV.

20. Комплексное число задается своей вещественной и мнимой частями (например, 5+3i задается парой 5,3). Реализовать:

а) вычитание чисел (операция –);

б) возведение числа в целую положительную степень (операция ^).

Членом класса сделать функцию printf() для вывода конечного числа в виде U+iV.

Контрольные вопросы

1. Зачем нужен механизм полиморфизма?
2. Что понимается под динамическим полиморфизмом?
3. К какому из видов полиморфизма относится перегрузка оператора?
4. Что такое интерфейс класса?
5. Зачем нужен чисто виртуальный метод? Как он выглядит?
6. Какой класс называется абстрактным?

Библиотека шаблонов STL

Цель работы: изучить правила работы со стандартными шаблонами в C++.

Теоретические сведения

Стандартная библиотека шаблонов (STL) (англ. Standard Template Library) — набор согласованных обобщённых алгоритмов, контейнеров, средств доступа к их содержимому и различных вспомогательных функций в C++.

Стандартная библиотека шаблонов до включения в стандарт C++ была сторонней разработкой, в начале — фирмы HP, а затем SGI. Стандарт языка не называет её «STL», так как эта библиотека стала неотъемлемой частью языка, однако многие люди до сих пор используют это название, чтобы отличать её от остальной части стандартной библиотеки (потоки ввода/вывода (iostream), подраздел Си и др.).

Проект под названием STLPort, основанный на SGI STL, осуществляет постоянное обновление STL, iostream и строковых классов. Некоторые другие проекты также занимаются разработкой частных применений стандартной библиотеки для различных конструкторских задач. Каждый производитель компиляторов C++ обязательно поставляет какую-либо реализацию этой библиотеки, так как она является очень важной частью стандарта и широко используется. Архитектура STL была разработана Александром Степановым и Менгом Ли.

В библиотеке выделяют пять основных компонентов:

- Контейнер (англ. container) — хранение набора объектов в памяти.
- Итератор (англ. iterator) — обеспечение средств доступа к содержимому контейнера.
- Алгоритм (англ. algorithm) — определение вычислительной процедуры.
- Адаптер (англ. adaptor) — адаптация компонентов для обеспечения различного интерфейса.
- Функциональный объект (англ. functor) — сокрытие функции в объекте для использования другими компонентами.

Такое разделение позволяет уменьшить количество компонентов. Например, вместо написания отдельной функции поиска элемента для каждого типа контейнера обеспечивается единственная версия, которая работает с каждым из них, пока соблюдаются основные требования.

Контейнеры

Контейнеры библиотеки STL можно разделить на четыре категории: последовательные, ассоциативные, контейнеры-адаптеры и псевдоконтейнеры.

Таблица 1 - Последовательные контейнеры

vector	C-подобный динамический массив произвольного доступа с автоматическим изменением размера при добавлении/удалении элемента. Существует специализация шаблона vector для типа bool, которая требует меньше памяти за счёт хранения элементов в виде битов, однако она не поддерживает всех возможностей работы с итераторами.
list	Двусвязный список, элементы которого хранятся в произвольных кусках памяти, в отличие от контейнера vector, где элементы хранятся в непрерывной области памяти. Поиск перебором медленнее, чем у вектора из-за большего времени доступа к элементу. В любом месте контейнера вставка и удаление производятся очень быстро.
deque	Дэк. Контейнер похож на vector, но с возможностью быстрой вставки и удаления элементов на обоих концах. Реализован в виде двусвязного списка линейных массивов. С другой стороны, в отличие от vector, дек не гарантирует расположение всех своих элементов в непрерывном участке памяти, что делает невозможным безопасное использование арифметики указателей для доступа к элементам контейнера.

Таблица 2 - Ассоциативные контейнеры

Set	Упорядоченное множество уникальных элементов. При вставке/удалении элементов множества итераторы, указывающие на элементы этого множества, не становятся недействительными. Обеспечивает стандартные операции над множествами типа объединения, пересечения, вычитания. Тип элементов

	множества должен реализовывать оператор сравнения operator< или требуется предоставить функцию-компаратор. Реализован на основе самобалансирующего дерева двоичного поиска.
Multiset	То же что и set, но позволяет хранить повторяющиеся элементы.
Map	Упорядоченный ассоциативный массив пар элементов, состоящих из ключей и соответствующих им значений. Ключи должны быть уникальны. Порядок следования элементов определяется ключами. При этом тип ключа должен реализовывать оператор сравнения operator<, либо требуется предоставить функцию-компаратор.
multimap	То же что и map, но позволяет хранить несколько одинаковых ключей.

Таблица 3 - Контейнеры-адаптеры

stack	Стек — контейнер, в котором добавление и удаление элементов осуществляется с одного конца.
queue	Очередь — контейнер, с одного конца которого можно добавлять элементы, а с другого — вынимать.
priority_queue	Очередь с приоритетом, организованная так, что самый большой элемент всегда стоит на первом месте.

Таблица 4 - Псевдоконтейнеры

bitset	Служит для хранения битовых масок. Похож на vector фиксированного размера. Размер фиксируется тогда, когда объявляется объект bitset. Итераторов в bitset нет. Оптимизирован по размеру памяти.
basic_string	Контейнер, предназначенный для хранения и обработки строк. Хранит в памяти элементы подряд единым блоком, что позволяет организовать быстрый доступ ко всей последовательности. Элементы должны быть простых (фундаментальных) типов данных. Определена конкатенация с помощью +.

valarray	Шаблон служит для хранения числовых массивов и оптимизирован для достижения повышенной вычислительной производительности. В некоторой степени похож на vector, но в нём отсутствует большинство стандартных для контейнеров операций. Определены операции над двумя valarray и над valarray и скаляром (поэлементные). Эти операции возможно эффективно реализовать как на векторных процессорах, так и на скалярных процессорах с блоками SIMD.

В контейнерах для хранения элементов используется семантика передачи объектов по значению. Другими словами, при добавлении контейнер получает копию элемента. Если создание копии нежелательно, то используют контейнер указателей на элементы. Присвоение элементов реализуется с помощью оператора присваивания, а их разрушение происходит с использованием деструктора. В таблице 5 приведены основные требования к элементам в контейнерах:

Таблица 5 - Требования к элементам контейнеров

Метод	Описание	Примечание
Конструктор копии	Создает новый элемент, идентичный старому	Используется при каждой вставке элемента в контейнер
Оператор присваивания	Заменяет содержимое элемента копией исходного элемента	Используется при каждой модификации элемента
Деструктор	Разрушает элемент	Используется при каждом удалении элемента
Конструктор по умолчанию	Создает элемент без аргументов	Применяется только для определенных операций
operator==	Сравнивает два элемента	Используется при выполнении operator== для двух контейнеров
operator <	Определяет, меньше	Используется при

	ли один элемент другого	выполнении operator< для двух контейнеров

Все «полноценные» стандартные контейнеры удовлетворяют определенному набору требований (или соглашений). В приведенной ниже таблице 6 полагается, что C — класс контейнера, содержащий объекты типа T.

Таблица 6

Выражение	Возвращаемый тип	Сложность	Примечание
C::value_type	T	Время компиляции	
C::reference	T	Время компиляции	
C::const_reference		Время компиляции	
C::pointer	Тип указателя, указывающего на C::reference	Время компиляции	Указатель на T
C::iterator	Тип итератора, указывающего на C::reference	Время компиляции	Итератор любого типа, кроме итератора вывода
C::const_iterator	Тип итератора, указывающего на C::const_reference	Время компиляции	Итератор любого типа, кроме итератора вывода
C::size_type	Беззнаковый целочисленный тип	Время компиляции	
C obj;		Постоянная	После: obj.size() == 0
C obj1; obj1 =		Линейная	После: obj1

obj2;			== obj2
C obj; (&obj)->~C();	Результат не используется	Линейная	После: a.size() == 0.
obj.begin()		Постоянная	
obj.end()		Постоянная	
obj1 == obj2	Обратимый в bool	Линейная	
obj1 != obj2	Обратимый в bool	Линейная	
obj.size()	size_type	Зависит от типа	Не рекомендуется применять для проверки, пуст ли контейнер
obj.empty()	Обратимый в bool	Постоянная	
obj1 < obj2	Обратимый в bool	Линейная	
obj1 > obj2	Обратимый в bool	Линейная	
obj1 <= obj2	Обратимый в bool	Линейная	
obj1 >= obj2	Обратимый в bool	Линейная	
obj.swap(obj2)	void	Постоянная	

Итераторы

В библиотеке STL для доступа к элементам в качестве посредника используется обобщённая абстракция, именуемая итератором. Каждый контейнер поддерживает «свой» вид итератора, который представляет собой «модернизированный» интеллектуальный указатель, «знающий» как получить доступ к

элементам конкретного контейнера. Стандарт C++ определяет пять категорий итераторов, описанных в таблице 7:

Таблица 7. Категории итераторов

Категория	**Поддерживаемые операции**	**Примечание**
Входные	operator++, operator*, operator->, конструкторкопии, operator=, operator==, operator!=	Обеспечивают доступ для чтения в одном направлении. Позволяют выполнить присваивание или копирование с помощью оператора присваиваивания и конструктора копии
Выходные	operator++, operator*, конструктор копии	Обеспечивают доступ для записи в одном направлении. Их нельзя сравнивать на равенство.
Однонаправленные	operator++, operator*, operator->, конструкторкопии, конструкторпоумолчанию, operator=, operator==, operator!=	Обеспечивают доступ для чтения и записи в одном направлении. Позволяют выполнить присваивание или копирование с помощью оператора

		присваиваивания и конструктора копии. Их можно сравнивать на равенство.
Двунаправленные	operator++, operator--, operator*, operator->, конструкторкопии, конструкторпоумолчанию, operator=, operator==, operator!=	Поддерживают все функции, описанные для однонаправленных итераторов (см. выше). Кроме того, они позволяют переходить к предыдущему элементу.
Произвольного доступа	operator++, operator--, operator*, operator->, конструкторкопии, конструкторпоумолчанию, operator=, operator==, operator!=, operator+, operator-, operator+=, operator-=, operator<, operator>, operator<=, operator>=, operator[]	Эквивалентны обычным указателям: поддерживают арифметику указателей, синтаксис индексации массивов и все формы сравнения.

Варианты заданий

1. Создать шаблон **vector** для массива данных об авто.
2. Создать шаблон **list** для двусвязного списка данных об авто.
3. Создать шаблон **deque** для учёта данных об очереди авто на заправке.
4. Создать шаблон **set** для построения двух множеств целых чисел и вычисления их пересечения.
5. Создать шаблон **multiset** для подсчета числа вхождений каждого числа во множество целых чисел с повторами.

6. Создать шаблон **map** для исключения повторов среди множества целых чисел.

7. Создать шаблон **multimap** для исключения повторов комбинаций среди множества пар целых чисел.

8. Создать шаблон **stack** для стека вещественных чисел.

9. Создать шаблон **queue** для очереди авто на мойке.

10. Создать шаблон **priority_queue** для очереди заказов, чтобы обслуживать самые большие заказы в первую очередь.

11. Создать шаблон **bitset** для хранения информации о простоте первых 10000 целых чисел.

12. Создать шаблон **basic_string** для хранения фамилий имен и отчеств.

13. Создать шаблон **valarray** для массива данных об авто.

14. Создать шаблон **hash_map** для массива данных об авто.

15. Создать шаблон **unordered_map** для массива данных об авто.

Контрольные вопросы

1. Что такое шаблоны и с какой целью они используются?
2. Какого типа шаблоны используются в программах?
3. Как оформляются шаблоны функций?
4. Какие требования предъявляются к фактическим параметрам шаблонов?
5. Какие преимущества программы обеспечиваются при использовании шаблонов классов?

Список литературы

1. Лафоре, Р. Объектно-ориентированное программирование в C++ [Текст] / Р. Лафоре. - 4-е изд. - СПб. [и др.] : Питер, 2012. – 928 с.

2. Зыков, С.В. Введение в теорию программирования [Электронный ресурс] / С.В. Зыков. – М. : Национальный открытый университет «ИНТУИТ», 2016. – 189 с. – Режим доступа / https://biblioclub.ru/

3. Буч, Г. Объектно-ориентированный анализ и проектирование с примерами и приложениями на С++ [Текст] / Пер. с англ. - 2-е изд. - СПб. : Бином ; СПб. : Невский диалект, 2001. - 560 с.

4. Сорокин, А.А. Объектно-ориентированное программирование [Электронный ресурс] : учебное пособие (курс лекций) / А.А. Сорокин. – Ставрополь : Изд-во СКФУ, 2014. – 174 с. Режим доступа / https://biblioclub.ru/